Lecture: Introduction to MongoDB

- What is MongoDB?
- Key Features of MongoDB
- Installation and Setup
- Basic CRUD Operations
- Data Modeling in MongoDB
- Indexing and Query Optimization
- Aggregation Framework
- Replica Sets and Sharding
- MongoDB Atlas: Managed Cloud Database
- Best Practices and Tips

Introduction to MongoDB

- MongoDB is a NoSQL database management system.
- Developed by MongoDB Inc.
- Designed for flexibility, scalability, and performance.
- Uses a document-oriented data model (documents are objects)
- Ideal for applications with rapidly changing schema and large amounts of data.

Key Features of MongoDB

- Flexible Schema: MongoDB stores data in flexible, JSON-like documents.
- Scalability: Horizontal scalability with automatic sharding.
- High Performance: Supports indexes, aggregations, and native replication.
- Rich Query Language: Supports rich queries and dynamic updates.
- High Availability: Replication with automatic failover.

Flexible Schema

- Every row in a collection does not have to be the same
- Documents (objects) may contain arrays and other documents
- Collection can easily represent a hierarchy of data
- This flexibility comes at a price in performance

Installation and Setup

- Download MongoDB Community Edition
- Install MongoDB
- Start MongoDB Server: After installation, start the MongoDB server using the appropriate command for your OS.
- Verify Installation: Open a command prompt and type ‘mongo –version‘ to verify that MongoDB is installed correctly.
- Run mongosh to connect to the local database by default

Collection: Definition

Collection:

- In MongoDB, a collection is a group of documents.
- Analogous to a table in relational databases.
- Collections do not enforce a schema.
- Documents within a collection can have different fields and structure.

Basic CRUD Operations

- CRUD stands for Create, Read, Update, and Delete.
- These operations are fundamental for interacting with data in MongoDB.
- MongoDB provides simple and powerful methods for performing CRUD operations.
- Examples of CRUD operations include:
  - Inserting documents into a collection (Create).
  - Retrieving documents from a collection (Read).
  - Updating existing documents in a collection (Update).
  - Removing documents from a collection (Delete).

Basic CRUD Operations: Example

Create (Insert):

- To insert a document into a collection named "users":
- `db.users.insertOne({name: "John", age: 30})`

Read (Retrieve):

- To retrieve all documents from the "users" collection:
- `db.users.find()`

Data Modeling in MongoDB: Overview

- Document Structure and Schema Design
- Embedded Documents vs. References
- One-to-One Relationships
- One-to-Many Relationships
- Many-to-Many Relationships
- Denormalization and Performance
- Indexing Strategies
- Case Studies
- Best Practices

Introduction: Data Modeling

- Process of designing the structure and organization of data within MongoDB databases.
- Determine how data will be stored, accessed, and manipulated to meet the requirements of the application.
- Effective data modeling is crucial for optimizing performance, ensuring data integrity, and facilitating scalability.
- This presentation will cover various aspects of data modeling in MongoDB, including document structure, relationships, denormalization, indexing strategies, and best practices.

MongoDB Primitive Data Types

- String
- Integer
- Double
- Decimal
- BigNumber
- BigDecimal
- Boolean
- Date
- Null

MongoDB Compound Data Types

- Object
- Array
- ObjectId
- Binary Data
- Regular Expression

Examples: Scalar Data Types

```
{
    "string": "Hello, MongoDB!",
    "integer": 42,
    "double": 3.14,
    "boolean": true,
    "date": ISODate("2022-04-05T12:00:00Z"),
    "null_value": null
}
```

Example: Integer Values

MongoDB integers are 32-bit signed

```
{
    "small_integer": 42,
    "large_integer": 2147483647,
    "negative_integer": -123
}
```

Range: -2,147,483,648 to 2,147,483,647

Example: Double Values and Range

```
{
    "small_double": 3.14,
    "large_double": 1.7976931348623157e+308,
    "negative_double": -123.456
}
```

Range: $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$

Example: Date Values

Dates are internally stored as a 64-bit integer

```
{
    "current_date": ISODate(),
    "date_with_time": ISODate("2022-04-05T12:00:00Z"),
    "date_only": ISODate("2022-04-05"),
    "date_with_timezone": ISODate("2022-04-05T00
        :00:00-04:00")
}
```

Big Integer Types

MongoDB supports two big integer types:

- **NumberLong**: A 64-bit signed integer type, capable of storing integers larger than the 32-bit signed integer range.
- **NumberDecimal**: A decimal128 type, capable of representing high-precision decimal numbers.

Usage:

- Use `NumberLong` to store integers larger than 32 bits, such as IDs, counters, or large numerical values.
- Use `NumberDecimal` for high-precision decimal calculations or when accuracy is critical.

Example: NumberLong

NumberLong is a 64-bit integer

```
{
    "user_id": NumberLong("12345678901234567 89"),
    "large_value": NumberLong("9223372036854775807"),
}
```

Document Structure and Schema Design

- Document Structure:
  - MongoDB stores data in flexible JSON-like documents called BSON.
  - Documents represent records in a collection and can vary in structure.
  - Fields within documents can be of any data type, including arrays and nested documents.
- Schema Design:
  - Designing an appropriate schema is crucial for efficient data storage and retrieval.
  - Considerations include data access patterns, query performance, and scalability.
  - Schema design decisions impact read and write operations, as well as database performance.

Example: Simple BSON Document

```
{
    "name": "John Doe",
    "age": 30,
    "email": "john@example.com"
}
```

Example: Nested BSON Document

```
{
    "name": "John Doe",
    "age": 30,
    "address": {
    "city": "New York",
    "zip": "10001"
    }
}
```

Example: Array in BSON Document

```
{
    "name": "John Doe",
    "age": 30,
    "hobbies": ["reading", "hiking", "photography"]
}
```

Object Type

- Represents a nested document within a MongoDB document.
- Allows for the grouping of related fields and values.
- Can be used to model complex data structures.
- Supports nested objects within objects (subdocuments).

Example: Object Type

```
{
    "student": {
    "name": "Alice",
    "age": 25,
    "grades": [90, 85, 95],
    "address": {
        "city": "New York",
        "zipcode": "10001"
    }
    }
}
```

Collection Naming Conventions

- Use plural nouns
- Be descriptive
- Use camelCase or underscores
- Avoid reserved characters $, ., 0
- Keep it short and concise
- Maintain consistency

Collection Naming: Legal and Illegal Examples

- users
- blog_posts
- customerFeedback
- userAccounts
- $users
- my.collection
- user.name
- text
  0me
- collection1
- 1collection

Insert Examples

```
1  // Create the "users" collection
2  db.createCollection("users")
3
4  // Insert a single document using insertOne()
5  db.users.insertOne({
6      username: "john_doe",
7      email: "john@example.com",
8      age: 30
9  })
```

Example: insertMany

```
1  // Insert multiple documents using insertMany()
2  db.users.insertMany([
3  {
4      username: "jane_doe",
5      email: "jane@example.com",
6      age: 25
7  },
8  {
9      username: "bob_smith",
0      email: "bob@example.com",
1      age: 35
2  }
3  ])
```

Create Collection with Validation

```
1  // Create the foodSubstances collection with validation
2  db.createCollection("foodSubstances", {
3      validator: {
4      $jsonSchema: {
5          bsonType: "object",
6          required: ["name", "type", "altname", "unit"],
7          properties: {
8          name: { bsonType: "string" },
9          type: { enum: ["vitamin", "mineral", "poison", "
               allergen", "drug"] },
0          altname: { bsonType: ["string", "null"] },
1          unit: { enum: ["mg", "mcg", "IU", "g", "ug"] }
2          }
3      }
4      }
5  });
```

Inserting Records with Validation

```
1  // Inserting a correct record (Vitamin C)
2  db.foodSubstances.insertOne
3  ({name: "vC", type: "vitamin", altname: "Ascorbic Acid",
       unit: "mg"});
4
5  // Inserting an incorrect record (violates constraint)
6  db.foodSubstances.insertOne
7  ({name: "mFe", type: "foo", altname: "Iron", unit: "mg"
      });
8
9  db.foodSubstances.insertOne
0  ({name: "vC", type: "vitamin", altname: "Ascorbic Acid",
       unit: "mg"});
1  // error, vitamin C already exists (no constraint
      prevents this)
```

MongoDB Comparison Operators

- $eq - Matches values that are equal to a specified value.
- $ne - Matches values that are not equal to a specified value.
- $gt - Matches values that are greater than a specified value.
- $gte - Matches values that are greater than or equal to a specified value.
- $lt - Matches values that are less than a specified value.
- $lte - Matches values that are less than or equal to a specified value.
- $in - Matches any of the values specified in an array.
- $nin - Matches none of the values specified in an array.

Examples of Comparison Operations

```
1 // Find documents where the "age" field is equal to 30
2 db.collection.find({ age: { $eq: 30 } })
3
4 // Find documents where the "price" field is greater
      than $100
5 db.collection.find({ price: { $gt: 100 } })
6
7 // Find documents where the "quantity" field is less
      than or equal to 10
8 db.collection.find({ quantity: { $lte: 10 } })
```

Examples of Array Operations

```
1  // Add a grade of 85 to Melanie's grades array
2  db.students.updateOne( { _id: 1 }, { $push: { grades: 85
       } })
3
4  // Add a grade of 90 if it doesn't already exist, note
       creates hashset
5  db.students.updateOne({ _id: 1 }, { $addToSet: { grades:
       90 } })
6
7  // Remove the last grade from Melanie's grades array
8  db.students.updateOne( { _id: 1 }, { $pop: { grades: 1 }
       } )
9
0  // Remove all grades less than 80 from Melanie's grades
       array
1  db.students.updateOne( { _id: 1 }, { $pull: { grades: {
       $lt: 80 } } } )
2
3  // Remove grades 85 and 90 from Melanie's grades array
4  db.students.updateOne( { _id: 1 }, { $pullAll: { grades:
       [85, 90] } } )
-
```

Examples of Regex Queries in MongoDB

```
1  // Finding a student whose name starts with "E"
2  db.students.find({ name: /^E/ })
3
4  // Finding students whose name contains "a", "b", or "c"
5  db.students.find({ name: /[abc]/ })
6
7  // Finding students whose name ends with "y"
8  db.students.find({ name: /y$/ })
9
0  // Finding students whose name contains "mel" case
       insensitively
1  db.students.find({ name: /mel/i })
```

Storing JavaScript in a Database

- **Custom Business Logic**: Store validation rules, workflow logic, or calculation formulas as JavaScript functions in the database.
- **Dynamic Scripting**: Dynamically generate and execute scripts based on user input or configuration for flexible application behavior.
- **Stored Procedures**: Define complex database operations as JavaScript functions for server-side execution.
- **Content Management Systems (CMS)**: Store templates, themes, or dynamic content logic as JavaScript code for easy customization.
- **User-defined Functions**: Allow users to define custom functions or scripts, stored in the database, for extended application functionality.

Example Business Logic

```
1 // Example business logic: Calculate total price with
      discount
2 function calculateTotalPrice(price, discount) {
3     return price - (price * discount);
4 }
```

Indexing and Query Optimization

- BTree is a popular indexing type for databases
- BTrees allow sorting data
- Hashing is faster, but only allows equal/not equal
- Hashing can only be used where ordering is not relevant
- Hashing floating point is weird because of roundoff error

BTree

Indexing and Query Optimization

Indexing:

- MongoDB supports various types of indexes to optimize query performance.
- Common index types include single-field, compound, multi-key, text, and geospatial indexes.
- Indexes help MongoDB efficiently locate and retrieve documents based on query conditions.

Query Optimization:

- Effective query optimization is essential for improving database performance.
- Techniques include utilizing indexes, minimizing the number of documents scanned, and optimizing query execution plans.
- Understanding query patterns and access patterns is crucial for identifying optimization opportunities.

Index Options and Validation

- Ascending/Descending Order:
  - Specifies the sorting order of the index.
  - 1 indicates ascending order, -1 indicates descending order.
- Unique:
  - Ensures that indexed field(s) contain unique values.
  - MongoDB rejects operations that would result in duplicate values.
- Not Null Constraint:
  - Enforced by the validator, not the index.
  - Validators specify rules for document validation, including required fields.

Creating Unique Index on Short Name

```
1 // Create unique index on the shortName field
2 db.foodSubstances.createIndex({ name: 1 }, { unique:
      true });
```

Example: Index Substances by Type

To efficiently query substances by their type, create an index on the `type` field in the `foodSubstances` collection:

```
1 db.foodSubstances.createIndex({ "type": 1 });
2 db.foodSubstances.createIndex({ "type": "hashed"})
```

Example query to retrieve all substances of a specific type:

```
1 db.foodSubstances.find({ "type": "vitamin" });
```

Updating Food Substances Collection

```
1 db.foodSubstances.updateOne(
2     { name: "vC" },
3     { $set: { "RDA.child<12": "25-75" } }
4 );
5 db.foodSubstances.updateMany(
6     { type: "vitamin" }, // Filter: Update all documents
          where type is "vitamin"
7     { $set: { "RDA.child<12": "30-70" } } // syntax
        probably wrong. I should not have used < in
        string
8 );
```

Replica Sets

- High availability and fault tolerance
- Primary and secondary nodes
- Automatic failover
- Read scalability
- Ensures data durability and minimizes downtime

Updating Food Substances Collection

```
# Start MongoDB instances on different servers
mongod --port 27017 --dbpath /data/db1 --replSet rs0
mongod --port 27018 --dbpath /data/db2 --replSet rs0
mongod --port 27019 --dbpath /data/db3 --replSet rs0

# Connect to one of the MongoDB instances
mongo --port 27017

# Initialize the replica set configuration
rs.initiate({_id: "rs0", members: [{_id: 0, host: "
    localhost:27017"}]})

# Add the other MongoDB instances to the replica set
rs.add("localhost:27018")
rs.add("localhost:27019")
```

Sharding Overview

- Horizontal scaling technique
- Distributes data across shards
- A shard is a subset of data
- Each shard can run on a separate MongoDB instance
- Net performance increases by parallelizing MongoDB
- Data distribution based on shard key
- Enables scalability

Horizontal Sharding

- Splitting data horizontally involves dividing the entire dataset into smaller subsets or partitions.
- Each partition is stored on a separate server.
- This approach is useful for distributing large datasets across multiple servers to improve performance and scalability.

Horizontal Sharding Example

- Sharded Collection: "users"
- Shard Key: "country"
- Servers:
  - S1: Stores users from USA, Canada, Mexico
  - S2: Stores users from UK, Germany, France
  - S3: Stores users from Japan, China, India

Vertical Sharding

- Splitting data vertically involves dividing the columns of a table across different servers.
- Each server holds a subset of columns for all rows in the table.
- This approach is useful for separating frequently accessed columns or those requiring more resources.

Conclusion

- MongoDB is a powerful and flexible
- Distributed database, easy to use
- Slower than mysql, easy parallelism might make up for that