UINT-5

MongoDB: Understanding NoSQL and MongoDB, Getting Started with MongoDB, Getting Started with MongoDB and Node.js, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js.

Q) What is NoSQL? What is the need of NoSQL? Explain different types of NoSQL databases.

NoSQL Stands for **N**ot **O**nly **SQL.** These are non-relational, open source, distributed databases.

Features of NoSQL:

1. NoSQL databases are non-relational: They do not adhere to relational data model. In fact either key-value pairs or document oriented or column oriented or graph based databases.
2. Distributed: The data is distributed across several nodes in a cluster constituted of low commodity hardware.
3. No Support for ACID properties: They do not offer support for ACID properties of transactions. On the contrary, they adherence to CAP theorem.
4. No fixed table schema: NoSQL databases are becoming increasing popular owing to their support for flexibility to the schema. They do not mandate for the data to strict adhere to any schema structure at the time of storage.

Need of NoSQL:

1. It has scale out architecture instead of the monolithic architecture of relational databases.
2. It can house large volumes of structured, semi-structured and unstructured data.
3. Dynamic Schema: It allows insertion of data without a predefined schema.
4. Auto Sharding: It automatically spread data across an arbitrary numer of servers or nodes in a cluster.
5. Replication: It offers good support for replication which in turn guarantees high availability, fault tolerance and disaster recovery.

Types of NoSQL databases: They broadly divided into Key-Value or big hash table and Schemal-less.

1. **Key-Value:** It maintains a big hash table of keys and values.
   Key are unique.
   It is fast, scalable and fault tolerance.
   It can't model more complex data structure such as objects
   Eg. Dynamo, Redis, Riak etc.
   Sample Key-Value pair database:

Key	Value
Fname	Praneeth
Lname	Ch

2. **Document:** It maintains data in collections constituted of documents.
   Eg. MongoDB, Apace CouchDB, Couchbase, MarkLogic etc.
   Sample Document in Document DB:

```
{
        "Book Name": "Big Data and Analytics",
        "Publisher": "Wiley India",
        "Year": "2015"
}
```

3. **Column:** Each storage block has data from only one column. It only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation à data locality
   Eg. Cassandra, HBase etc.

   Sample column database:
   UserProfile = {
   Cassandra = { emailAddress:"casandra@apache.org" , age:"20"}
   TerryCho = { emailAddress:"terry.cho@apache.org" , gender:"male"}
   Cath = { emailAddress:"cath@apache.org" ,
   age:"20",gender:"female",address:"Seoul"}
   }

4. **Graph:** They are also called Network database. A graph stores data in nodes.
   Data model:
   - (Property Graph) nodes and edges
     - Nodes may have properties (including ID)
     - Edges may have labels or roles
   - Key-value pairs on both

   Eg. Neo4j, HyperGraphDB, InfiniteGraph etc.

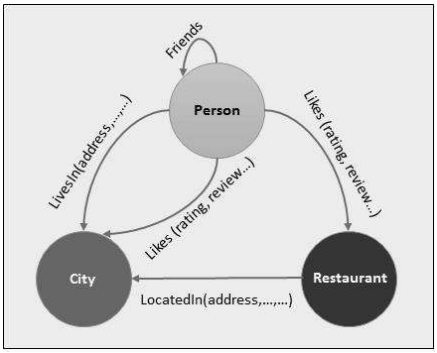


Fig. Sample Graph Database

Q) What are the advantages and disadvantage of NoSQL?

Advantages:

- Big Data Capability
- No Single Point of Failure
- Easy Replication
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- NoSQL databases don't need a dedicated high-performance server
- It can serve as the primary data source for online applications.
- Excels at distributed database and multi-data centre operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages:

- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- Open source options so not so popular for enterprises.
- No support for join and group-by operations.

Q) Differentiate SQL and MongoDB.

RDBMS	MongoDB
Relational database management system	Document-oriented NoSQL database
Stores data in tables with fixed schemas	Stores data in flexible documents with dynamic schemas
Uses SQL (Structured Query Language) for querying and manipulating data	Uses its own query language and also supports full-text search
Supports complex transactions with ACID (Atomicity, Consistency, Isolation, Durability) properties	Does not support ACID transactions but offers atomic operations on a single document

Ensures data consistency with referential integrity constraints and foreign keys	Does not enforce referential integrity but supports embedded documents and document references for data modeling
Suitable for applications that require strong consistency and predefined schemas	Suitable for applications that require scalability, performance, and flexibility with semi-structured or unstructured data
Uses vertical scaling to increase performance by adding more powerful hardware	Uses horizontal scaling to increase performance by distributing data across multiple servers

Q) Explain advantages and disadvantages of MongoDB

Advantages of MongoDB:

Flexible data model: MongoDB's document-oriented data model is flexible and dynamic, allowing for easy handling of semi-structured and unstructured data. This makes it suitable for applications with changing or unpredictable data requirements.

Scalability: MongoDB can scale horizontally by sharding (distributing data across multiple servers) and replicating data across nodes. This allows it to handle large amounts of data and traffic with high availability.

Performance: MongoDB's indexing and querying capabilities, along with its ability to store related data within the same document, make it performant for certain use cases, especially those involving complex queries.

Developer-friendly: MongoDB is easy to set up and use, with a flexible schema that allows for rapid prototyping and iteration. Its query language is also designed to be developer-friendly and easily understandable.

Community and ecosystem: MongoDB has a large and active community, with many third-party tools and libraries available for integration.

Disadvantages of MongoDB:

No ACID transactions: MongoDB does not support ACID transactions, which can be a disadvantage for certain applications that require strong consistency guarantees.

Memory usage: MongoDB's memory usage can be high, especially when dealing with large collections or indexes. This can lead to higher hardware costs or slower performance on systems with limited resources.

Limited query functionality: Although MongoDB has a powerful query language, it lacks some of the advanced querying features that are available in SQL-based databases. For example, there is no support for joins, and certain queries can be slow on large datasets.

Indexing overhead: Creating and maintaining indexes in MongoDB can be a resource-intensive operation, which can impact write performance and storage requirements.

Relatively new technology: MongoDB is a relatively new technology compared to SQL-based databases, which means that there may be less industry expertise and established best practices available.

Q) Explain different data types in MongoDB

MongoDB is a document-oriented NoSQL database that stores data in JSON-like documents. MongoDB supports several data types that can be used to represent different types of data in a document.

String, Number(integers and folating-point numbers),Boolean,Date,Array,Null,Timestamp,Binary data, Regular expressions, Object ID

Eg.

```
{
  "_id": ObjectId("627cdd77d11c334c6d43b85e"),
  "name": "John Smith",
  "age": 35,
  "is_active": true,
  "score": 8.5,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "state": "NY"
  },
  "phone_numbers": ["555-1234", "555-5678"],
  "last_login": ISODate("2022-05-10T10:30:00Z"),
  "profile_picture": new BinData(0, "iVBORw0KGg..."),
  "interests": ["programming", "reading", "traveling"],
  "is_admin": false,
  "created_at": Timestamp(1647088549, 1)
}
```

Q) Explain CRUD operations in MongoDB.

1. Create:
view all existing databases:
show dbs

create or change a db:
use chp

creating a collection(table):
db.createCollection("fac")

displaying collections:
show collections

2. Read:
You can also create a collection during the insert process:This will create the "fac" collection if it does not already exist.

```
db.fac.insertOne({
  name: "chp",
  branch: "it",
  sal: 10000,
  domain: ["bda", "ml"],
  date: Date()
})
```

insert multiple documents:

```
db.fac.insertMany([
  {
  name: "gr",
  branch: "it",
  sal: 20000,
  domain: ["wt", "java"],
  date: Date()
},
  {
  name: "vr",
  branch: "cse",
  sal: 30000,
  domain: ["rl", "react"],
  date: Date()
}
])
```

Find Data
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.

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.
Note: This method only returns the first match it finds.

db.fac.find({domain: "ml"})
db.fac.find({ sal: 12000 })
db.fac.find({ sal: { $gt: 20000 } })

Projection:
db.fac.find({}, {name:1, date: 1})

3. Update:
update an existing document we can use the updateOne() or updateMany() methods.

The first parameter is a query object to define which document or documents should be updated.
The second parameter is an object defining the updated data.

updateOne()
The updateOne() method will update the first document that is found matching the provided query.

db.fac.updateOne({ name: "chp" }, { $set: { sal: 25000 } })

updateMany()
The updateMany() method will update all documents that match the provided query.
db.fac.updateMany({}, { $inc: { sal: 1000 } })

4. Delete:
db.fac.deleteOne({ name: "gr" })
db.fac.deleteMany({ domain: "ml" })

drop a collection:
db.fac.drop()

drop a database:
db.dropDatabase()

Q) Differentiate inserting and upserting with an example.

Inserting: Inserting a document in MongoDB means adding a new document to a collection. If a document with the same _id already exists in the collection, the insertion operation will fail with a duplicate key error. Inserting a document does not modify any existing documents in the collection.

Eg.

```
db.fac.insertOne({
  name: "chp",
  branch: "it",
  sal: 10000,
  domain: ["bda", "ml"],
  date: Date()
})
```

Upserting: Upserting in MongoDB means updating a document if it exists or inserting a new document if it does not exist. If a document with the same _id already exists in the collection, the upsert operation will update the existing document with the specified fields. If a document with the same _id does not exist in the collection, the upsert operation will insert a new document with the specified fields.

Eg.

```
db.fac.updateOne(
  { name: "pnr" },
  {
    $set:
      {
        name: "pnr",
          branch: "cse",
          sal: 40000,
          domain: ["ml", "se"],
          date: Date()
      }
  },
  { upsert: true }
)
```

Q) Explain the steps involved in Node.js application that connect to MongoDB database with an example.

1. Import the MongoClient module from the 'mongodb' package

2. Define the URL for the MongoDB instance to connect to
3. Call the connect() method of the MongoClient object, passing in the URI
4. Handle the resolved Promise in a .then() callback and output a success message
5. Handle any errors that occur during the connection process in a .catch() callback and output an error message
6. Close the client connection

Eg.

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://127.0.0.1:27017';

MongoClient.connect(url)
  .then(client => {
    console.log("Connected to database!");
    console.log("Database name:", client.db().databaseName);
    client.close();
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

Output:

```
node sample.js
Connected to database!
Database name: test
```

Q) Write a program to access documents in MongoDB from Node.js

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

console.log("Before connecting to database");

MongoClient.connect(uri)
  .then(client => {
    console.log("Connected to database!");
    console.log("Database name:", client.db().databaseName);
    //creating a database
    var dbo = client.db("mydb9");
    console.log("Database name:", dbo.databaseName);
    //creating a collection "fac"
    dbo.createCollection("fac")
      .then(() => {
        //inserting document
        return dbo.collection("fac").insertOne({
          name: "chp",
          branch: "it",
          sal: 10000,
```

```
        domain: ["bda", "ml"],
        date: Date()
      });
    })
    .then(() => {
      return dbo.collection("fac").insertMany([
        {
        name: "gr",
        branch: "it",
        sal: 20000,
        domain: ["wt", "java"],
        date: Date()
      },
        {
        name: "vr",
        branch: "cse",
        sal: 30000,
        domain: ["rl", "react"],
        date: Date()
      }
      ])

    })

    .then(() => {
      // retrieving updated document
      return dbo.collection("fac").find().toArray();;
    })
    .then(result => {
      console.log("Updated document:", result);
      client.close();
    })
    .catch(err => {
      console.log("Error performing operation:", err);
    });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

O/P:

```
node acessing.js
Before connecting to database
Connected to database!
Database name: test
Database name: mydb9
Updated document: [
  {
    _id: new ObjectId("645c711a4abcb3378a73da6a"),
```

```
    name: 'chp',
    branch: 'it',
    sal: 10000,
    domain: [ 'bda', 'ml' ],
    date: 'Thu May 11 2023 10:07:46 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("645c711a4abcb3378a73da6b"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Thu May 11 2023 10:07:46 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("645c711a4abcb3378a73da6c"),
    name: 'vr',
    branch: 'cse',
    sal: 30000,
    domain: [ 'rl', 'react' ],
    date: 'Thu May 11 2023 10:07:46 GMT+0530 (India Standard Time)'
  }
]
```

Q) Write a program to Manipulating MongoDB documents from Node.js.

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {
    console.log("Connected to database!");
    console.log("Database name:", client.db().databaseName);
    //creating a database
    var dbo = client.db("mydb99");
    console.log("Database name:", dbo.databaseName);
    //creating a collection "fac"
    dbo.createCollection("fac")
      .then(() => {
        //inserting document
        return dbo.collection("fac").insertOne({
          name: "chp",
          branch: "it",
          sal: 10000,
          domain: ["bda", "ml"],
          date: Date()
        });
      })
```

```
    .then(() => {
      return dbo.collection("fac").insertMany([
        {
        name: "gr",
        branch: "it",
        sal: 20000,
        domain: ["wt", "java"],
        date: Date()
      },
        {
        name: "vr",
        branch: "cse",
        sal: 30000,
        domain: ["ml", "react"],
        date: Date()
      }
      ])

    })
    .then(() => {
      // updating document
      return dbo.collection("fac").updateOne( { name: "chp" }, { $set: { sal:
25000 } } );
    })

    .then(() => {
      // retrieving updated document
      return dbo.collection("fac").find().toArray();;
    })
    .then(result => {
      console.log("Updated document:", result);

    })

    .then(() => {
      // deleting documents with domain as ml
      return dbo.collection("fac").deleteMany({ domain: "ml" });
    })

    .then(() => {
      // retrieving updated document
      return dbo.collection("fac").find().toArray();;
    })
    .then(result => {
      console.log("Updated document:", result);
      client.close();
    })

    .catch(err => {
```

```
        console.log("Error performing operation:", err);
      });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

O/P:

```
node manipulating.js
Connected to database!
Database name: test
Database name: mydb99
Updated document: [
  {
    _id: new ObjectId("645c718d8b0ee669325ede25"),
    name: 'chp',
    branch: 'it',
    sal: 25000,
    domain: [ 'bda', 'ml' ],
    date: 'Thu May 11 2023 10:09:41 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("645c718d8b0ee669325ede26"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Thu May 11 2023 10:09:41 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("645c718d8b0ee669325ede27"),
    name: 'vr',
    branch: 'cse',
    sal: 30000,
    domain: [ 'ml', 'react' ],
    date: 'Thu May 11 2023 10:09:41 GMT+0530 (India Standard Time)'
  }
]
Updated document: [
  {
    _id: new ObjectId("645c718d8b0ee669325ede26"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Thu May 11 2023 10:09:41 GMT+0530 (India Standard Time)'
  }
]
```

Q) Write a program to check status of database server and collection statistics.

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {
    // Check server status
    client.db().admin().serverStatus()
      .then(status => {
        console.log("Server status:", status);
      })
      .catch(err => {
        console.log("Error getting server status:", err);
      });

    // Select database
    var dbo = client.db("mydb3");

    // Check collection status
    dbo.command({ collStats: "fac" })
      .then(stats => {
        console.log("Collection status:", stats);
      })
      .catch(err => {
        console.log("Error getting collection status:", err);
      });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

Q) Write a program to retrieve Specific documents from a collection.

a. find documents with branch as “it”

b. find documents with name that contains letter “r”

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {
    const dbo = client.db("mydb9");
    const query1 = { branch: "it" };
    const query2 = { name: { $in: [/r/] } };
    dbo.collection("fac").find(query1).toArray()
      .then(result => {
```

```
      console.log("Result with branch as it:", result);

      dbo.collection("fac").find(query2).toArray()
        .then(result => {
          console.log("Result with name containing 'r':", result);
          client.close();
        })
        .catch(err => {
          console.log("Error retrieving data:", err);
        });
    })
    .catch(err => {
      console.log("Error retrieving data:", err);
    });
})
.catch(err => {
  console.log("Error connecting to database:", err);
});
```

O/P:

```
node specific_doc.js
Result with branch as it: [
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f20"),
    name: 'chp',
    branch: 'it',
    sal: 25000,
    domain: [ 'bda', 'ml' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f21"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
```

```
  }
]
Result with name containing 'r': [
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f21"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f22"),
    name: 'vr',
    branch: 'cse',
    sal: 30000,
    domain: [ 'rl', 'react' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  }
]
```

Q) Write a program in Node.js to count no.of documents in a collection.

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017/mydb9';

MongoClient.connect(uri)
  .then(client => {
    console.log("Connected to database!");
    var dbo = client.db();
    var collection = dbo.collection('fac');
    collection.countDocuments()
      .then(count => {
        console.log(`The 'fac' collection has ${count} documents.`);
        client.close();
      })
```

```
    .catch(err => {
      console.log("Error getting documents count:", err);
      client.close();
    });
 })
 .catch(err => {
   console.log("Error connecting to database:", err);
 });
```

O/P:

node countdoc.js

Connected to database!

The 'fac' collection has 3 documents.

Q) Write a program in Node.js to sort documents in descending order on sal field.

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {
    console.log("Connected to database!");
    const db = client.db("mydb9");
    const collection = db.collection("fac");

    collection.find().sort({ sal: -1 }).toArray()
      .then(docs => {
        console.log(`Found ${docs.length} documents:`);
        console.log(docs);
        client.close();
      })
      .catch(err => {
        console.log("Error retrieving documents:", err);
        client.close();
      });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

O/P:

node sorting.js

Connected to database!

Found 3 documents:

```
[
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f22"),
    name: 'vr',
    branch: 'cse',
    sal: 30000,
    domain: [ 'rl', 'react' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f20"),
    name: 'chp',
    branch: 'it',
    sal: 25000,
    domain: [ 'bda', 'ml' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f21"),
    name: 'gr',
    branch: 'it',
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  }
]
```

Q) Write a program to limit no.of documents of a collection.

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {

    var dbo = client.db("mydb9");
    //finding all documents in collection "fac"
    dbo.collection("fac").find({}).sort({ name: 1 }).limit(2).toArray()
      .then(result => {
        console.log("Result:", result);
        client.close();
      })
      .catch(err => {
        console.log("Error performing operation:", err);
      });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

O/P:

```
node limiting.js

Result: [
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f20"),
    name: 'chp',
    branch: 'it',
    sal: 25000,
    domain: [ 'bda', 'ml' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  },
  {
    _id: new ObjectId("6459c2d1c5dafd86e6345f21"),
    name: 'gr',
    branch: 'it',
```

```
    sal: 20000,
    domain: [ 'wt', 'java' ],
    date: 'Tue May 09 2023 09:19:37 GMT+0530 (India Standard Time)'
  }
]
```

Q) Write query execution plan in mongoDB

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

MongoClient.connect(uri)
  .then(client => {
    var dbo = client.db("mydb9");

    // Finding all documents in collection "fac" and explaining the query
    dbo.collection("fac").find({}).sort({ name: 1 }).limit(2).explain()
      .then(explainResult => {
        console.log("Query Execution Plan:", explainResult);

        // Executing the original query
      })
      .catch(err => {
        console.log("Error explaining query:", err);
      });
  })
  .catch(err => {
    console.log("Error connecting to database:", err);
  });
```

O/P:

```
node explain1.js
Query Execution Plan: {
  explainVersion: '1',
  queryPlanner: {
    namespace: 'mydb9.fac',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: 'C8EDFE04',
```

```
    planCacheKey: 'C8EDFE04',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'SORT',
      sortPattern: [Object],
      memLimit: 104857600,
      limitAmount: 2,
      type: 'simple',
      inputStage: [Object]
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 2,
    executionTimeMillis: 13,
    totalKeysExamined: 0,
    totalDocsExamined: 5,
    executionStages: {
      stage: 'SORT',
      nReturned: 2,
      executionTimeMillisEstimate: 10,
      works: 10,
      advanced: 2,
      needTime: 7,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
```

```
    isEOF: 1,
    sortPattern: [Object],
    memLimit: 104857600,
    limitAmount: 2,
    type: 'simple',
    totalDataSizeSorted: 384,
    usedDisk: false,
    spills: 0,
    inputStage: [Object]
  },
  allPlansExecution: []
},
command: {
  find: 'fac',
  filter: {},
  sort: { name: 1 },
  limit: 2,
  '$db': 'mydb9'
},
serverInfo: {
  host: 'PRANEETH',
  port: 27017,
  version: '6.0.5',
  gitVersion: 'c9a99c120371d4d4c52cbb15dac34a36ce8d3b1d'
},
serverParameters: {
  internalQueryFacetBufferSizeBytes: 104857600,
  internalQueryFacetMaxOutputDocSizeBytes: 104857600,
  internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
  internalDocumentSourceGroupMaxMemoryBytes: 104857600,
```

```
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1
}
```

Understanding the Db Object:

Table 13.4 **Methods on the Db object**

Method	Description
`open(callback)`	Connects to the database. The `callback` function is executed once the connection has been made. The first parameter to the `callback` is an error if one occurs, and the second is the `Db` object. For example: `function(error, db){}`
`db(dbName)`	Creates a new instance of the `Db` object. The connections sockets are shared with the original.
`close([forceClose], callback)`	Closes the connection to the database. The `forceClose` parameter is a Boolean that, when `true`, forces closure of the sockets. The `callback` function is executed when the database is closed and accepts an `error` object and a `results` object: `function(error, results){}`
`admin()`	Returns an instance of an `Admin` object for MongoDB. (See Table 13.5.)

`collectionInfo ([name], callback)`	Retrieves a `Cursor` object that points to collection information for the database. If `name` is specified, then only that collection is returned in the cursor. The `callback` function accepts `error` and `cursor` parameters. `function(err, cursor){}`
`collectionNames (callback)`	Returns a list of the collection names for this database. The `callback` function accepts an `error` and `names` parameters, where `names` is an array of collection names: `function(err, names){}`
`collection(name, [options], callback)`	Retrieves information about a collection and creates an instance of a `Collection` object. The `options` parameter is an object that has properties that define the access to the collection. The `callback` function accepts an `error` and `Collection` object as parameters: `function(err, collection){}`
`createCollection (collectionName, callback )`	Creates a new collection in the database. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`dropCollection (collectionName, callback )`	Deletes the collection specified by collection name from the database. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`renameCollection (oldName, newName, callback)`	Renames a collection in the database. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`dropDatabase(dbName, callback)`	Deletes this database from MongoDB. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`

`collections(callback)`	Retrieves information about all collections in this database and creates an instance of a `Collection` object for each of them. The `callback` function accepts an `error` and `collections` as parameters, where `collections` is an array of `Collection` objects: `function(err, collections){}`
`logout(callback)`	Logs the user out from the database. The `callback` accepts an `error` object and a `results` object: `function(error, results){}`
`authenticate(username, password, callback)`	Authenticates as a user to this database. You can use this to switch between users while accessing the database. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`addUser(username, password, callback)`	Adds a user to this database. The currently authenticated user needs user administration rights to add the user. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`removeUser(username, callback)`	Removes a user from the database. The callback function accepts an `error` object and a `results` object: `function(error, results){}`

Admin Object:

Table 13.5 **Methods on the `Admin` object**

Method	Description
`serverStatus(callback)`	Retrieves status information from the MongoDB server. The `callback` function accepts an `error` object and a `status` object: `function(error, status){}`
`ping(callback)`	Pings the MongoDB server. This is useful since you can use your Node.js apps to monitor the server connection to MongoDB. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`listDatabases (callback)`	Retrieves a list of databases from the server. The callback function accepts an `error` object and a `results` object: `function(error, results){}`
`authenticate(username, password, callback)`	Same as for `Db` in Table 13.4 except for the `admin` database.
`logout(callback)`	Same as for `Db` in Table 13.4 except for the `admin` database.
`addUser(username, password, [options], callback)`	Same as for `Db` in Table 13.4 except for the `admin` database.
`removeUser(username, callback)`	Same as for `Db` in Table 13.4 except for the `admin` database.

Cursor object:

Table 13.7 **Basic methods on the `Cursor` Object**

Method	Description
`each(callback)`	Iterates on each item in the `Cursor` from the current cursor index and calls the `callback` each time. This allows you to perform the `callback` function on each item represented by the cursor. The `callback` function accepts an `error` object and the `item` object:

```
function(err, item){}
```

`toArray(callback)`	Iterates through the items in the `Cursor` from the current index forward and returns an array of objects to the callback function. The `callback` function accepts an `error` object and the `items` array: `function(err, items){}`
`nextObject(callback)`	Returns the next object in the `Cursor` to the `callback` function and increments the index. The `callback` function accepts an error object and the item object: `function(err, item){}`
`rewind()`	Resets the `Cursor` to the initial state. This is useful if you encounter an error and need to reset the cursor and begin processing again.
`count(callback)`	Determines the number of items represented by the cursor. The `callback` function accepts an `error` object and the `count` value: `function(err, count){}`
`sort(keyOrList, direction, callback)`	Sorts the items represented by the `Cursor`. The `keyOrList` parameter is a `String` or `Array` of field keys that specifies the field(s) to sort on. The `direction` parameter is a number, where `1` is ascending and `-1` is descending. The `callback` function accepts an `error` as the first parameter and the `sortedCursor` object as the second: `function(err, sortedCursor){}`
`close(callback)`	Closes the `Cursor`, which frees up memory on the client and on the MongoDB server.
`isClosed()`	Returns `true` if the `Cursor` has been closed; otherwise, returns `false`.

Understanding Database Change Options:

Table 14.1 Options that can be specified in the `options` parameter of database changing requests to define behavior

Option	Description
`w`	Specifies the write concern level for database connections. See Table 13.1 for the available values.
`wtimeout`	Specifies the amount of time in milliseconds to wait for the write concern to finish. This value is added to the normal connection timeout value.
`fsync`	A Boolean that, when `true`, indicates that write requests wait for `fsync` to finish before returning.
`journal`	A Boolean that, when `true`, indicates that write requests wait for the journal sync to complete before returning.
`serializeFunctions`	A Boolean that, when `true`, indicates that functions attached to objects will be serialized when stored in the document.
`forceServerObjectId`	A Boolean that, when `true`, indicates that any object ID (`_id`) value set by the client is overridden by the server during insert.
`checkKeys`	A Boolean that, when `true`, causes the document keys to be checked when being inserted into the database. The default is `true`. (Warning: Setting this to `false` can open MongoDB up for injection attacks.)
`upsert`	A Boolean that, when `true`, specifies that if no documents match the update request, a new document is created.
`multi`	A Boolean that, when `true`, specifies that if multiple documents match the query in an update request, all documents are updated. When `false`, only the first document found is updated.
`new`	A Boolean that, when `true`, specifies that the newly modified object is returned by the `findAndModify()` method instead of the pre-modified version. The default is `false`.

Table 14.2 Operators that can be specified in the `update` object when performing update operations

Operator	Description
`$inc`	Increments the value of the field by the specified amount. Operation format: `field:inc_value`
`$rename`	Renames a field. Operation format: `field:new_name`
`$setOnInsert`	Sets the value of a field when a new document is created in the update operation. Operation format: `field:value`
`$set`	Sets the value of a field in an existing document. Operation format: `field:new_value`
`$unset`	Removes the specified field from an existing document. Operation format: `field:""`
`$`	Acts as a placeholder to update the first element that matches the query condition in an update.
`$addToSet`	Adds elements to an existing array only if they do not already exist in the set. Operation format: `array_field:new_value`

`$pop`	Removes the first or last item of an array. If the `pop_value` is `-1`, the first element is removed. If the `pop_value` is `1`, the last element is removed. Operation format: `array_field:pop_value`
`$pullAll`	Removes multiple values from an array. The values are passed in as an array to the field name. Operation format: `array_field:[value1, value2, ...]`
`$pull`	Removes items from an array that match a query statement. The query statement is a basic query object with field names and values to match. Operation format: `array_field:[<query>]`

`$push`	Adds an item to an array. Simple array format: `array_field:new_value` Object array format: `array_field:{field:value}`
`$each`	Modifies the `$push` and `$addToSet` operators to append multiple items for array updates. Operation format: `array_field:{$each:[value1, ...]}`
`$slice`	Modifies the `$push` operator to limit the size of updated arrays.
`$sort`	Modifies the `$push` operator to reorder documents stored in an array. Operation format: `array_field:{$slice:<num>}`

`$bit`	Performs bitwise AND and OR updates of integer values.
	
	Operation format:
	
	`integer_field:{and:<integer>}`
	`integer_field:{or:<integer>}`

Understanding Query Objects:

Table 15.1 `query` object operators that define the result set returned by MongoDB requests

Operator	Description
`$eq`	Matches documents with fields that have a value equal to the value specified.
`$gt`	Matches values that are greater than the value specified in the query. For example: `{size:{$gt:5}}`
`$gte`	Matches values that are equal to or greater than the value specified in the query. For example: `{size:{$gte:5}}`
`$in`	Matches any of the values that exist in an array specified in the query. For example: `{name:{$in:['item1', 'item2']}}`
`$lt`	Matches values that are less than the value specified in the query. For example: `{size:{$lt:5}}`
`$lte`	Matches values that are less than or equal to the value specified in the query. For example: `{size:{$lte:5}}`
`$ne`	Matches all values that are not equal to the value specified in the query. For example: `{name:{$ne:"badName"}}`
`$nin`	Matches values that do not exist in an array specified to the query. For example: `{name:{$nin:['item1', 'item2']}}`
`$or`	Joins query clauses with a logical OR; returns all documents that match the conditions of either clause. For example: `{$or:[{size:{$lt:5}}, {size:{$gt:10}}]}`

`$and`	Joins query clauses with a logical AND; returns all documents that match the conditions of both clauses. For example: `{$and:[{size:{$gt:5}}, {size:{$lt:10}}]}`
`$not`	Inverts the effect of a query expression and returns documents that do not match the query expression. For example: `{$not: {size:{$lt:5}}}`
`$nor`	Joins query clauses with a logical NOR; returns all documents that fail to match both clauses. For example: `{$nor:{size: {$lt:5}},{name:"myName"}}`
`$exists`	Matches documents that have the specified field. For example: `{specialField:{$exists:true}}`
`$type`	Selects documents if a field is of the specified BSON type number. Table 11.1 lists the different BSON type numbers. For example: `{specialField: {$type:<BSONtype>}}`
`$mod`	Performs a modulo operation on the value of a field and selects documents with a specified result. The value for the modulo operation is specified as an array with the first number being the number to divide by and the second being the remainder. For example: `{number:{$mod:[2,0]}}`
`$regex`	Selects documents where values match a specified regular expression. For example: `{myString: {$regex:'some.*exp'}}`
`$all`	Matches arrays that contain all elements specified in the query. For example: `{myArr:{$all: ['one','two','three']}}`
`$elemMatch`	Selects documents if an element in the array of subdocuments has fields that match all the specified `$elemMatch` conditions. For example: `{myArr:{$elemMatch:{value: {$gt:5},size:{$lt:3}}}}`
`$size`	Selects documents if the array field is a specified size. For example: `{myArr:{$size:5}}`

Understanding the Collection Object:

Table 13.6 **Basic methods on the Collection object**

Method	Description
`insert(docs, [callback])`	Inserts one or more documents into the collection. The `docs` parameter is an object describing the documents. The `callback` function must be included when using a write concern. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`remove([query], [options], [callback])`	Deletes documents from the collection. The `query` is an object used to identify the documents to remove. If no `query` is supplied, all documents are deleted. If a `query` object is supplied, the documents that match the query are deleted. The `options` allow you to specify the write concern using `w`, `wtimeout`, `upsert`, and `options` when modifying documents. The `callback` function must be included when using a write concern. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`rename(newName, callback)`	Renames the collection to `newName`. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`

`save([doc], [options], [callback])`	Saves the document specified in the `doc` parameter to the database. This is useful if you are making ad-hoc changes to objects and then needing to save them, but is not as efficient as `update()` or `findAndModify`. The `options` allow you to specify the write concern using `w`, `wtimeout`, `upsert`, and `new` options when modifying documents. The `callback` function must be included when using a write concern. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`update(query, update, [options], [callback])`	Updates the documents that match the `query` in the database with the information specified in the `document` parameter. The `options` allow you to specify the write concern using `w`, `wtimeout`, `upsert`, and `new` options when modifying documents. The `callback` function must be included when using a write concern. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`find(query, [options], callback)`	Creates a `Cursor` object that points to a set of documents that match the query. The `options` parameter is an object that allows you to specify the limit, sort, and many more options when building the cursor on the server side. The `callback` function accepts an `error` as the first parameter and the `Cursor` object as the second: `function(error, cursor){}`
`findOne(query, [options], callback)`	Same as `find()` except that only the first document found is included in the `Cursor`.
`findAndModify(query, sort, update,`	Performs modifications on documents that match the `query` parameter. The `sort` parameter

`[options], callback)`	determines which objects are modified first. The `doc` parameter specifies the changes to make on the documents. The `options` allow you to specify the write concern using `w`, `wtimeout`, `upsert`, and `new` options when modifying documents. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`findAndRemove(query, sort, [options], callback)`	Removes documents that match the `query` parameter. The `sort` parameter determines which objects are modified first. The `options` allow you to specify the write concern using `w`, `wtimeout`, `upsert`, and new options when deleting documents. The `callback` function accepts an `error` object and a `results` object: `function(error, results){}`
`distinct(key, [query], callback)`	Creates a list of distinct values for a specific document `key` in the collection. If a `query` is specified, only those documents that match the query are included. The `callback` function accepts `error` and `values` parameters, where `values` is an array of distinct values for the specified `key`: `function(error, values){}`
`count([query], callback)`	Counts the number of documents in a collection. If a `query` parameter is used, only documents that match the query are included. The `callback` function accepts an `error` object and a `count` parameter, where `count` is the number of matching documents: `function(error, count){}`
`drop(callback)`	Drops the current collection. The `callback` function accepts an `error` object and a `results`

	object: `function(error, results){}`
`stats(callback)`	Retrieves the `stats` for the collection. The stats include the `count` of items, `size` on disk, `average` object size, and much more. The `callback` function accepts an `error` object and a `stats` object: `function(error, stats){}`

Understanding Query Options Objects

Table 15.2 Options that can be specified in the `options` object when querying documents

Option	Description
`limit`	Specifies the maximum number of documents to return.
`sort`	Specifies the sort order of documents as an array of `[field, <sort_order>]` elements where `sort order` is `1` for ascending and `-1` for descending. For example: `sort:[['name':1],['value':-1]]`
`fields`	Specifies an object whose fields match fields that should be included or excluded from the returned documents. A value of `1` means include, a value of `0` means exclude. You can only include or exclude, not both. For example: `fields:{name:1,value:1}`
`skip`	Specifies the number of documents from the query results to skip before returning a document. Typically used when paginating result sets.
`hint`	Forces the query to use specific indexes when building the result set. For example: `hint:{'_id':1}`
`explain`	Returns an explanation of what will happen when performing the query on the server instead of actually running the query. This is essential when trying to

	debug/optimize complex queries.
`snapshot`	{Boolean, default:false}; specifies whether to use a snapshot query.
`timeout`	A Boolean; when true, the cursor is allowed to timeout.
`maxScan`	Specifies the maximum number of documents to scan when performing a query before returning. This is useful if you have a collection that has millions of objects and you don't want queries to run on forever.
`comment`	Specifies a string that is printed out in the MongoDB logs. This can help when troubleshooting because you can identify queries more easily.
`readPreference`	Specifies whether you want to read from a primary or secondary replica or just the nearest MongoDB server in the replica set to perform the query.
`numberOfRetries`	Specifies the number of timeout retries to perform on the query before failing. The default is 5.
`partial`	A Boolean that, when `true`, indicates the cursor will return partial results when querying against data shared between sharded systems.

Q) Develop a simple application form to take data from users to store into MongoDB database and retrieve using Express.js

index.html:

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>Express Project</title>
</head>
<body>
    <form action="http://localhost:3000/submit-data" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name"><br>

        <label for="branch">Branch:</label>
        <input type="text" id="branch" name="branch"><br>

        <label for="sal">Salary:</label>
        <input type="text" id="sal" name="sal"><br>
```

```
        <label for="domain">Domain:</label>
        <input type="text" id="domain" name="domain" placeholder="If multiple
values separated by commas"><br>

        <input type="submit" value="Submit">

    </form>
</body>
</html>
```

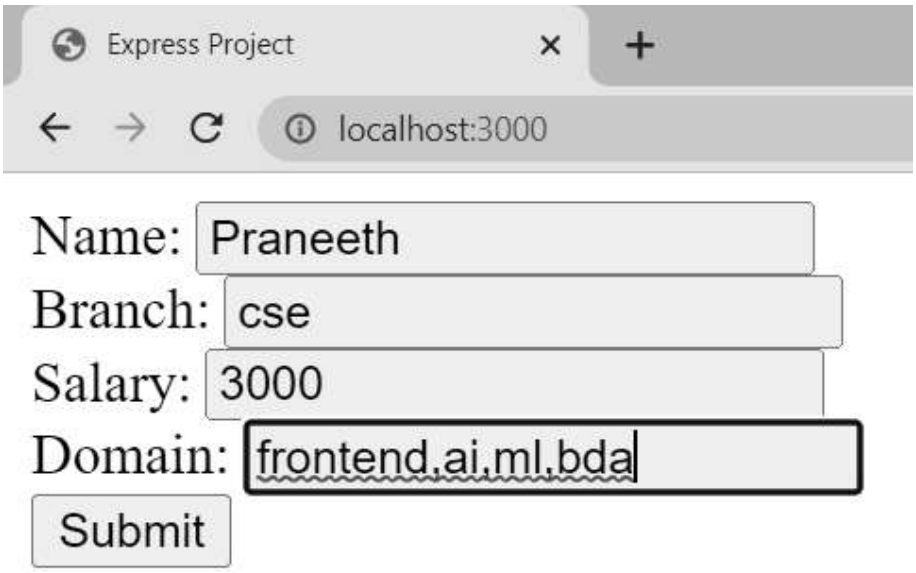
index.js

```
const express = require('express');
const app = express();

const path = require('path');
const bodyParser = require('body-parser');
const { MongoClient } = require('mongodb');
const uri = 'mongodb://127.0.0.1:27017';

app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', function (req, res) {
  const filePath = path.join(__dirname, 'index.html');
  res.sendFile(filePath);
});

app.post('/submit-data', function (req, res) {
  const name = req.body.name;
  const branch = req.body.branch;
  const salary = req.body.sal;
  const domains = req.body.domain.split(',').map(domain => domain.trim());

  MongoClient.connect(uri)
    .then(client => {
      console.log('Connected to database!');
      const db = client.db('mydb999');
      const collection = db.collection('fac');
      const date = Date();
      return collection.insertOne({ name, branch, salary, domains, date });
    })
    .then(result => {
      console.log('Data submitted successfully');
      res.send('Data submitted successfully');
    })
    .catch(err => {
      console.log('Error performing operation:', err);
      res.status(500).send('Error submitting data');
```

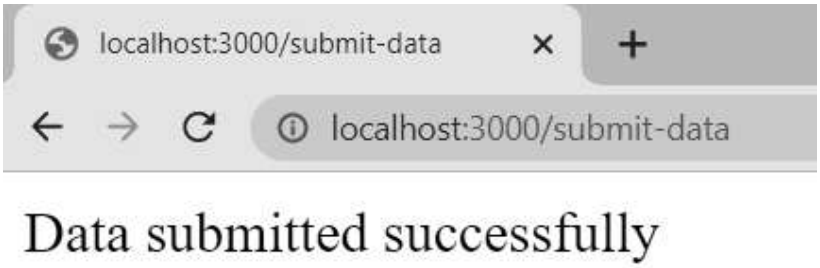
```
    });
});

app.get('/get-data', function (req, res) {
  MongoClient.connect(uri)
    .then(client => {
      console.log('Connected to database!');
      const db = client.db('mydb999');
      const collection = db.collection('fac');
      return collection.find().toArray();
    })
    .then(data => {
      console.log('Retrieved data:', data);
      const tableRows = data.map(record => {
        const { name, branch, salary, domains, date } = record;

return `
          <tr>
            <td>${name}</td>
            <td>${branch}</td>
            <td>${salary}</td>
            <td>${domains.join(', ')}</td>
            <td>${date}</td>
          </tr>
        `;
      });
      const table = `
        <table>
          <thead>
            <tr>
              <th>Name</th>
              <th>Branch</th>
              <th>Salary</th>
              <th>Domains</th>
              <th>Date</th>
            </tr>
          </thead>
          <tbody>
            ${tableRows.join('')}
          </tbody>
        </table>
      `;
      res.send(table);
    })
    .catch(err => {
      console.log('Error performing operation:', err);
      res.status(500).send('Error retrieving data');
    });
```

```
});

const server = app.listen(3000, function () {
  console.log('Node server is running on 3000..');
});
```

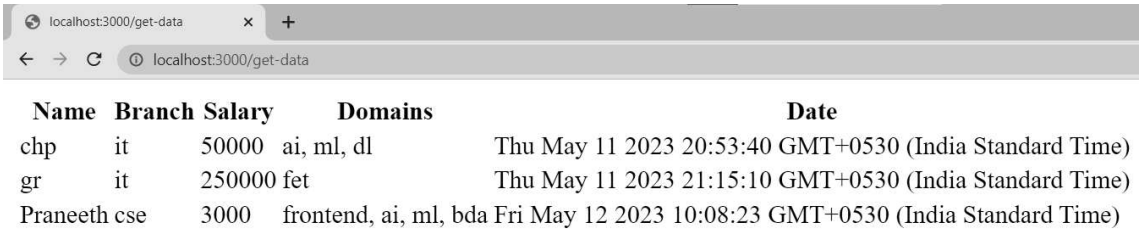
O/P:



(a) User entering data in the form



(b) User successfully stored data into the MongoDB



Name	Branch	Salary	Domains	Date
chp	it	50000	ai, ml, dl	Thu May 11 2023 20:53:40 GMT+0530 (India Standard Time)
gr	it	250000	fet	Thu May 11 2023 21:15:10 GMT+0530 (India Standard Time)
Praneeth	cse	3000	frontend, ai, ml, bda	Fri May 12 2023 10:08:23 GMT+0530 (India Standard Time)

(c) User retrieved data from the MongoDB