Savitribai Phule Pune University

(Formerly University of Pune)

T.Y.B.Sc.(Computer Science)

With

Major: Computer Science

(Faculty of Science and Technology)

(For Colleges Affiliated to Savitribai Phule Pune University)

Choice Based Credit System (CBCS) Syllabus Under National Education Policy (NEP)

To be implemented from Academic Year 2026-2027

Title of the Course: B.Sc. (Computer Science)

Savitribai Phule Pune University

(Formerly University of Pune)

T.Y.B.Sc.(Computer Science) Semester V

WORKBOOK

Course Code: CS -3010-MJ-P

(Lab Course based on CS-309-MJ-T)

WORKBOOK

T.Y.B.Sc. (Computer Science)

Database Technologies

Course Type: Major Elective Course Code: CS -3010-MJ-P

Course Title: Lab Course based on CS-309-MJ-T

SEMESTER V

Student Name:			
College:			
Roll No:		**Exam Seat No:**	
Year:		**Division:**	

BOARD OF STUDIES

1. Dr. Patil Ranjeet	2. Dr. Joshi vinayak
3. Dr. Wani Vilas	4. Dr. Mulay Prashant
5. Dr. Sardesai Anjali	6. Dr. Shelar Madhukar
7. Dr. Bharambe Manisha	8. Dr. Deshpande Madhuri
9. Dr. Kardile Vilas	10. Dr. Korpal Ritambhara
11. Dr. Gangurde Rajendra	12. Dr. Manza Ramesh
13. Dr. Bachav Archana	14. Dr. Bhat Shridhar

Co-ordinators

- **Dr. Prashant Mulay**

Annasaheb Magar College, Hadapsar , Pune.
Member, BOS Computer Science, Savitribai Phule Pune University

- **Dr. Sardesai Anjali**

Modern College of Arts, Science and Commerce , Shivajinagar, Pune.
Member, BOS Computer Science, Savitribai Phule Pune University

Editor:

Dr. Reena Shinde	Sinhgad College of Science, Pune

Prepared by:

Ms. Shital Jadhav	Sinhgad College of Science, Pune
Ms. Rupali Pise	Sinhgad College of Science, Pune
Ms. Kajal Londhe	Sinhgad College of Science, Pune

Introduction

About the workbook

This workbook is intended to be used by T.Y.B. Sc (Computer Science) students for the Lab Course based on CS-309-MJ-T (Database Technologies)

Workbook contains assignments on Database Technologies. Database Technologies is major elective subject of computer science curriculum and hands-on laboratory experience is critical to the understanding of theoretical concepts studied as part of this course. Study of any programming language is incomplete without hands on experience of implementing solutions using programming paradigms and verifying them in the lab. This workbook provides rich set of problems covering the basic algorithms as well as numerous computing problems demonstrating the applicability and importance of various data structures and related algorithms.

The objectives of this book are

- Defining the scope of the course
- Bringing uniformity in the way the course is conducted across different colleges
- Continuous assessment of the course
- Bring variation and variety in experiments carried out by different students in a batch
- Providing ready reference for students while working in the lab
- Catering to the need of slow paced as well as fast paced learners

How to use this workbook?

The Database Technologies practical syllabus is divided into five assignments. Each assignment has problems divided into three sets A, B and C.

- Set A is used for implementing the basic algorithms or implementing data structure along with its basic operations. **Set A is mandatory.**
- Set B is used to demonstrate small variations on the implementations carried out in set A to improve its applicability. Depending on the time availability the students should be encouraged to complete set B.
- In Set C, Students should spend additional time either at home or in the Lab and solve these problems so that they get a deeper understanding of the subject.

The Database Technologies practical syllabus is divided into six assignments. For assignment number 1 to assignment number 6 includes three sets A, B, and C. Students should solve any **two Sets** of them compulsory.

Instructions to the students

Please read the following instructions carefully and follow them.

- Students are expected to carry workbook during every practical.
- Students should prepare oneself beforehand for the Assignment by reading the relevant material.

- Instructor will specify which problems to solve in the lab during the allotted slot & student should complete them and get verified by the instructor. However, student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.

- Students will be assessed for each exercise on a scale from 0 to 5
  - Not done 0
  - Incomplete 1
  - Late Complete 2
  - Needs improvement 3
  - Complete 4
  - Well Done 5

Instruction to the Practical In-Charge

- Explain the assignment and related concepts.
- Choose appropriate problems to be solved by students. Set A is mandatory. Choose problems from set B depending on time availability. Discuss set C with students and encourage them to solve the problems by spending additional time in lab or at home.
- Make sure that students follow the instruction as given above.
- You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
- The value should also be entered on assignment completion page of the respective Lab course.

Instructions to the Lab administrator and Exam guidelines

- You have to ensure appropriate hardware and software is made available to each student.
- Do not provide Internet facility in Computer Lab while examination
- Do not provide pen drive facility in Computer Lab while examination.

The operating system and software requirements are as given below:

- Operating system: Linux
- Editor: Any Linux based editor like vi, gedit etc.
- MongoDB, Neo4j

Assignment Completion Sheet

Sr. No.	Assignment Name	Marks	Signature
1.	Design and Representation of Data using Structured, Semi Structured, and JSON Formats		
2.	Installation and Configuration of MongoDB with Database and Collection Creation		
3.	Execution of CRUD Operations and Data Manipulation in MongoDB		
4.	Application of Aggregation Framework and Indexing Techniques in MongoDB		
5.	Modeling of Graph Data using Nodes and Relationships in Neo4j		
6.	Implementation of Graph Queries and Relationship Analysis using Cypher in Neo4j		
Total			
Marks out of total to be converted to out of 15			

This is to certify that **Mr./Ms.** _____
Seat Number _____have successfully and satisfactorily completed and submitted the course work for **T.Y.B.Sc.(CS)** Lab course CS -3010-MJ-P Semester V prescribed by Savitribai Phule Pune University during the academic year _____.

Instructor **Head of Department**

Internal Examiner **External Examiner**

Assignment 1

Design and Representation of Data using Structured, Semi-Structured, and JSON Formats (Total Slots = 2)

Introduction

Data representation is a fundamental concept in database management and information systems. In the modern computing era, data is stored, exchanged, and processed in a variety of formats depending on the application requirements. Broadly, data can be classified into three major categories based on its structure:

- Structured Data
- Semi-Structured Data
- Unstructured Data

MongoDB, a popular NoSQL database, primarily works with semi-structured data in the form of JSON (JavaScript Object Notation) documents. Understanding these data formats is essential before working with MongoDB.

1. Structured Data

Structured data refers to data that is organized in a well-defined schema or format, typically in rows and columns like a relational database table. Each field has a fixed data type and the schema must be defined before data can be inserted.

Characteristics of Structured Data:

- Organized in tables with rows and columns (relations)
- Follows a predefined schema (fixed structure)
- Data types for each column are strictly defined
- Supports SQL (Structured Query Language) for querying
- Easy to search, sort, filter, and analyse
- Examples: Relational databases like MySQL, PostgreSQL, Oracle

Example: Structured Data Table

Consider a student record stored in a relational database table:

Student_ID	Name	Age	Marks	Department
101	Anushka	20	85	Computer Science

102	Rahul	21	78	Information Technology
103	Priya	19	91	Computer Science
104	Amit	22	67	Electronics

In the above table, the schema is fixed: each row has exactly Student_ID, Name, Age, Marks, and Department. No additional fields can be added without altering the table structure.

Advantages of Structured Data:

- Easy to store and retrieve using SQL
- Efficient for complex queries and joins
- Strong data integrity and consistency
- Well-suited for transactional systems (banking, payroll, etc.)

Disadvantages of Structured Data:

- Rigid schema — difficult to modify once data is stored
- Not suitable for unstructured or complex data like images, documents
- Scaling horizontally is challenging

2. Semi-Structured Data

Semi-structured data is data that does not conform to a rigid, tabular structure but still contains some organizational properties such as tags, markers, or hierarchies that make it easier to analyze. It lies between structured (relational) and unstructured data.

MongoDB stores data as semi-structured documents (BSON — Binary JSON internally), which allows flexibility in the schema. Different documents within the same collection can have different fields.

Characteristics of Semi-Structured Data:

- No fixed schema — each record can have different fields
- Uses self-describing tags or markers (e.g., XML tags, JSON keys)
- Supports hierarchical and nested data structures
- More flexible than structured data
- Examples: JSON, XML, YAML, Email headers, HTML

Example: Semi-Structured Data (XML Format)

The same student records represented in XML:

```
<students>
  <student>
    <student_id>101</student_id>
    <name>Anushka</name>
    <age>20</age>
    <department>Computer Science</department>
    <marks>85</marks>
  </student>
  <student>
    <student_id>102</student_id>
    <name>Rahul</name>
    <age>21</age>
    <department>Information Technology</department>
    <marks>78</marks>
    <hostel>Yes</hostel>
  </student>
</students>
```

Notice that the second student record has an additional field 'hostel' which is absent in the first record. This flexibility is a key feature of semi-structured data.

Advantages of Semi-Structured Data:

- Flexible schema — fields can vary from record to record
- Supports complex and nested data structures
- Easy to exchange data between different systems
- Ideal for web data, APIs, IoT sensor data

Disadvantages of Semi-Structured Data:

- Harder to query compared to structured data
- Inconsistent structure can lead to data quality issues
- May require additional parsing or processing

3. JSON (JavaScript Object Notation) Format

JSON (JavaScript Object Notation) is a lightweight, text-based, and language-independent data interchange format. It is the most widely used format for representing semi-structured data in modern web applications, REST APIs, and NoSQL databases like MongoDB.

JSON is easy for humans to read and write, and easy for machines to parse and generate. It is derived from the JavaScript programming language but is supported by almost all modern programming languages.

JSON Syntax Rules:

- Data is represented as key-value pairs
- Keys must be strings enclosed in double quotes (")
- Values can be: String, Number, Boolean, Array, Object, or null
- Objects are enclosed in curly braces { }
- Arrays are enclosed in square brackets [ ]
- Key-value pairs are separated by commas (,)

JSON Data Types:

Data Type	Example Value	Description
String	"Anushka"	Text enclosed in double quotes
Number	20, 85.5	Integer or floating point number
Boolean	true, false	Logical true or false value
Array	[85, 90, 78]	Ordered list of values
Object	{ "city": "Pune" }	Nested key-value pairs
Null	null	Represents an empty/missing value

Example: JSON Representation of Student Data

```
{
  "student_id": 101,
  "name": "Anushka",
  "age": 20,
  "department": "Computer Science",
  "marks": 85,
  "is_active": true,
  "address": {
    "street": "123, MG Road",
    "city": "Pune",
    "pincode": "411001"
  },
  "subjects": ["Python", "MongoDB", "OS", "DBMS"]
}
```

In the above JSON document:

- "student_id" is a Number
- "name" and "department" are Strings

- "is_active" is a Boolean
- "address" is a nested Object
- "subjects" is an Array of Strings

Array of JSON Objects (Collection of Records):

Multiple student records can be represented as a JSON array:

```
[
  {
    "student_id": 101,
    "name": "Anushka",
    "age": 20,
    "department": "Computer Science"
  },
  {
    "student_id": 102,
    "name": "Rahul",
    "age": 21,
    "department": "Information Technology",
    "hostel": true
  },
  {
    "student_id": 103,
    "name": "Priya",
    "age": 19,
    "department": "Computer Science",
    "marks": [85, 90, 78]
  }
]
```

Notice that each object (document) in the array can have different fields — this is the flexibility of JSON/semi-structured data.

4. Comparison: Structured vs Semi-Structured vs JSON

Feature	Structured	Semi-Structured	JSON
Schema	Fixed/Rigid	Flexible	Flexible
Format	Tables (Rows/Cols)	XML, HTML, Email	Key-Value Pairs
Query Language	SQL	XQuery, XPath	MongoDB Query
Nesting	Not Supported	Supported	Supported
Data Types	Strict	Mixed	Mixed

Scalability	Vertical	Horizontal	Horizontal
Example	MySQL, Oracle	XML files, Emails	MongoDB, REST APIs
Human Readable	Partially	Partially	Yes

5. JSON and BSON in MongoDB

MongoDB stores data in a format called BSON (Binary JSON). BSON is a binary-encoded serialization of JSON-like documents. It extends JSON by adding additional data types not available in standard JSON.

Why BSON instead of JSON?

- BSON is more efficient to encode and decode compared to plain JSON text
- BSON supports additional data types: Date, ObjectId, Binary, Decimal128
- BSON allows MongoDB to traverse document structure quickly
- BSON supports embedded documents and arrays natively

BSON Additional Data Types:

BSON Type	Example	Description
ObjectId	ObjectId("64ab...")	Unique identifier for MongoDB documents
Date	ISODate("2025-01-01")	Date and time values
Binary	BinData(0, "...")	Binary data like images, files
Decimal128	NumberDecimal("9.99")	High precision decimal numbers
Int32 / Int64	NumberInt(100)	32-bit and 64-bit integers

Example: MongoDB Document (BSON/JSON)

```
{
  "_id": ObjectId("64ab1234ef567890abcd1234"),
  "student_id": 101,
  "name": "Anushka",
  "dob": ISODate("2005-06-15"),
  "department": "Computer Science",
  "marks": {
    "python": 88,
    "mongodb": 92,
    "os": 79
```

```
  },
  "is_active": true,
  "enrolled_date": ISODate("2023-07-01")
}
```

The '_id' field is automatically generated by MongoDB and is of type ObjectId, which uniquely identifies each document in a collection.

6. Design Principles for Data Representation in MongoDB

When designing data models in MongoDB, the following principles should be considered:

A. Embedding vs Referencing

MongoDB allows two approaches to model relationships between data:

i) Embedding (Denormalization):

Related data is stored within the same document as nested objects or arrays. This is preferred when data is frequently accessed together.

```
// Embedded document example
{
  "order_id": 501,
  "customer": "Rahul",
  "items": [
    { "product": "Laptop", "qty": 1, "price": 55000 },
    { "product": "Mouse", "qty": 2, "price": 500 }
  ],
  "total": 56000
}
```

ii) Referencing (Normalization):

Related data is stored in separate collections and linked using references (similar to foreign keys in SQL). This is preferred when data is large or shared across many documents.

```
// Referencing example
// Student document:
{ "_id": ObjectId("abc"), "name": "Priya", "dept_id": ObjectId("xyz") }

// Department document:
{ "_id": ObjectId("xyz"), "dept_name": "Computer Science", "hod": "Dr. Mulay" }
```

B. Schema Design Guidelines:

- Embed data that is always accessed together

- Use references when data entities have many-to-many relationships
- Avoid deeply nested documents (more than 3-4 levels)
- Design documents based on your application's query patterns
- Use arrays to store list-type data (e.g., subjects, phone numbers)
- Keep document size reasonable (MongoDB has a 16MB document size limit)

SET A

1. Design a JSON document to represent a student record containing name, roll number, age, department, list of subjects, and address (city and pincode). Write the JSON structure manually.
2. Compare structured data (table format) and semi-structured data (JSON format) using an example of an employee record containing empid, name, salary, department, and skills.
3. Write a JSON document to represent a library book record containing book_id, title, author, price, availability (boolean), and an array of genres.
4. Represent the following structured table data as a JSON array of objects for a product catalog containing product_id, product_name, category, price, and stock_quantity.
5. Create a JSON document for a hospital patient record using embedded documents. The record should include patient_id, name, age, diagnosis, doctor details (name and specialization), and a list of prescribed medicines.
6. Design a JSON document for a movie record containing movie_id, title, release_year, director (name and country), runtime, rating (out of 5), and an array of cast members with their character names.
7. Convert the following structured table data (Customer: cust_id, fname, lname, email, phone) into a JSON array of objects and add an optional "loyalty_points" field to only some records to demonstrate schema flexibility.
8. Create a JSON document representing a weather record with location (city, country, latitude, longitude), current conditions (temperature, humidity, wind_speed), and a 5-day forecast array with daily temperatures.
9. Design a JSON structure for a course enrollment record that includes course_id, course_name, instructor_name, credits, semester, max_capacity, and an array of enrolled students with their registration_ids.
10. Create a JSON document for an online quiz with quiz_id, title, subject, questions array (each question containing question_text, options, and correct_answer), and total_marks.

SET B

1. Design a JSON schema for an e-commerce order management system. The document should include order_id, customer (with embedded address), list of items (product name, quantity, price), order_status, and payment details (method and transaction_id).
2. Design two separate JSON documents demonstrating the referencing (normalization) approach for a university system where students belong to departments. Show how department_id is referenced in the student document.

3. Convert the following XML semi-structured data into equivalent JSON format for a company employee record containing employee details, projects assigned, and contact information.
4. Create a JSON array with at least 5 documents representing an IPL player dataset. Include fields: player_id, name, team, category (Batsman/Bowler/All-rounder), bid_price, runs_scored, and wickets_taken. Ensure that different documents have different optional fields to demonstrate schema flexibility.
5. Explain with example the difference between BSON and JSON. List at least four additional data types that BSON supports which are not available in standard JSON, with examples of each.
6. Design a MongoDB data model for a Banking System using referencing approach. Create separate JSON documents for: Account (account_id, account_type, balance, customer_id), Customer (customer_id, name, email), and Branch (branch_id, branch_name, location). Show how documents reference each other.
7. Create a JSON array representing a real estate property listing dataset with at least 6 documents. Include fields: property_id, title, location (city, area, latitude, longitude), price, property_type, bedrooms, bathrooms, and optional fields like "pool", "garage", "furnished" to demonstrate schema flexibility in MongoDB.
8. Compare BSON and JSON by providing a practical MongoDB scenario. Show: (1) How ObjectId is used for document identification, (2) How ISODate handles timestamps better than JSON, (3) How Decimal128 ensures precision for financial data, and (4) How Binary data type handles file storage. Provide examples for each.
9. Convert the following XML structure into JSON format for a company's project management system:

```
<project>
<project_id>P101</project_id>
<project_name>AI Implementation</project_name>
<team_members>
<member><name>Arun</name><role>Lead</role></member>
<member><name>Pooja</name><role>Developer</role></member>
</team_members>
<milestones>
<milestone><title>Planning</title><status>Complete</status></milestone>
</milestones>
</project>
```

SET C

1. Design a complete MongoDB data model for a Social Media application. The model should have at least three collections: Users, Posts, and Comments. Demonstrate embedding for comments within posts and referencing for user profiles. Design JSON documents for each collection with appropriate fields.
2. Case Study: Online Food Delivery System — Design a complete JSON-based data representation for a food delivery platform. Create JSON documents for: Restaurant (with

embedded menu items), Customer (with address), Order (referencing restaurant and customer IDs, with embedded ordered items). Justify your choice of embedding vs referencing for each relationship.

3. Research and document the limitations of JSON format and how BSON overcomes those limitations in MongoDB. Provide concrete examples showing where standard JSON would fail and how BSON handles the same scenario correctly.

Assignment Evaluation

0:Not Done	☐	2:Late Complete	☐	4:Complete	☐
1:Incomplete	☐	3:Needs Improvement	☐	5: Well Done	☐

Signature of the instructor: _____ **Date :**_____

Assignment No.2

Installation and Configuration of MongoDB with Database and Collection Creation (Total Slots = 2)

Aim

To install MongoDB, configure it, create a database, and create collections for storing data.

Software Requirement

- Ubuntu 16.04/18.04/20.04
- MongoDB
- Terminal

Introduction to MongoDB

MongoDB is a document database. It stores data in a type of JSON format called BSON. It provides flexibility, scalability, and high performance for modern applications. A record in MongoDB is a document, which is a data structure composed of key value pairs similar to the structure of JSON objects.

MongoDB Applications

MongoDB is a NoSQL database. MongoDB provides following functionality to the database programmers :

- Stores user data, comments and metadata
- MongoDB performs complex analytics queries and stores the behavioral data.
- This is used to manage chain data and optimize logistics.
- Environmental data and IoT devices are stored and analyzed.

Important Terms

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of documents and is similar to an RDBMS table. A collection exists within a single database. Documents within a collection can have different fields.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Why Use MongoDB?

- **Document Oriented Storage** − Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries

- Fast in-place updates
- Professional support by MongoDB

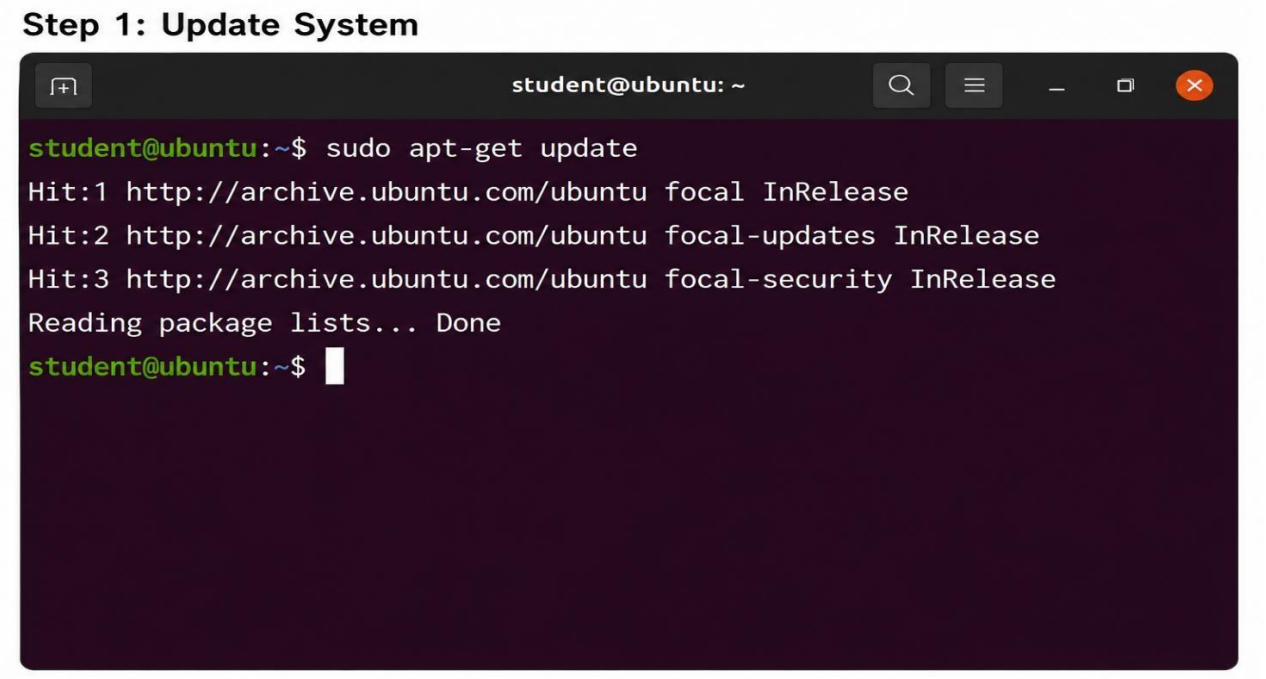
Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Commands / Procedure

Step 1: Update System

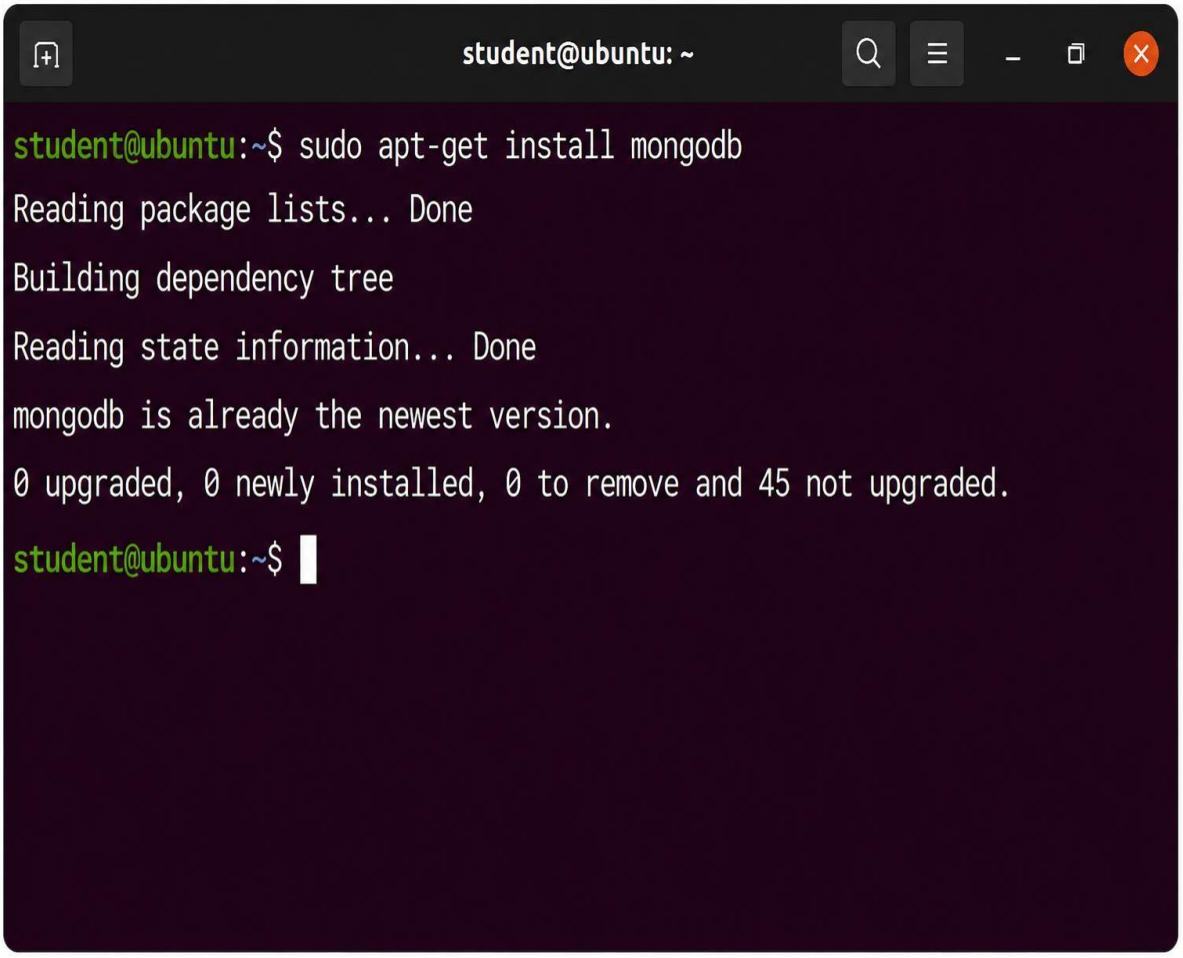
sudo apt-get update



Step 2: Install MongoDB

sudo apt-get install mongodb

Step 2: Install MongoDB

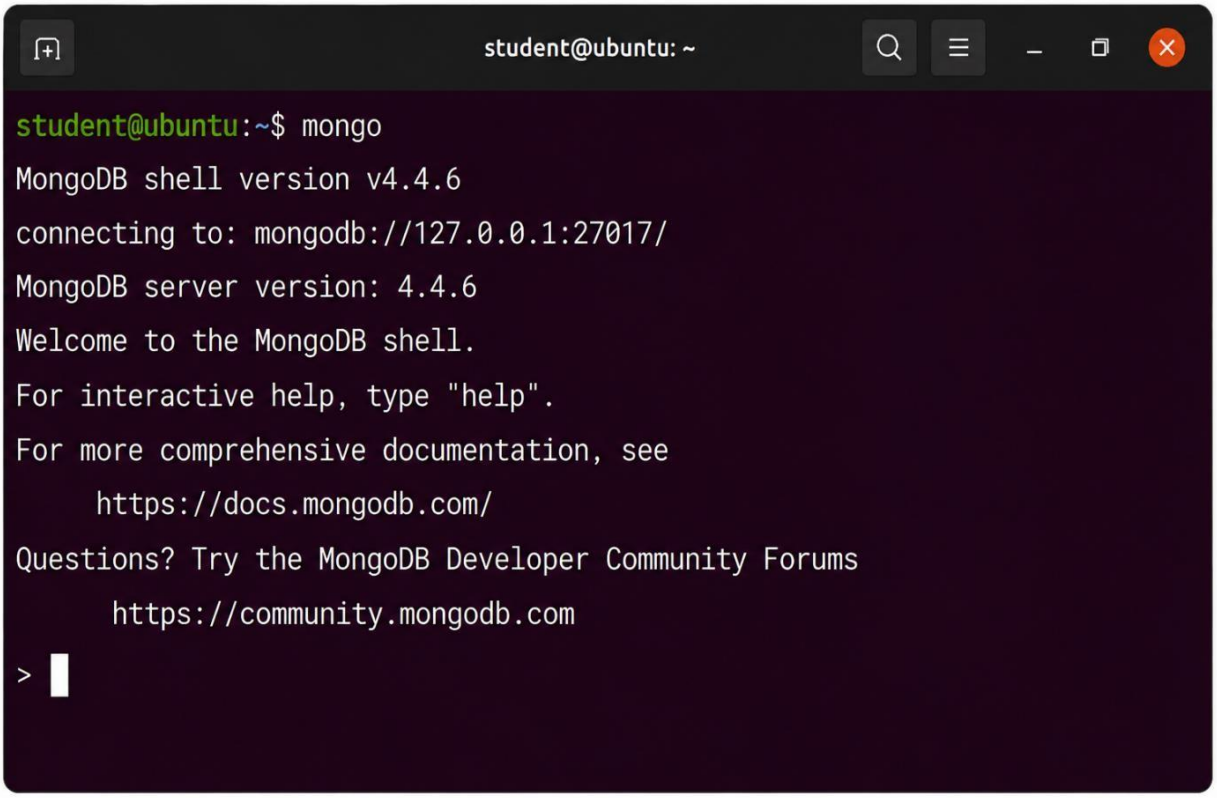


Step 3: Start MongoDB Service

sudo service mongodb start

Step 4: Open Mongo Shell

Step 4: Open Mongo Shell



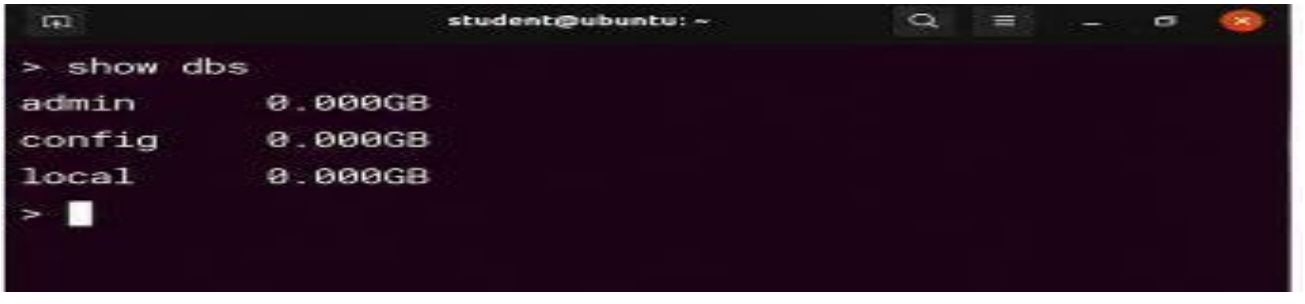
mongo

After connecting to your database using mongo, you can see which database you are using by typing db in your terminal.

> db

To see all available databases, in your terminal type show dbs.

>show databases



Step 5: Create Database

You can change or create a new database by typing use then the name of the database.

Syntax: use <your_db_name>
eg.
> use College
> switched to db College

Step 6: Create a Collection
You can create a collection using the createCollection() database method.
Syntax: db.createCollection("Collection name")
Eg.
> db.createCollection("Students")
{ "ok" : 1 }

Step 7: Insert a Documents
There are 2 methods to insert documents into a MongoDB database.
insertOne()
To insert a single document, use the insertOne() method.
This method inserts a single object into the database.
db.Students.**insertOne**({
RollNo: TYBSC101,
Name: "Sarika",
Course: "BSC(Computer Science)"
})
>WriteResult({ "nInserted" : 1 })
insertMany()
To insert multiple documents at once, use the insertMany() method.
This method inserts an array of objects into the database.

```
db.Students.insertMany([
  {
RollNo: TYBSC101,
    Name: "Sarika",
    Course: "BSC(Computer Science)"
  },
  {
    RollNo: TYBSC102,
    Name: "Rajendra",
    Course: "BSC(Computer Science)"
  },
  {
RollNo: TYBSC103,
    Name: "Manisha",
    Course: "BSC(Computer Science)"
  }
])
```

Step 8: View Documents

There are 2 methods to find and select data from a MongoDB collection, **find() and findOne().**

find()

To select data from a collection in MongoDB, we can use the find() method.
This method accepts a query object. If left empty, all documents will be returned.
>db.Students.find()

findOne()

To select only one document, we can use the findOne() method.
This method accepts a query object. If left empty, it will return the first document it finds.
>db.Students.findOne()

If you prefer your result nicely formatted, use pretty():

>db.Students.find().pretty()
Eg.
>db.Customer.find(
{},
{Name:1, ContactNumber:1, id:0}
)

This query is used to display only the **Name** and **Contact Number** fields from all documents in the **Customer** collection.

- db.Customer.find() : Retrieves documents from the Customer collection.
- {} : An empty filter condition that selects all documents in the collection.
- {Name:1, ContactNumber:1, id:0} : Projection clause used to display specific fields.
    - Name:1 → Includes the Name field in the output.
    - ContactNumber:1 → Includes the Contact Number field in the output.

- id:0 → Excludes the default MongoDB id field from the output.

Count Total
> db.Students.countDocuments()

deleteOne() Method
MongoDB's **deleteOne()** method is used to remove a document from the collection. deleteOne() method accepts deletion criteria.
Syntax: db.COLLECTION_NAME.deleteOne(DELLETION_CRITTERIA)
Eg. Delete the document whose Name is 'Manisha'.
>db.Students.deleteOne({'Name': 'Manisha'})

remove() Method
MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria** − (Optional) deletion criteria according to documents will be removed.
- **justOne** − (Optional) if set to true or 1, then remove only one document.

Syntax: db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
Eg. Remove all the documents whose Roll No. is 'TYBSC102'.
>db.Students.remove({'RollNo':TYBSC102 '})

Remove Only One
If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.
Syntax: db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)

Remove All Documents
If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection.
Syntax: db.Students.remove({})

dropcollection() Method
MongoDB's db.collection.drop() is used to drop a collection from the database.
Syntax: db.COLLECTION_NAME.drop()
Eg.> db.Students.drop()

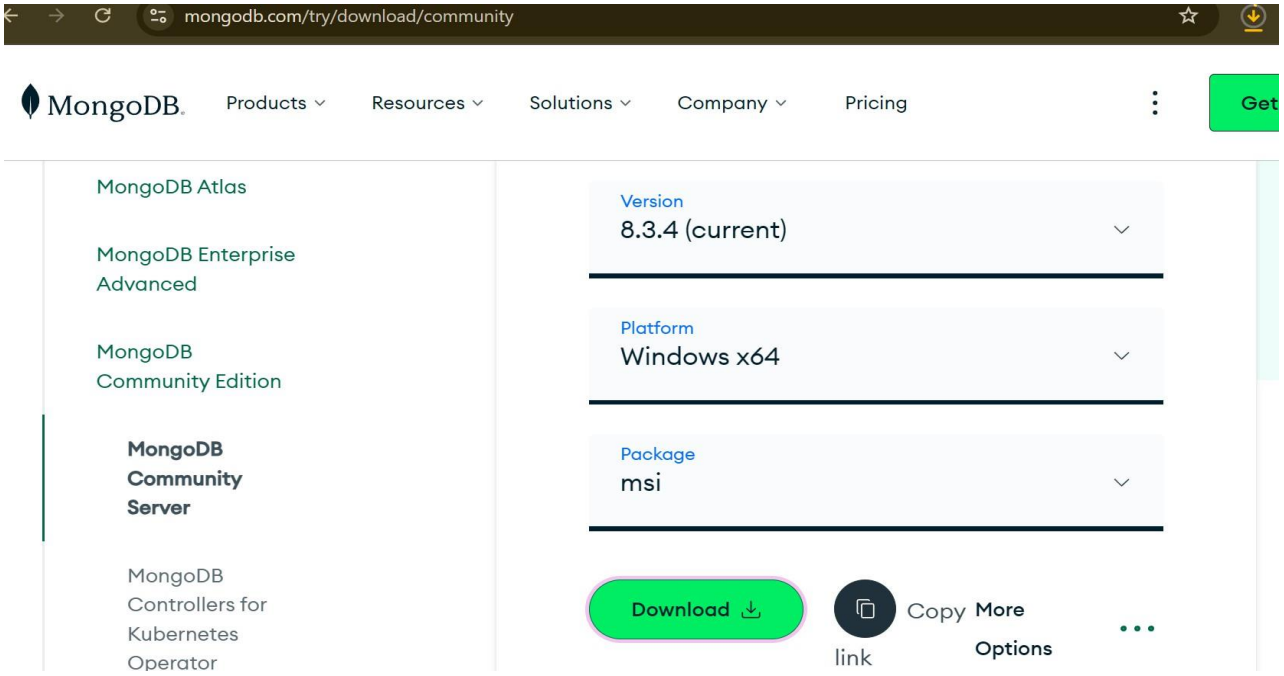
dropDatabase() Method
MongoDB **db.dropDatabase()** command is used to drop a existing database.
Syntax:db.dropDatabase()
Eg.>db.dropDatabase("College ")

If you have not selected any database, then it will delete default 'test' database.

Installation On Windows 10

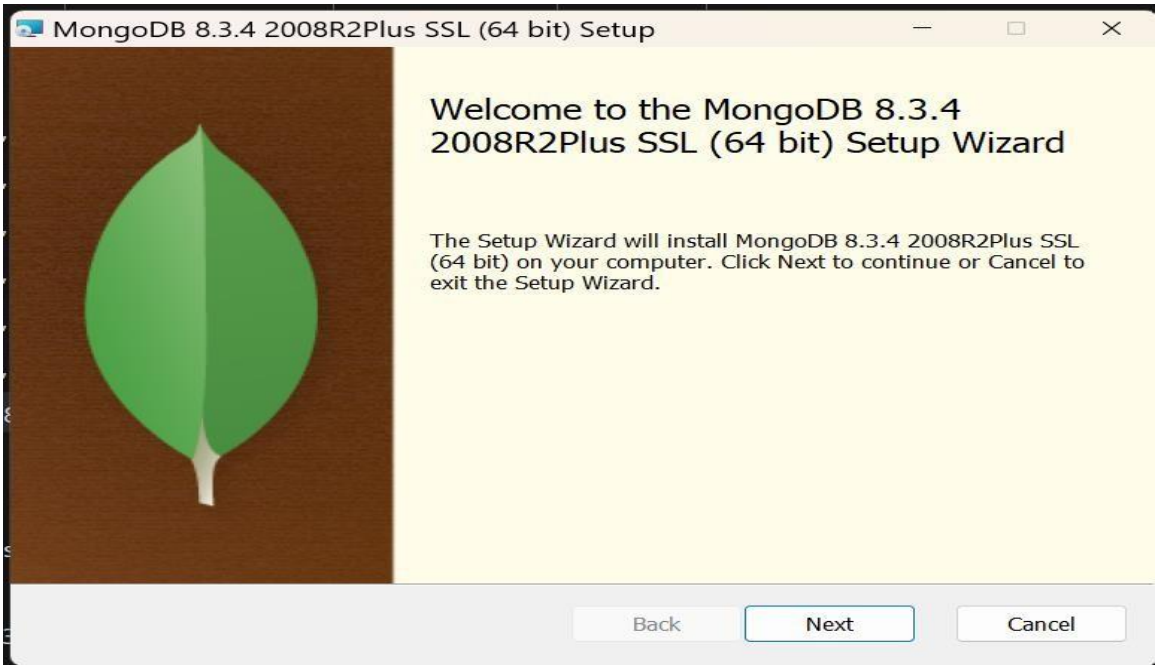
Step 1: Download MongoDB Community Server

1. Go to MongoDB Community Server Download.
2. Select:
    - Version : Latest Stable Process
    - Platform: Windows
    - Package:MSI
2. 3. Click Download.

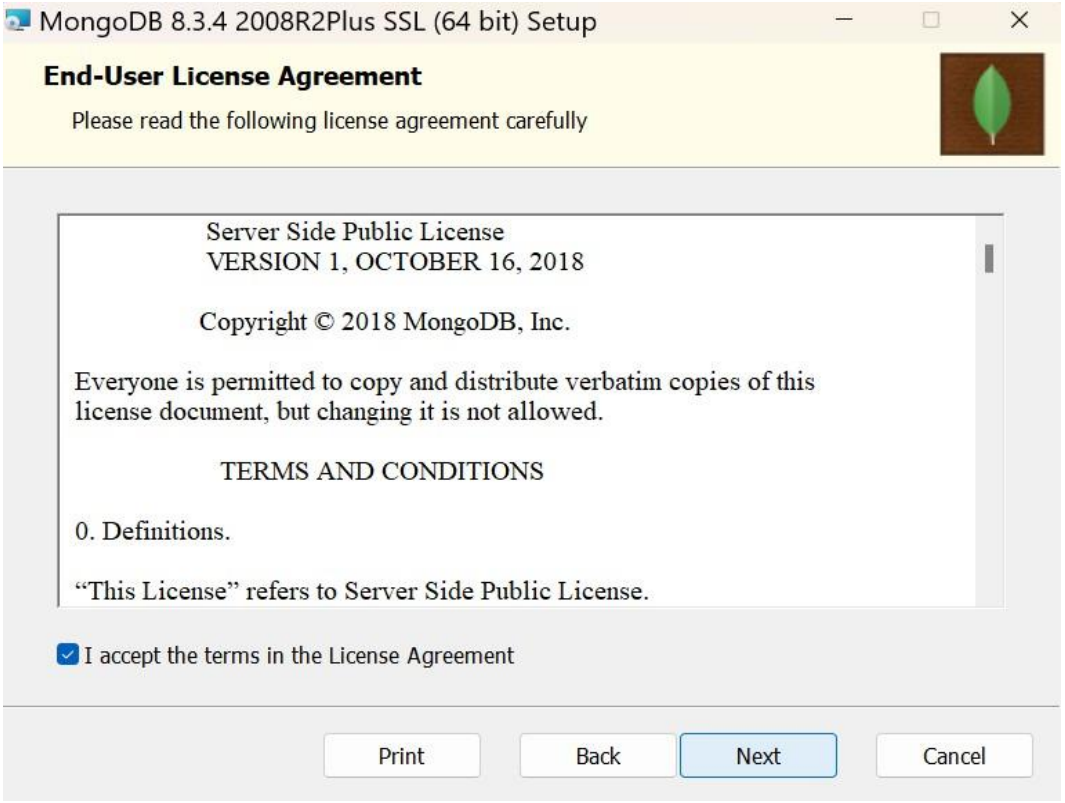


Step 2: Install MongoDB

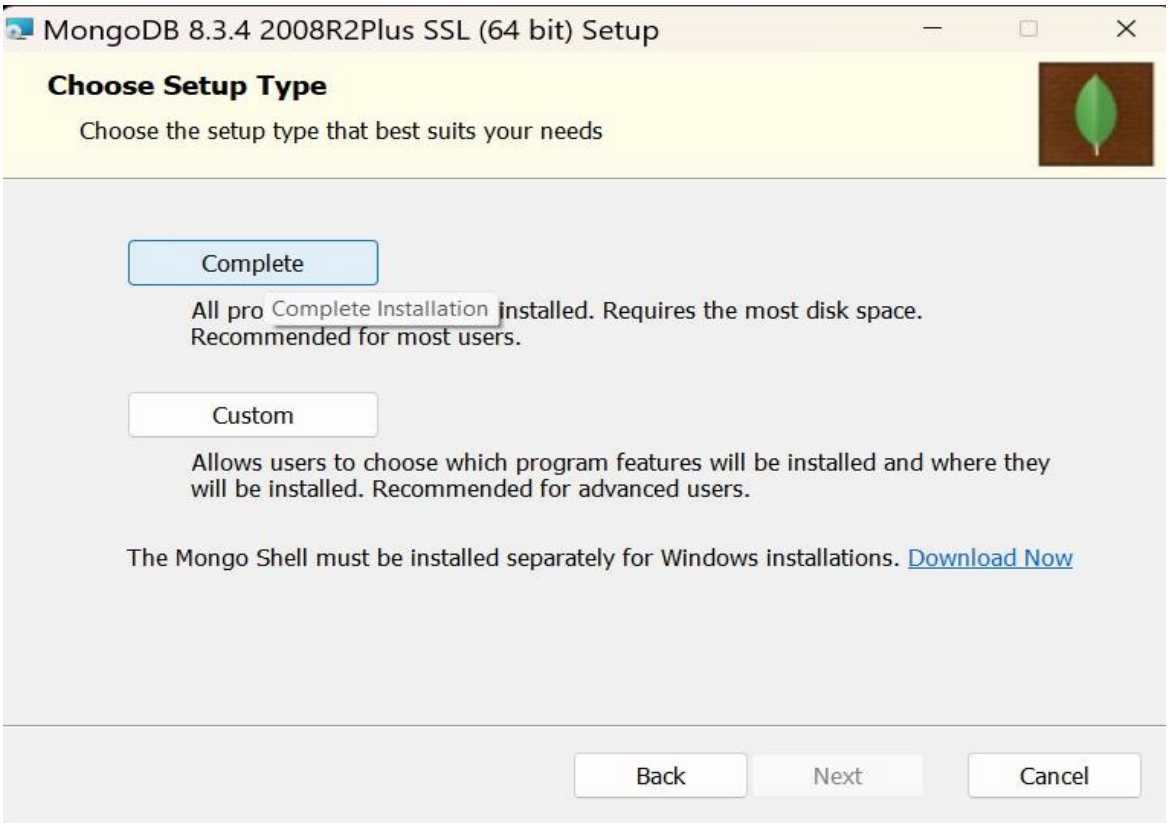
1. Open the downloaded .msi file.
2. Click next.



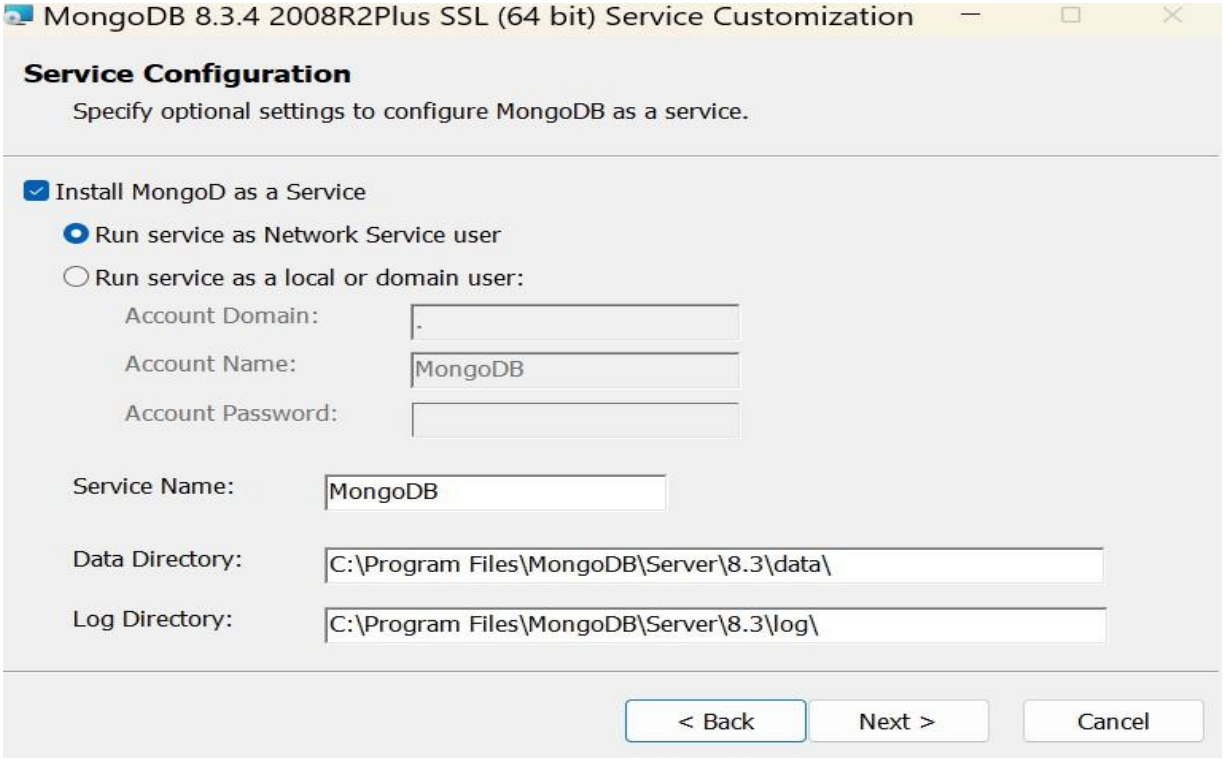
3. Accept the License Agreement. Click Next.



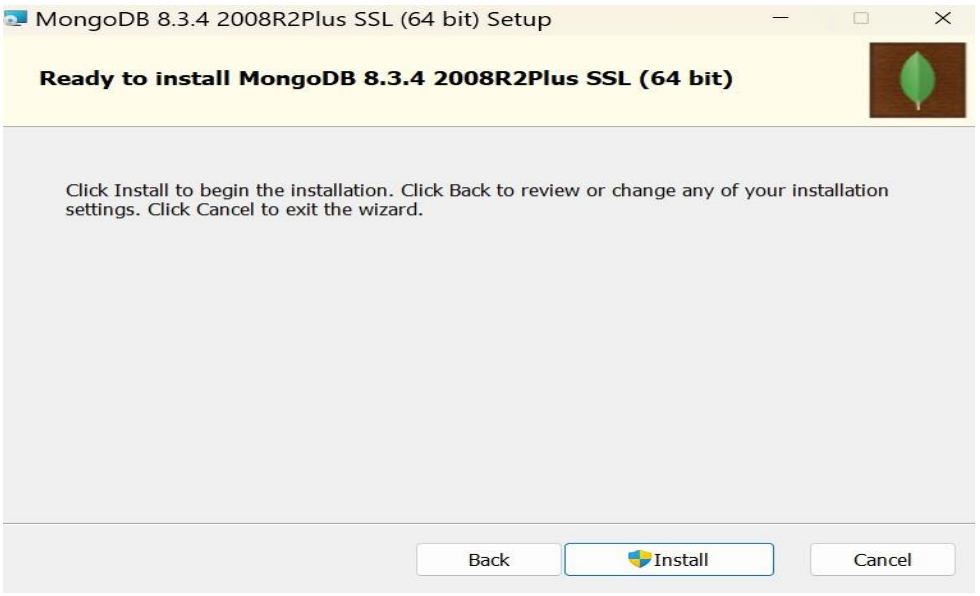
4. Select Complete installation.

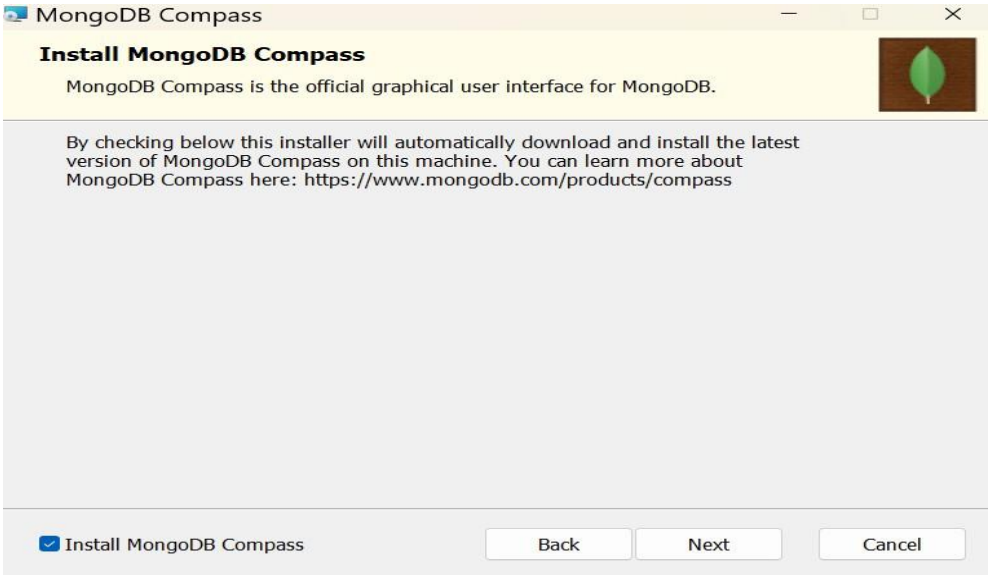


5. Check Install MongoDB as a Service.



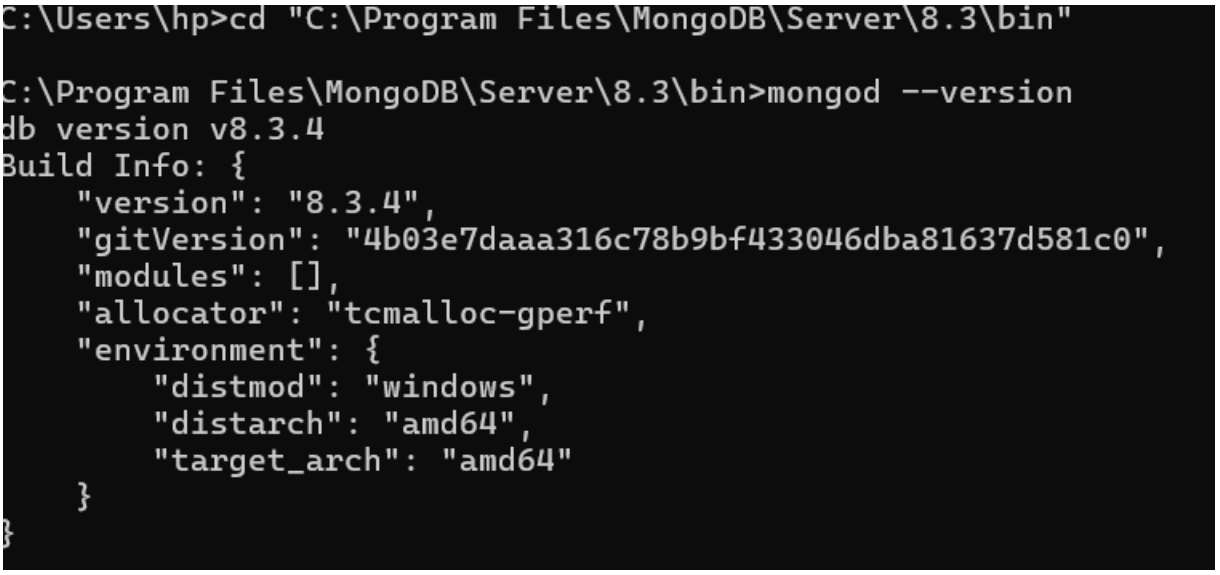
1. Click Next -> Install -> Finish.


MongoDB 8.3.4 2008R2Plus SSL (64 bit) Setup
Ready to install MongoDB 8.3.4 2008R2Plus SSL (64 bit)
Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.
Back
Install
Cancel


MongoDB Compass
Install MongoDB Compass
MongoDB Compass is the official graphical user interface for MongoDB.
By checking below this installer will automatically download and install the latest version of MongoDB Compass on this machine. You can learn more about MongoDB Compass here: https://www.mongodb.com/products/compass
Install MongoDB Compass
Back
Next
Cancel


MongoDB 8.3.4 2008R2Plus SSL (64 bit) Setup
Completed the MongoDB 8.3.4 2008R2Plus SSL (64 bit) Setup Wizard
Click the Finish button to exit the Setup Wizard.
Back
Finish
Cancel

Step 3: Verification Installation

1. Open Command Prompt.
2. Run: mongod –version
3. 

Step 4: Start MongoDB service open command prompt as Administrator and Run

net start MongoDB

Step 5: Open MongoDB Shell Run

mongosh

If Successful You will see a prompt like:

test>

Step 6: Create Database

MongoDB supports many datatypes. Some of them are –

- String − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- Integer − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- Boolean − This type is used to store a boolean (true/ false) value.
- Double − This type is used to store floating point values.
- Min/ Max keys − This type is used to compare a value against the lowest and highest BSON elements.
- Arrays − This type is used to store arrays or list or multiple values into one key.
- Timestamp − ctimestamp. This can be handy for recording when a document has been modified or added.
- Object − This datatype is used for embedded documents.
- Null − This type is used to store a Null value.
- Symbol − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

- Date − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- Object ID − This datatype is used to store the documents ID.
- Binary data − This datatype is used to store binary data.
- Code − This datatype is used to store JavaScript code into the document.
- Regular expression − This datatype is used to store regular expression.

SET A

1. Create a database with the name “Movie”.
2. A “Film” is a collection of documents with the following fields:
    a. Film Id
    b. Title of the film
    c. Year of release
    d. Genre / Category (like adventure, action, sci-fi, romantic etc.) A film can belong to more than one genre.
    e. Actors (First name and Last name) A film can have more than one actor.
    f. Director (First name and Last name) A film can have more than one director.
    g. Release details (It consists of places of release, dates of release and rating of the film.)
3. An “Actor” is a collection of documents with the following fields:
    a. Actor Id
    b. First name
    c. Last Name
    d. Address (Street, City, State, Country, Pin-code)
    e. Contact Details (Email Id and Phone No)
    f. Age of an actor.

    Queries:

    1. Insert at least 10 documents in the collection Film and collection Actor.
    2. Display all the documents inserted in both the collections.
    3. Find Film by Title ‘---------‘.
    4. Find Actor by First Name
    5. Display Only Film Titles.
    6. Delete a Film ‘ ------ ‘.
    7. Count Total Films.
    8. Remove all the documents whose Film id is ‘-------‘.
    9. Remove first record in the collection Actor.
    10. Drop database ‘Movie’.

SET B

1. Create a database with name “Company”.
2. An “Employee” is a collection of documents with the following fields:
    a. Employee ID
    b. First Name
    c. Last Name
    d. Email

e. Phone No.
f. Address (House No, Street, City, State, Country, Pin-code)
g. Salary
h. Designation
i. Experience
j. Date of Joining
k. Birthdate

3. A "Transaction" is a collection of documents with the following fields:
   a. Transaction Id,
   b. Transaction Date
   c. Name (First Name of employee who processed the transaction)
   d. Transaction Details (Item Id, Item Name, Quantity, Price)
   e. Payment (Type of Payment (Debit/Credit/Cash), Total amount paid, Payment Successful.

Queries:

1. Insert at least 10 documents in the collection Employee.
2. Insert at least 10 documents in the collection Transaction.
3. Display all the documents inserted in both the collections.
4. Find Employee by Employee ID ' ----'.
5. Find Employee by Designation ' ------ '.
6. Delete only one Transaction id ' -----'.
7. Count all Transactions and Count all Employee.
8. Find Credit Card Payments.
9. Delete Employee whose First name '-----' and Last name is '-----'.
10. Remove first record in the collection Employee.

SET C

1. Create a database with name "Hotel".
2. An "Customer" is a collection of documents with the following fields:
   a. Customer ID
   b. Name
   c. Contact Number
   d. Email
   e. Address
3. A "Room" is a collection of documents with the following fields:
   a. Room Number
   b. Room Type
   c. Rent Per Day
   d. Availability Status
4. A "Booking" is a collection of documents with the following fields:
   a. Booking ID
   b. Customer Name
   c. Room Number
   d. Check-In Date
   e. Check-Out Date
   f. Total Bill

Queries:

1. Insert at least 10 documents in the collection Customer,Room and Booking.
2. Display all the documents inserted in above collections.
3. Delete whose Customer id is ‘102’, ‘105’ and ‘107’.
4. Find Booking by Booking ID ‘ --- ‘.
5. Count Customers and Room collection.
6. Find Customer from ‘Satara’.
7. Remove a Specific Booking id ‘-----‘.
8. Display Customer Name is ‘Rani’ and Contact Number is ‘9845671237’.
9. Display Room Number ‘-----' and RentPerDay ‘----- ’ database Hotel.
10. Remove a Specific Booking whose customer name is ‘ ---- ‘.
11. Find Room Availability Status ‘-----'.
12. Find Room Type=‘Deluxe' Rooms.
13. Remove All Documents from Customer Collection.
14. Delete all collection in Booking.
15. Drop database ‘Hotel’.

Assignment Evaluation

0:Not Done	☐	2:Late Complete	☐	4:Complete	☐
1:Incomplete	☐	3:Needs Improvement	☐	5: Well Done	☐

Signature of the instructor: _____ **Date :**_____

Assignment 3

Execution of CRUD Operations and Data Manipulation in MongoDB (Total Slots = 2)

Introduction to CRUD Operations in MongoDB

CRUD stands for Create, Read, Update, and Delete – the four fundamental operations performed on data stored in a database. In MongoDB, these operations are executed using built-in methods that work on collections of documents (JSON-like structures stored in BSON format).

CRUD Operation	MongoDB Method(s)	SQL Equivalent
Create	insertOne(), insertMany()	INSERT INTO
Read	find(), findOne()	SELECT
Update	updateOne(), updateMany(), replaceOne()	UPDATE
Delete	deleteOne(), deleteMany()	DELETE FROM

Prerequisites / Operating Environment

- MongoDB Community Server (v6.0 or higher)
- MongoDB Compass (GUI) or MongoDB Shell (mongosh)
- Windows / Linux Operating System
- Command Line Interface / VS Code with MongoDB Extension

CREATE Operations

CREATE operations insert new documents into a MongoDB collection. If the collection does not exist, MongoDB creates it automatically. MongoDB provides two primary methods for inserting documents.

3.1 insertOne()

Inserts a single document into a collection.

Syntax:

```
db.collection_name.insertOne({ field1: value1, field2: value2, ... })
```

Example – Insert one student document:

```
use studentDB
db.students.insertOne({
   name: "Aarav Sharma",
   age: 20,
   department: "Computer Science",
   marks: 88,
   city: "Pune"
})
```

Output:

```
{
acknowledged: true,
insertedId: ObjectId("64f3a1b2c3d4e5f6a7b8c9d0")

}
```

3.2 insertMany()

Inserts multiple documents into a collection in a single operation. Accepts an array of documents.

Syntax:

```
db.collection_name.insertMany([ {doc1}, {doc2}, ... ])
```

Example – Insert multiple student documents:

```
db.students.insertMany([
  { name: "Priya Patil",      age: 21, department: "Electronics", marks: 75, city: "Nashik" },
  { name: "Rohan Desai",      age: 22, department: "Mechanical",  marks: 82, city: "Mumbai" },
  { name: "Sneha Joshi",      age: 20, department: "Computer Science", marks: 91, city: "Pune" },
  { name: "Amit Kulkarni", age: 23, department: "Civil", marks: 68, city: "Nagpur" }
])
```

READ Operations

READ operations retrieve documents from a collection. MongoDB's find() method supports query filters, projections, sorting, and limiting results.

4.1 find()

Returns all documents that match the given query filter. If no filter is provided, it returns all documents in the collection.

Syntax:

```
db.collection_name.find(query, projection)
```

4.1.1 Find All Documents

```
db.students.find()
// or with pretty print:
db.students.find().pretty()
```

4.1.2 Find with Filter (Condition)

Example – Find all students from "Pune":

```
db.students.find({ city: "Pune" })
```

Example – Find students with marks greater than 80:

```
db.students.find({ marks: { $gt: 80 } })
```

4.1.3 Projection – Select Specific Fields

Projection controls which fields are returned. Use 1 to include a field and 0 to exclude it.

Example – Show only name and department (exclude _id):

```
db.students.find({}, { name: 1, department: 1, _id: 0 })
```

4.2 findOne()

Returns only the first document that matches the query filter.

Example:

```
db.students.findOne({ name: "Priya Patil" })
```

4.3 Query Operators for READ

Operator	Description	Example
$eq	Equal to	{ marks: { $eq: 88 } }
$ne	Not equal to	{ city: { $ne: "Mumbai" } }
$gt	Greater than	{ marks: { $gt: 80 } }
$gte	Greater than or equal	{ marks: { $gte: 80 } }
$lt	Less than	{ marks: { $lt: 70 } }
$lte	Less than or equal	{ marks: { $lte: 70 } }
$in	Matches any value in array	{ city: { $in: ["Pune","Mumbai"] } }
$nin	Not in array	{ dept: { $nin: ["Civil"] } }
$and	Logical AND	{ $and: [{marks:{$gt:70}},{city:"Pune"}] }
$or	Logical OR	{ $or: [{marks:{$gt:90}},{city:"Nagpur"}] }

4.4 Sorting, Limiting and Skipping

Example – Sort students by marks in descending order:

```
db.students.find().sort({ marks: -1 })          // -1 = descending, 1 = ascending
```

Example – Limit results to top 3:

```
db.students.find().sort({ marks: -1 }).limit(3)
```

Example – Skip first 2 documents:

```
db.students.find().skip(2).limit(2)
```

4.5 countDocuments()

Returns the count of documents matching the filter.

```
db.students.countDocuments()                    // total count
db.students.countDocuments({ city: "Pune" })    // count by condition
```

UPDATE Operations

UPDATE operations modify existing documents in a collection. MongoDB provides updateOne(), updateMany(), and replaceOne() for this purpose.

5.1 Update Operators

Operator	Description
$set	Sets the value of a field (adds the field if not present)
$unset	Removes a field from a document
$inc	Increments or decrements a numeric field by a specified value
$push	Appends a value to an array field
$pull	Removes a value from an array field
$rename	Renames a field
$mul	Multiplies the value of a field by a number
$min	Updates only if the new value is less than the current value
$max	Updates only if the new value is greater than the current value

5.2 updateOne()

Updates the first document that matches the filter.

Syntax:

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

Example – Update city of student "Aarav Sharma":

```
db.students.updateOne(
   { name: "Aarav Sharma" },
   { $set: { city: "Mumbai" } }
)
```

Example – Increment marks by 5 for a student:

```
db.students.updateOne(
   { name: "Rohan Desai" },
   { $inc: { marks: 5 } }
)
```

5.3 updateMany()

Updates all documents that match the filter.

Example – Add a field "status" to all CS department students:

```
db.students.updateMany(
   { department: "Computer Science" },
   { $set: { status: "Active" } }
)
```

Example – Remove a field from all documents:

```
db.students.updateMany(
   {},
   { $unset: { status: "" } }
)
```

5.4 replaceOne()

Replaces the entire content of a matched document with a new document (the _id field is preserved).

Example:

```
db.students.replaceOne(
   { name: "Amit Kulkarni" },
   { name: "Amit Kulkarni", age: 24, department: "IT", marks: 79, city: "Aurangabad" }
)
```

5.5 upsert Option

When upsert is set to true, MongoDB inserts a new document if no document matches the filter.

Example:

```
db.students.updateOne(
   { name: "Kavita Rane" },
   { $set: { age: 21, department: "Chemistry", marks: 72, city: "Solapur" }
},
   { upsert: true }
```

)

DELETE Operations

DELETE operations remove documents from a collection. MongoDB provides deleteOne() and deleteMany() for this purpose.

6.1 deleteOne()

Deletes the first document that matches the filter.

Syntax:

```
db.collection_name.deleteOne(filter)
```

Example – Delete one student by name:

```
db.students.deleteOne({ name: "Rohan Desai" })
```

Output:

```
{ acknowledged: true, deletedCount: 1 }
```

6.2 deleteMany()

Deletes all documents that match the filter.

Example – Delete all students with marks less than 70:

```
db.students.deleteMany({ marks: { $lt: 70 } })
```

Example – Delete ALL documents from the collection:

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

Data Manipulation in MongoDB

Beyond basic CRUD, MongoDB supports powerful data manipulation including working with embedded documents, arrays, and bulk write operations.

7.1 Working with Embedded Documents

MongoDB documents can contain nested (embedded) objects. CRUD operations can target nested fields using dot notation.

Example – Insert a document with an embedded address:

```
db.students.insertOne({
  name: "Meera Joshi",
  age: 20,
  address: {
    street: "MG Road",
    city: "Pune",
    pincode: "411001"
  },
  marks: 85
})
```

Example – Query on embedded field using dot notation:

```
db.students.find({ "address.city": "Pune" })
```

Example – Update an embedded field:

```
db.students.updateOne(
  { name: "Meera Joshi" },
  { $set: { "address.pincode": "411002" } }
)
```

7.2 Working with Arrays

Example – Insert a document with an array field:

```
db.students.insertOne({
  name: "Karan Mehta",
  subjects: ["Mathematics", "Physics", "Chemistry"], marks:
  78
})
```

Example – Push a new subject into the array:

```
db.students.updateOne(
  { name: "Karan Mehta" },
```

```
  { $push: { subjects: "Computer Science" } }
)
```

Example – Pull (remove) a subject from the array:

```
db.students.updateOne(
  { name: "Karan Mehta" },
  { $pull: { subjects: "Physics" } }
)
```

Example – Find all students who have "Mathematics" in their subjects:

```
db.students.find({ subjects: "Mathematics" })
```

7.3 bulkWrite()

bulkWrite() executes multiple write operations (insert, update, delete) in a single call, which improves performance.

Example:

```
db.students.bulkWrite([
  { insertOne: { document: { name: "Pooja Nair", age: 21, marks: 80, city: "Kochi" } }
},
  { updateOne: { filter: { name: "Sneha Joshi" }, update: { $set: { marks:
95 } } } },
  { deleteOne: { filter: { name: "Amit Kulkarni" } } }
])
```

Complete Practice Session – Step by Step

Follow these steps in sequence in MongoDB Shell (mongosh) or MongoDB Compass to practise all CRUD operations.

1. **Step 1: Select / Create Database**

   ```
   use collegeDB
   ```

2. **Step 2: Insert Records (CREATE)**

   ```
   db.students.insertMany([
     { name: "Aarav Sharma",  age: 20, dept: "CS",                marks: 88, city: "Pune"
   ```

```
},
  { name: "Priya Patil",    age: 21, dept: "ECE",  marks: 75, city: "Nashik"
},
{   name: "Rohan Desai",    age: 22, dept: "Mech", marks: 82, city: "Mumbai"
},
{   name: "Sneha Joshi",    age: 20, dept: "CS",   marks: 91, city: "Pune"
},
{   name: "Amit Kulkarni",  age: 23, dept: "Civil",marks: 68, city: "Nagpur"
}
])
```

3. **Step 3: Retrieve Records (READ)**

```
db.students.find().pretty()                                    // all
records db.students.find({ dept: "CS" })                       // filter by
department
db.students.find({ marks: { $gte: 80 } }).sort({ marks: -1 })  // sorted
db.students.find({}, { name: 1, marks: 1, _id: 0 })            // projection
```

4. **Step 4: Modify Records (UPDATE)**

```
db.students.updateOne({ name: "Aarav Sharma" }, { $set: { city: "Mumbai" }
})
db.students.updateMany({ dept: "CS" }, { $inc: { marks: 2 } }) db.students.updateOne({
name: "Priya Patil" }, { $unset: { city: "" } })
```

5. **Step 5: Remove Records (DELETE)**

```
db.students.deleteOne({ name: "Amit Kulkarni" })
db.students.deleteMany({ marks: { $lt: 75 } })
```

6. **Step 6: Verify Final State**

```
db.students.find().pretty() db.students.countDocuments()
```

SET A

1. Create a database named "LibraryDB" and a collection "books". Insert at least 5 book documents with fields: title, author, year, genre, price, available (boolean).

2. Perform the following READ operations on the "books" collection:
   a) Display all books.
   b) Display only the title and author fields.
   c) Find all books with price greater than Rs. 300.
   d) Find books sorted by year in ascending order.
3. Update the price of a specific book using updateOne(). Also update the "available" field of all books in genre "Fiction" to true using updateMany().
4. Add a new field "discount" (10%) to all books with price > 400 using updateMany() and $set.
5. Delete one book by its title using deleteOne(). Then delete all books with "available: false" using deleteMany().
6. Use countDocuments() to count: (a) total books, (b) books with price < 200, (c) available books.
7. Find all books whose genre is either "Science Fiction" or "Technology" using the $in operator.
8. Display books published between 2015 and 2025 using $gte and $lte.
9. Increase the price of all books published before 2018 by 5% using the $mul operator.
10. Find the top 3 most expensive books using sort() and limit().

SET B

1. Create a "hospital" database with a "patients" collection. Insert 5+ patient documents with fields: patientId, name, age, disease, ward, admissionDate. Perform all CRUD operations.
2. Use $and and $or operators to retrieve: (a) patients in "General Ward" aged > 40, (b) patients with disease "Diabetes" OR "Fever".
3. Create a student collection with an embedded "address" document (street, city, pincode). Use dot notation to query and update embedded fields.
4. Add a "courses" array to student documents. Use $push to add a course, $pull to remove a course, and find() to search by course name.
5. Demonstrate the upsert option: write an updateOne() that inserts a new employee if not found, else updates the salary.
6. Use bulkWrite() to perform 3 operations together: insert a new product, update an existing product's stock, and delete a discontinued product.
7. Increase the age of all patients in "General Ward" by 1 year using $inc.
8. Remove the ward field from all patient documents using $unset.

9. Insert student documents containing an array field skills. Use:
    a. $push to add a skill
    b. $pull to remove a skill
    c. find() to search students by skill

SET C

1. Design an "ecommerce" database with "products" and "orders" collections. Implement complete CRUD for both collections with embedded order items.
2. Demonstrate use of $rename to rename a field across all documents, and $mul to multiply the price field by 1.18 (to add GST).
3. Create an "employees" collection. Use $min and $max update operators to conditionally update salary only when the new value is lower/higher than the existing one.

Assignment Evaluation

0:Not Done	☐	2:Late Complete	☐	4:Complete	☐
1:Incomplete	☐	3:Needs Improvement	☐	5: Well Done	☐

Signature of the instructor: _____ Date :_____

Assignment No. 4
Application of Aggregation Framework and Indexing Techniques in MongoDB (Total Slots = 2)

Aim

To study and implement Aggregation Framework and Indexing Techniques in MongoDB for data analysis and query optimization.

Objectives

1. To understand the Aggregation Framework in MongoDB.
2. To perform grouping, filtering, sorting, and statistical operations on documents.
3. To create and use different types of indexes.
4. To analyze query performance using indexing.

Aggregation Framework

Aggregation operations process multiple documents and return computed results. It is similar to the GROUP BY clause in SQL.

Common Aggregation Stages

Stage	Description
$match	Filters documents
$group	Groups documents
$sort	Sorts documents
$project	Selects fields
$limit	Limits output
$count	Counts documents

Indexing in MongoDB

Indexes are data structures that improve the speed of search operations.

Types of Indexes

1. Single Field Index

2. Compound Index
3. Unique Index
4. Text Index

Advantages

- Faster query execution
- Improved sorting performance
- Reduced scanning of documents
- Better database efficiency

Example

Create a collection **Student** with the following fields:

- RollNo
- Name
- Course
- Marks
- City

Insert 10 documents and perform Aggregation and Indexing operations.

Program

Step 1: Create Database

use CollegeDB

Output

switched to db CollegeDB

Step 2: Insert Documents

```
db.Student.insertMany([
{RollNo:1, Name:"Amit", Course:"BSc CS", Marks:85, City:"Pune"},
{RollNo:2, Name:"Priya", Course:"BSc CS", Marks:92, City:"Mumbai"},
{RollNo:3, Name:"Rahul", Course:"BCA", Marks:78, City:"Pune"},
{RollNo:4, Name:"Sneha", Course:"BCA", Marks:88, City:"Nashik"},
{RollNo:5, Name:"Neha", Course:"BSc CS", Marks:95, City:"Pune"},
{RollNo:6, Name:"Rohan", Course:"BCA", Marks:81, City:"Mumbai"},
{RollNo:7, Name:"Pooja", Course:"BSc CS", Marks:76, City:"Nashik"},
{RollNo:8, Name:"Karan", Course:"BCA", Marks:67, City:"Pune"},
```

{RollNo:9, Name:"Anjali", Course:"BSc CS", Marks:89, City:"Mumbai"},
{RollNo:10, Name:"Vikas", Course:"BCA", Marks:72, City:"Pune"}
])

Output

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

Aggregation Operations

Query 1: Calculate Average Marks Course-wise

```
db.Student.aggregate([
{
$group:
{
_id:"$Course",
AverageMarks:{$avg:"$Marks"}
}
}
])
```

Output

```
[
 { _id: 'BSc CS', AverageMarks: 87.4 },
 { _id: 'BCA', AverageMarks: 77.2 }
]
```

Query 2: Find Maximum and Minimum Marks Course-wise

```
db.Student.aggregate([
{
$group:
{
_id:"$Course",
MaximumMarks:{$max:"$Marks"},
MinimumMarks:{$min:"$Marks"}
}
}
```

])

Output

```
[
 { _id:'BSc CS', MaximumMarks:95, MinimumMarks:76 },
 { _id:'BCA', MaximumMarks:88, MinimumMarks:67 }
]
```

Query 3: Count Students in Each Course

```
db.Student.aggregate([
{
$group:
{
_id:"$Course",
TotalStudents:{$sum:1}
}
}
])
```

Output

```
[
 { _id:'BSc CS', TotalStudents:5 },
 { _id:'BCA', TotalStudents:5 }
]
```

Query 4: Display Students Scoring More Than 80 Marks

```
db.Student.aggregate([
{
$match:
{
Marks:{$gt:80}
}
}
])
```

Output

```
Amit   85
Priya  92
Sneha  88
Neha   95
Rohan  81
Anjali 89
```

Query 5: Sort Students by Marks (Descending)

```
db.Student.aggregate([
{
$sort:
{
Marks:-1
}
}
])
```

Output

```
Neha   95
Priya  92
Anjali 89
Sneha  88
Amit   85
Rohan  81
...
```

Indexing Operations

Query 6: Create Single Field Index

```
db.Student.createIndex(
{
Name:1
}
)
```

Output

```
Name_1
```

Query 7: Create Compound Index

```
db.Student.createIndex(
{
Course:1,
Marks:-1
}
)
```

Output

```
Course_1_Marks_-1
```

Query 8: Create Text Index

```
db.Student.createIndex(
```

```
{
Name:"text"
}
)
```
Output
Name_text

Query 9: Create Unique Index
```
db.Student.createIndex(
{
RollNo:1
},
{
unique:true
}
)
```
Output
RollNo_1

Query 10: Display All Indexes
```
db.Student.getIndexes()
```
Output
```
[
 { name: '_id_' },
 { name: 'Name_1' },
 { name: 'Course_1_Marks_-1' },
 { name: 'Name_text' },
 { name: 'RollNo_1' }
]
```

Query 11: Text Search Using Index
```
db.Student.find(
{
$text:
{
$search:"Amit"
}
}
)
```
Output

{
RollNo:1,
Name:'Amit',
Course:'BSc CS',
Marks:85,
City:'Pune'
}

Query 12: Query Performance Analysis

```
db.Student.find(
{
Name:"Amit"
}
).explain("executionStats")
```

Output

executionSuccess : true
totalDocsExamined : 1
totalKeysExamined : 1
executionTimeMillis : 0

Observation Table

Operation	**Result**
Average Marks Calculation	Successful
Maximum/Minimum Marks	Successful
Student Counting	Successful
Filtering Documents	Successful
Sorting Documents	Successful
Single Index Creation	Successful
Compound Index Creation	Successful
Text Index Creation	Successful
Unique Index Creation	Successful
Query Optimization	Improved

Result

Successfully implemented Aggregation Framework operations ($group, $match, $sort) and various Indexing Techniques (Single Field, Compound, Text, and Unique Indexes) in MongoDB. Query performance was improved through indexing, and aggregation operations provided meaningful analysis of student data.

Conclusion

The Aggregation Framework and Indexing Techniques were successfully implemented in MongoDB. Aggregation operations helped in data analysis, while indexes improved query performance and reduced document scanning.

SET A – Employee Management System

Problem Statement

Create a collection **Employee** with the following fields:

- EmpID
- Name
- Department
- Salary
- City
- Experience

Insert at least 10 employee records and perform the following operations.

Questions

1. Display all employee records.
2. Find employees whose salary is greater than ₹60,000.
3. Calculate the average salary department-wise using Aggregation Framework.
4. Find the highest salary in each department.
5. Count the number of employees in each city.
6. Display employees having more than 5 years of experience.
7. Sort employees by salary in descending order.
8. Create a Single Field Index on Employee Name.
9. Create a Unique Index on EmpID.
10. Analyze query performance before and after indexing using explain().

SET B – Movie Database Management System

Problem Statement

Create a collection **Movie** with the following fields:

- MovieID
- MovieName
- Genre
- Rating
- ReleaseYear
- Language

Insert at least 10 movie records and perform the following operations.

Questions

1. Display all movie records.
2. Find movies released after the year 2020.
3. Calculate the average movie rating genre-wise.
4. Find the highest-rated movie in each genre.
5. Count the number of movies available in each language.
6. Display movies with ratings greater than 8.5.
7. Sort movies by release year in ascending order.
8. Create a Text Index on MovieName.
9. Search a movie using Text Search.
10. Create a Compound Index on Genre and Rating and analyze query performance.

SET C – Product Inventory Management System

Problem Statement

Create a collection **Product** with the following fields:

- ProductID
- ProductName
- Category
- Price
- Quantity

- Supplier

Insert at least 10 product records and perform the following operations.

Questions

1. Display all product records.
2. Find products whose quantity is less than 20.
3. Calculate the total quantity available category-wise.
4. Find the average product price category-wise.
5. Determine the most expensive product in each category.
6. Count the number of products supplied by each supplier.
7. Sort products by price in descending order.
8. Create a Single Field Index on ProductName.
9. Create a Compound Index on Category and Price.
10. Create a Unique Index on ProductID and compare query execution statistics using explain().

Assignment Evaluation

0:Not Done	☐	2:Late Complete	☐	4:Complete	☐
1:Incomplete	☐	3:Needs Improvement	☐	5: Well Done	☐

Signature of the instructor: _____ Date :_____

Assignment No.5

Modeling of Graph Data using Nodes and Relationship in Neo4j (Total Slots = 2)

What is a Graph Database?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. It is composed of two elements - nodes (vertices) and relationships

(edges).

Graph database is a database used to model the data in the form of graph. In here, the nodes of a graph depict the entities while the relationships depict the association of these nodes.

Popular Graph Databases

Neo4j is a popular Graph Database. Other Graph Databases are Oracle NoSQL Database,OrientDB, HypherGraphDB, GraphBase, InfiniteGraph, and AllegroGraph.

RDBMS Vs Graph Database

Following is the table which compares Relational databases and Graph databases.

Sr.No.	RDBMS	Graph Databse
1.	Tables	Graphs
2.	Rows	Nodes
3.	Columns and Data	Properties and its values
4.	Constraints	Relationships
5.	Joins	Traversal

Advantages of Neo4j

Following are the advantages of Neo4j.

- Flexible data model: Neo4j provides a flexible simple and yet powerful data model, which can be easily changed according to the applications and industries.
- Real-time insights: Neo4j provides results based on real-time data.
- High availability: Neo4j is highly available for large enterprise real-time applicationswith transactional guarantees.
- Connected and semi structures data: Using Neo4j, you can easily represent connected and semi-structured data.
- Easy retrieval: Using Neo4j, you can not only represent but also easily retrieve(traverse/navigate) connected data faster when compared to other databases.

- Cypher query language: Neo4j provides a declarative query language to represent the graph visually, using an ascii-art syntax. The commands of this language are in human readable format and very easy to learn.
- No joins: Using Neo4j, it does NOT require complex joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without joins or indexes.

Features of Neo4j

Following are the notable features of Neo4j -

- Data model (flexible schema): Neo4j follows a data model named native property graph model. Here, the graph contains nodes (entities) and these nodes are connected with each other (depicted by relationships). Nodes and relationships store data in key- value pairs known as properties.
- ACID properties: Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.
- Scalability and reliability: You can scale the database by increasing the number of reads/writes, and the volume without effecting the query processing speed and data integrity. Neo4j also provides support for replication for data safety and reliability.
- Cypher Query Language: Neo4j provides a powerful declarative query languageknown as Cypher. It uses ASCII-art for depicting graphs. Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like Joins.
- Built-in web application: Neo4j provides a built-in Neo4j Browser web application. Using this, you can create and query your graph data.

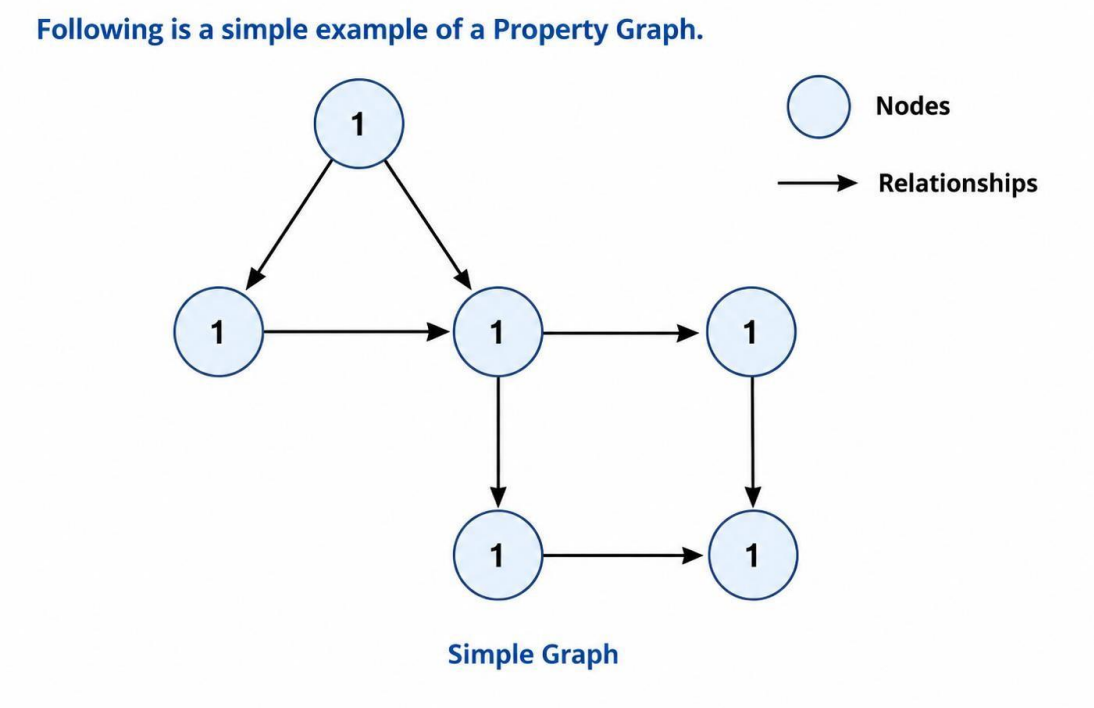
The main building blocks of Graph DB Data Model are:

 Nodes

 Relationships

 Properties

Following is a simple example of a Property Graph.



Here, we have represented Nodes using Circles. Relationships are represented using Arrows. Relationships are directional. We can represent Node's data in terms of Properties (key-value pairs). In this example, we have represented each Node's Id property within the Node's Circle

Node

Node is a fundamental unit of a Graph. It contains properties with key-value pairs as shown in the following image

Here, Node Name = "Employee" and it contains a set of properties as key-value pairs.

Properties

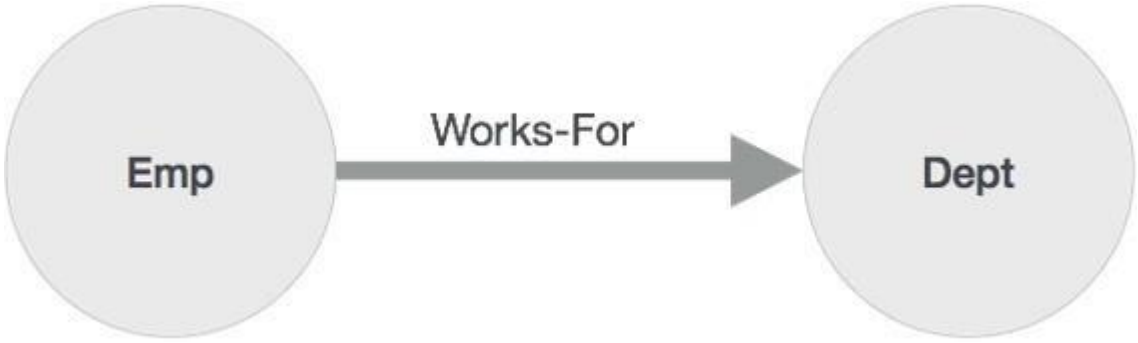
Property is a key-value pair to describe Graph Nodes and Relationships.

Key = Value

Where Key is a String and Value may be represented using any Neo4j Data types

Relationships

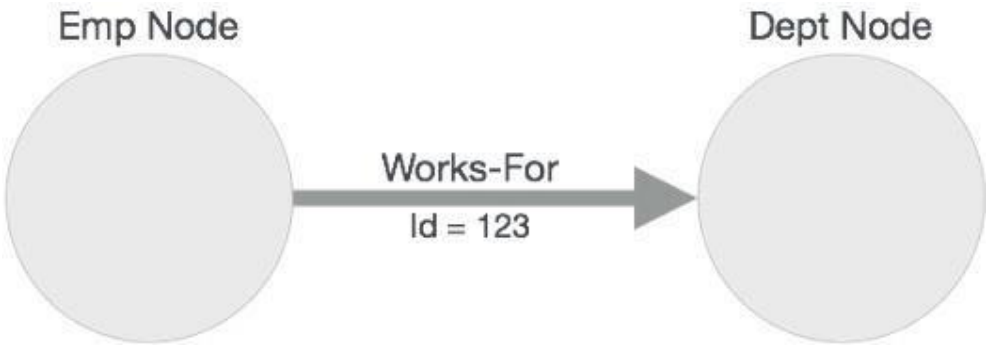
Relationships are another major building block of a Graph Database. It connects two nodes as depicted in the following figure.



Here, Emp and Dept are two different nodes. "WORKS_FOR" is a relationship between Emp and Dept nodes. As it denotes, the arrow mark from Emp to Dept, this relationship describes –

Emp WORKS_FOR Dept

Each relationship contains one start node and one end node. Here, "Emp" is a start node, and "Dept" is an end node. As this relationship arrow mark represents a relationship from "Emp" node to "Dept" node, this relationship is known as an "Incoming Relationship" to "Dept" Node and "Outgoing Relationship" to "Emp" node. Like nodes, relationships also can contain properties as key-value pairs



Here, "WORKS_FOR" relationship has one property as key-value pair.

Id=12

3 It represents an Id of this relationship

Labels

Label associates a common name to a set of nodes or relationships. A node or relationship can contain one or more labels. We can create new labels to existing nodes or relationships. We can remove the existing labels from the existing nodes or relationships. From the previous diagram, we can observe that there are two nodes. Left side node has a Label: "Emp" and the right side node has a Label: "Dept". Relationship between those two nodes also has a Label: "WORKS_FOR". Note: Neo4j stores data in Properties of Nodes or Relationships.

CREATE

a node is a data/record in a graph database. You can create a node in Neo4j using the CREATE clause.

Creating a Single node

You can create a node in Neo4j by simply specifying the name of the node that is to be created along with the CREATE clause.

Syntax

Following is the syntax for creating a node using Cypher Query Language.

CREATE (node_name);

Note − Semicolon (;) is optional.

Example

Following is a sample Cypher Query which creates a node in Neo4j.

CREATE (sample)

Display

Syntax :

MATCH (a:sample)
RETURN a;

Creating a Node with a Label

A label in Neo4j is used to group (classify) the nodes using labels. You can create a label for a node in Neo4j using the CREATE clause.

Syntax

Following is the syntax for creating a node with a label using Cypher Query Language.

CREATE (node:label)

Example

Following is a sample Cypher Query which creates a node with a label.

CREATE (Dhawan:player)

Creating a Node with Multiple Labels

You can also create multiple labels for a single node. You need to specify the labels for the node by separating them with a colon : .

Syntax

Following is the syntax to create a node with multiple labels.

CREATE (node:label1:label2:.....labeln)

Example

Following is a sample Cypher Query which creates a node with multiple labels in Neo4j.

CREATE (Dhawan:person:player)

Create Node with Properties

Properties are the key-value pairs using which a node stores data. You can create a node with properties using the CREATE clause. You need to specify these properties separated by commas within the flower braces { }.

Syntax

Following is the syntax to create a node with properties.

CREATE (node:label { key1: value, key2: value, ... })

Example

Following is a sample Cypher Query which creates a node with properties.

CREATE (Dhawan:player{name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})

Verification

To verify the creation of the node, type and execute the following query in the dollar prompt.

MATCH (n) RETURN n

Creating Relationships

We can create a relationship using the CREATE clause. We will specify relationship within the square braces [ ] depending on the direction of the relationship it is placed between hyphen - and arrow → as shown in the following syntax.

Syntax

Following is the syntax to create a relationship using the CREATE clause.

CREATE (node1)-[:RelationshipType]->(node2)

Example

First of all, create two nodes Ind and Dhawan in the database, as shown below.

CREATE (Dhawan:player{name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})

CREATE (Ind:Country {name: "India"})

Now, create a relationship named BATSMAN_OF between these two nodes as −

CREATE (Dhawan)-[r:BATSMAN_OF]->(Ind)

Finally, return both the nodes to see the created relationship.

RETURN Dhawan, Ind

Creating a Relationship with Label and Properties

You can create a relationship with label and properties using the CREATE clause.

Syntax

Following is the syntax to create a relationship with label and properties using the CREATE clause.

CREATE (node1)-[label:Rel_Type {key1:value1, key2:value2, . . . n}]-> (node2)

Example

Following is a sample Cypher Query which creates a relationship with label and properties.

MATCH (a:player), (b:Country) WHERE a.name = "Shikar Dhawan" AND b.name = "India"

CREATE (a)-[r:BATSMAN_OF {Matches:5, Avg:90.75}]->(b)

RETURN a,b

SET A

Consider a Song database, with labels as Artists, Song, Recording_company, Recording_studio, Song_author etc.

Relationships can be as follows:

- Artist → [Performs] → Song
- Song → [Written by] → Song_author
- Song → [Recorded in] → Recording_Studio
- Recording_Studio → [Managed by] → Recording_Company
- Recording_Company → [Finances] → Song

Simple Cypher Queries for

1. Display all Artists.
2. Display all Songs.
3. Display Artist and the Songs they Perform.
4. List the names of songs written by "_____".
5. List the names of record companies who have financed the song "_____"

6. List the names of artists performing the song "_____".
7. Name the songs recorded by the studio "_____"..
8. Display Song, Studio, and Company.
9. Display Complete Song Information.

SET B

Consider the Movie database with nodes as Actor,Movie,Role, Financier, Producer, Director. Assume appropriate relationship between the node include properties for nodes and relationship.

- Actor → ACTS_IN → Movie
- Actor → PLAYS → Role
- Director → DIRECTS → Movie
- Producer → PRODUCES → Movie
- Financier → FINANCES → Movie

Queries

1. List all movies.
2. List all actors.
3. List movies acted by an actor.
4. List movies directed by a director.
5. List producers of a movie.
6. List actors, movies and directors.
7. Find all actors who have acted in a movie ".....".
8. Find all movies produced by ". ....".
9. Complete Movie Information.

SET C

Create a Social network database, with labels as Person, Affiliations, Groups, Story, Timeline etc. Some of the relationships can be as follows:

Person friend of] Person [affiliated to] affiliations
Person [belongs to] Groups, Person→[create] Story [refers to] Person
Person(creates]→ Timeline [reference for] Story. Timeline [contains]→Messages

Queries

1. List all persons.
2. List friends of a person.
3. List affiliations of a person.
4. Find all friends of "John", along with the year, since when john knows them.
5. List out the affiliations of John.
6. Display Complete Social Network Graph.
7. Display all Friend Relationships.

Assignment Evaluation

0:Not Done		2:Late Complete		4:Complete	
1:Incomplete		3:Needs Improvement		5: Well Done	

Signature of the instructor: _____ **Date :**_____

Assignment No. 6

Implementation of Graph Queries and Relationship Analysis using Cypher in Neo4j (Total Slots = 2)

Aim

To implement graph queries and perform relationship analysis using Cypher Query Language in Neo4j Graph Database.

Objectives

1. To understand graph databases and Neo4j.
2. To create nodes and relationships using Cypher.
3. To perform graph traversal and pattern matching.
4. To analyze relationships between entities.
5. To retrieve connected data efficiently using graph queries.

Installation of Neo4j

1. Download Neo4j Community Edition from:
    - Neo4j Download Center
2. Install Java JDK 17 or above.
3. Install Neo4j Desktop.
4. Create a local database.
5. Start Neo4j Browser.

Verify installation:

```
RETURN "Neo4j Installed Successfully";
```

Expected Output:

```
Neo4j Installed Successfully
```

Introduction

Neo4j is a popular open-source Graph Database Management System (GDBMS) that stores data in the form of nodes, relationships, and properties. Unlike traditional relational databases that store data in tables and rows, Neo4j represents data as a graph, making it highly efficient for managing and analyzing interconnected data.

Graph databases are particularly useful when relationships between data entities are as important as the data itself. Examples include social networks, recommendation systems, fraud detection systems, transportation networks, and knowledge graphs.

Graph Database

A graph database is a NoSQL database that uses graph structures for semantic queries.

A Graph Database stores data in the form of:

- Nodes (Entities)
- Relationships (Connections)
- Properties (Attributes)

1. Nodes

Nodes represent entities or objects in the graph.

Examples:

- Student
- Employee
- Course
- Product
- Movie

Example:

(Student)

2. Relationships

Relationships connect nodes and define how entities are related to each other.

Examples:

- ENROLLED_IN
- FRIEND_OF
- WORKS_ON
- PURCHASED

Example:

(Student) ---- ENROLLED_IN ----> (Course)

3. Properties

Properties are attributes associated with nodes or relationships and are stored as key-value pairs.

Example:

(:Student {id:1, name:'Amit', city:'Pune'})

Here:

- id = 1
- name = Amit
- city = Pune

Neo4j Graph Database

Neo4j is based on the Property Graph Model, where both nodes and relationships can contain properties. It provides a flexible schema and supports efficient graph traversal operations.

Features of Neo4j

- High performance graph traversal
- ACID-compliant transactions
- Flexible schema design
- Interactive graph visualization
- Efficient relationship management
- Scalable and reliable architecture
- Powerful query language (Cypher)

Relational Database vs Graph Database

Relational Database	Graph Database
Data stored in tables	Data stored as nodes and relationships
Uses JOIN operations	Uses graph traversal
SQL queries	Cypher queries
Fixed schema	Flexible schema
Slower for connected data	Faster for connected data

Cypher Query Language

Cypher is Neo4j's declarative query language used to create, update, delete, and retrieve graph data. It uses an intuitive syntax based on graph patterns, making graph operations simple and readable.

Basic Cypher Commands

Command	Purpose
CREATE	Creates nodes and relationships
MATCH	Retrieves data from the graph
WHERE	Applies conditions to filter data
RETURN	Displays query results
SET	Updates node properties
DELETE	Removes nodes or relationships

Creating Nodes

Nodes can be created using the CREATE command.

Example:

```
CREATE (:Student {id:1, name:'Amit'})
```

This creates a Student node with properties id and name.

Creating Relationships

Relationships connect two nodes.

Example:

CREATE (s)-[:ENROLLED_IN]->(c)

Graph Representation:

(Student) ----> (Course)
ENROLLED_IN

Pattern Matching

Pattern matching is one of the most important features of Cypher. It allows users to search for nodes and relationships based on specific patterns.

Example:

MATCH (s:Student)-[:ENROLLED_IN]->(c:Course)
RETURN s,c

This query displays students and the courses in which they are enrolled.

Graph Traversal

Graph traversal refers to navigating from one node to another through relationships. Neo4j performs graph traversal efficiently without requiring complex JOIN operations.

Example:

Amit → Priya → Rahul

Traversal helps find:

- Direct relationships
- Indirect relationships
- Mutual connections

- Shortest paths

Relationship Analysis

Relationship analysis involves examining the connections between entities to discover patterns and insights.

Examples:

- Students enrolled in the same course
- Employees working on common projects
- Friends in a social network
- Customers purchasing similar products

Relationship analysis helps in:

- Recommendation systems
- Social network analysis
- Fraud detection
- Network optimization

Variable-Length Relationships

Cypher supports traversal across multiple relationship levels using variable-length patterns.

Example:

```
MATCH (a)-[:FRIEND*2]->(b)
RETURN b
```

This query finds friends of friends by traversing two FRIEND relationships.

Shortest Path Analysis

Neo4j can determine the shortest path between two nodes.

Example:

```
MATCH p=shortestPath((a)-[*]-(b))
RETURN p
```

Applications:

- Route planning
- Social networking
- Communication networks
- Logistics management

Observation Table

Operation	Observation
Node Creation	Successful
Relationship Creation	Successful
Pattern Matching	Successful
Graph Traversal	Successful
Relationship Analysis	Successful
Query Execution	Successful

Result

Successfully implemented graph queries and relationship analysis using Cypher in Neo4j. Nodes and relationships were created, queried, and analyzed using graph traversal and pattern matching techniques.

Conclusion

Neo4j is a powerful graph database that enables efficient storage, retrieval, and analysis of highly connected data. Using Cypher Query Language, users can create nodes, establish relationships, perform graph traversal, and analyze complex networks. It is widely used in social networks, recommendation systems, fraud detection, healthcare, and IoT applications due to its ability to manage and analyze relationships effectively. This experiment helps students understand graph databases and perform relationship analysis using Cypher queries in Neo4j.

Applications of Neo4j

1. Social Networking Sites
2. Recommendation Systems
3. Fraud Detection Systems
4. Healthcare Networks
5. Supply Chain Management

6. Network and Telecommunications
7. IoT and Smart Systems

SET A – Student-Course Relationship Analysis

Problem Statement

Create nodes for Students and Courses and establish enrollment relationships.

Fields

Student

- StudentID
- Name
- City

Course

- CourseID
- CourseName

Questions

1. Create Student nodes.
2. Create Course nodes.
3. Create ENROLLED_IN relationships between students and courses.
4. Display all Student nodes.
5. Display all Course nodes.
6. Find students enrolled in a particular course.
7. Display courses taken by a specific student.
8. Count the number of students enrolled in each course.
9. Find students from a particular city.
10. Display the complete graph structure.

SET B – Social Network Analysis

Problem Statement

Create a social network graph using Person nodes and FRIEND relationships.

Fields

Person

- PersonID
- Name
- City

Questions

1. Create Person nodes.
2. Create FRIEND relationships between persons.
3. Display all persons in the network.
4. Find direct friends of a specific person.
5. Find mutual friends between two persons.
6. Find friends of friends using variable-length relationships.
7. Count the number of friends each person has.
8. Display persons belonging to a specific city.
9. Find the shortest connection path between two persons.
10. Display the complete social network graph.

SET C – Employee-Project Management System

Problem Statement

Create a graph database representing employees working on various projects.

Fields

Employee

- EmpID
- EmpName
- Department

Project

- ProjectID
- ProjectName
- Technology

Questions

1. Create Employee nodes.
2. Create Project nodes.

3. Create WORKS_ON relationships between employees and projects.
4. Display all employees.
5. Display all projects.
6. Find employees working on a specific project.
7. Display projects assigned to a specific employee.
8. Count employees working on each project.
9. Find projects based on a specific technology.
10. Display the complete Employee–Project graph.

Assignment Evaluation

0:Not Done		2:Late Complete		4:Complete	
1:Incomplete		3:Needs Improvement		5: Well Done	

Signature of the instructor: _____ Date :_____