CipherSchools

MongoDB Class 3

Schema Design and Data Modeling

MongoDB Schema Validation

Schema Validation in MongoDB allows you to define rules or constraints for documents in a collection using JSON Schema format.
This helps enforce data quality, structure, and consistency—just like a schema in SQL databases.

Why Use Schema Validation?

- Avoid invalid or incomplete data
- Enforce required fields, types, and value formats
- Control nested structures (arrays, objects)
- Supports automated tooling, e.g., with Compass or IDEs

How to Use Schema Validation:

a. While creating collections:

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age", "email"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
      }
    }
```

```
})
```

b. Modify already existing schema:

```
db.runCommand( { collMod: "students",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name", “age”, "email", “password”],
      properties: {
       name: {
              bsonType: "string",
              description: "must be a string and is required"
       },
       password: {
            bsonType: "string",
            minLength: 12,
            description: "must be a string of at least 12 characters, and is
required"
         }
       }
    }
  },
  validationLevel: "strict",  // strict or moderate
  validationAction: "error"   // error or warn
} )
```

Validation Options	Options Description
validationLevel	"strict" (default): Validate all inserts & updates "moderate": only validate if document already has schema
validationAction	"error" (default): Reject invalid docs "warn": Allow insert/update but log a warning

Important Notes

- Schema validation only blocks bad inserts/updates, it doesn't fix existing bad data.
- Use tools like MongoDB Compass or Atlas UI for easier schema rule creation.
- Combine with indexing for optimized querying and integrity.

Embedding vs Referencing

Embedding: Store related data in the same document.
Referencing: Store related data in different documents and link them using IDs.

Embedding

Definition: Put related data inside the parent document.
Example: Student with subjects

```
{
  "_id": "student123",
  "name": "Riya",
  "email": "riya@example.com",
  "subjects": [
    { "name": "Math", "marks": 92 },
    { "name": "Science", "marks": 89 }
  ]
}
```

Pros:

- Fast reads (all data in one place).
- No joins needed.
- Easy to fetch nested data.

Cons:

- Can grow too large (MongoDB doc limit: 16MB).
- Harder to update sub-documents individually.
- Duplicates data across documents if reused.

Referencing

Definition: Store related data in separate collections and link using _id.
Example: Student and subjects in separate collections

→ students collection

```
{
  "_id": "student123",
  "name": "Riya",
  "email": "riya@example.com",
  "subjectIds": ["sub1", "sub2"]
}
```

→subjects collection

```
{ "_id": "sub1", "name": "Math", "marks": 92 }
{ "_id": "sub2", "name": "Science", "marks": 89 }
```

Pros:

- Reuse data (no duplication).
- Smaller document size.
- Flexible structure.

Cons:

- Requires manual joins using $lookup.
- More complex writes/reads.
- Slower performance for deeply nested data.

Best Practices

- Use embedding when data fits together naturally and doesn’t grow endlessly.
- Use referencing when you need flexibility or prevent duplication.
- Always index foreign fields in referencing.

ONE-TO-ONE, ONE-TO-MANY, MANY-TO-MANY

RELATIONSHIPS IN MONGODB

MongoDB is a NoSQL document-based database. Relationships can be handled using:

- Embedding - data inside the same document.
- Referencing - linking documents via _id.

ONE-TO-ONE RELATIONSHIP

One document is related to exactly one document in another collection.

Use When:

- Related data is accessed together.
- The size of both documents is small.

Options:

a. Embedded:

```
// User document with embedded profile
{
  _id: ObjectId("1"),
  name: "Alice",
  profile: {
    age: 22,
    gender: "Female"
  }
}
```

b. Referenced:

```
// User
{ _id: ObjectId("1"), name: "Alice", profileId: ObjectId("p1") }

// Profile
{ _id: ObjectId("p1"), age: 22, gender: "Female" }
```

ONE-TO-MANY RELATIONSHIP

One document is related to many documents.

Use When:
Parent data is frequently accessed, but child data is large or sparsely accessed.

Options:

a. Embedding (if child count is limited):

```
{
  _id: ObjectId("1"),
  name: "Blog Post",
  comments: [
    { user: "John", comment: "Great post!" },
    { user: "Jane", comment: "Nice one!" }
  ]
}
```

b. Referencing (if unbounded array):

```
// Blog
{ _id: ObjectId("1"), title: "Blog Post" }

// Comments
{ _id: ObjectId("c1"), postId: ObjectId("1"), comment: "Great!" }
{ _id: ObjectId("c2"), postId: ObjectId("1"), comment: "Nice!" }
```

MANY-TO-MANY RELATIONSHIP

Many documents relate to many others (e.g., students & courses).

Use When:
You want to track both sides of the relationship.

Example:

a. Using Referencing:

```
// Student
{
  _id: ObjectId("s1"),
  name: "Alice",
  courses: [ObjectId("c1"), ObjectId("c2")]
}

// Course
{
  _id: ObjectId("c1"),
  name: "Math",
  students: [ObjectId("s1"), ObjectId("s2")]
}
```

b. Through a Join Collection (best practice for many-to-many):

```
// Enrollment
{
  studentId: ObjectId("s1"),
  courseId: ObjectId("c1"),
  enrolledOn: ISODate("2023-01-01")
}
```

Embedding	Referencing
Fast read, slow write	Slow read (joins), fast write
Good for nested, static, limited data	Good for growing, shared, large data
Denormalized	Normalized

Normalization vs Denormalization

What is Normalization?

Definition:
Normalization is the process of organizing data in a database to minimize redundancy and dependency.

Objectives:

- Eliminate data duplication
- Ensure data integrity
- Simplify maintenance

Normal Forms (Popular Ones):

1. 1NF (First Normal Form):
   No repeating groups or arrays in a row
   - Atomic values only
   - Example: Remove multiple phone numbers from one column into a separate table

2. 2NF (Second Normal Form):
   - Must be in 1NF
   - All non-key attributes are fully functionally dependent on the entire primary key

3. 3NF (Third Normal Form):
   - Must be in 2NF
   - No transitive dependency (non-key → another non-key)

Example (Normalized):

Imagine a college database:
Students Table

student_id	name	department_id
1	Sriram	101
2	Puneet	102

Departments Table

department_id	department_name
101	CSE

102	ECE

- No redundancy in storing department names.
- Data is consistent: if department name changes, only update one row.

What is Denormalization?

Definition:
Denormalization is the process of adding redundant data to improve read performance at the cost of write/update complexity.

Objectives:

- Reduce complex joins
- Speed up read-heavy queries

Example (Denormalized):

student_id	name	department_name
1	Sriram	CSE
2	Puneet	ECE

- Department name is duplicated
- Easier and faster to query all student details with department name
- But if "CSE" is renamed, multiple rows must be updated

Normalization vs Denormalization

Feature	Normalization	Denormalization
Purpose	Reduce redundancy	Improve read performance
Data Duplication	Minimal	High
Query Complexity	High (requires joins)	Low (less joins)

Update Complexity	Low (single point of update)	High (need to update multiple rows)
Best For	Write-heavy systems	Read-heavy systems
Storage Efficiency	Better	More space required

Interview Point of View:

In a system that handles a lot of reads, denormalization helps speed up queries by avoiding joins. In a write-heavy system, normalization is preferred to maintain data integrity and reduce redundancy.

Creative Analogy to understand Normalization & Denormalization:

Imagine Normalization like organizing your wardrobe:

- Shirts go in one drawer
- Pants in another
- Socks in another
- Easy to keep clean, but time-consuming to get a full outfit (like a JOIN)

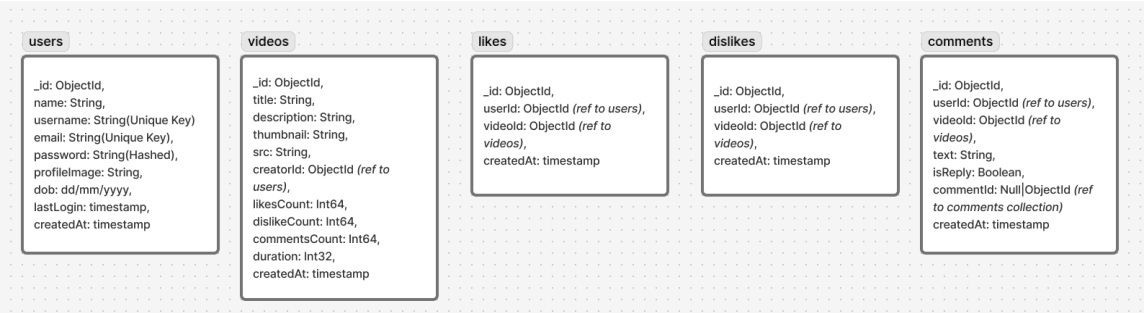
Denormalization is like keeping a full outfit ready-to-go together.

- Easy to grab and go (fast query)
- But if you change your shirt size, you need to update all outfits.

Designing Real Schemas (Practice)

Video Based Platform:

1. Requirements:
    a. Users can login signup with username or email and password.
    b. Users have the Profile section.
    c. Videos have the like and dislike actions on video.
    d. Users can comment on the video and can also reply to comments.
2. Schema Design:



Practice:

1. Social Media Platform

Requirements:

- Users can sign up with email/phone and password.
- Users can follow/unfollow other users.
- Users can make posts (text, image, video).
- Each post can be liked and commented on.
- Users can like/unlike comments.
- Comments can have threaded replies.

Goal: Design the schema to handle user profiles, follow relationships, posts, likes, comments, and replies.

2. E-commerce Platform

Requirements:

- Users can register and log in.
- Users can browse products with categories and tags.
- Products have variants (like size, color).
- Users can add products to cart and place orders.
- Orders include multiple products and store delivery status.
- Users can leave reviews and ratings on products.

Goal: Design schema for Users, Products, Variants, Categories, Cart, Orders, and Reviews.

3. Online Learning Platform

Requirements:

- Users can sign up as Students or Instructors.
- Instructors can create courses, which have multiple modules and lessons.
- Students can enroll in courses and track progress.
- Lessons can include videos and quizzes.
- Students can submit quiz answers and get feedback.

Goal: Design schema for Users (with roles), Courses, Enrollments, Lessons, Quizzes, Submissions, and Progress.

4. **Food Delivery Platform**

Requirements:

- Users can sign up and save multiple delivery addresses.
- Restaurants can register and add menus.
- Each menu has categories (e.g., starters, mains) and dishes.
- Users can place orders for items from a restaurant.
- Orders can have status updates (ordered, accepted, out for delivery, delivered).
- Users can leave reviews on restaurants.

Goal: Design schema for Users, Addresses, Restaurants, Menus, Dishes, Orders, and Reviews.

MongoDB Class 4

Indexing and Performance

What is Indexing?

Indexing in MongoDB is a mechanism for improving the performance of queries. Without indexes, MongoDB performs a collection scan, i.e. it scans every document to find the match - which is inefficient.

Indexes work like the index page of a book - you can find content faster without reading every page.

Why is Indexing Needed?

- Faster Read Operations (queries)
- Optimized Sorting and Range Queries
- Efficient Filtering (find, match, search)
- Scalability for Large Data
- Avoiding Full Collection Scans (bad for performance)

Example:
Imagine a university library with 10,000 books but no catalog. To find *Harry Potter and the Sorcerer's Stone*, you'd check every book manually. That's a collection scan.

Now, imagine the books are sorted alphabetically with a subject index. You go to "H", then "Harry", and quickly find your book. That's **indexing**.

How Indexing works?

Data Structure Used → B-Tree(Balanced Tree)
MongoDB uses a B-Tree or more precisely a B+Tree for its indexes.

Why B-Trees?

- Sorted: Helps in range queries and sorting.
- Balanced: Guarantees logarithmic search time (O(log n)).
- Efficient: Can hold many keys in a node—minimizes disk I/O.

A B-Tree (Balanced Tree) is a self-balancing tree data structure that maintains sorted data and allows searches, insertions, and deletions in logarithmic time.

Key Features:

- Each node can have more than two children (unlike a binary tree).
- Keeps keys in sorted order.
- Efficient for large datasets and disk-based storage systems.
- Maintains balance - height of tree is kept small.

Create, Get & Drop Syntax:

1. For creating a single field index
   db.students.createIndex({ email: 1 }) // 1 = ascending
2. For compound index (more than one field)
   db.students.createIndex({ batch: 1, score: -1 })
3. For get all Indexes of collection
   db.<collection>.getIndices()
4. For dropping an index from collection
   db.<collection>.dropIndex("index_name")

Types of Indexes

1. Single Field Index: *Index on one field*
   db.students.createIndex({ email: 1 })
   Example: Speeds up db.students.find({ email: "abc@gmail.com" })

2. Compound Index: *Index on multiple fields*
   db.students.createIndex({ username: 1, password: -1 })
   Example: Efficient for queries that filter/sort by both batch and score.

3. Multikey Index: *For arrays*
   db.students.createIndex({ subjects: 1 }) //subject:["Math", "Science", "History"]
   Example: db.students.find({ subjects: "Math" })

4. Text Index: *For full-text search*
   db.courses.createIndex({ title: "text", description: "text" })
   Example: Enables searching using db.courses.find({ $text: { $search: "Harry Potter" } })

5. TTL Index: *Automatically removes document after a time*

db.logs.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })
Example: After 3600 seconds (1 hour) from the createdAt value, the document is automatically deleted.

Explain Plans

Explain Plans are used to understand how a database engine (like MongoDB, MySQL, PostgreSQL, etc.) will execute a query. It helps developers and DBAs understand:

- How efficient a query is
- Whether indexes are used
- How much data will be scanned (full collection scan vs index scan)
- The execution strategy (or plan) of the database engine

Syntax & Example:

You use the .explain() method:

Example: db.users.find({ age: { $gt: 25 } }).explain("executionStats");

Output includes:

- stage (COLLSCAN or IXSCAN)
- nReturned (number of docs returned)
- executionTimeMillis (how long the query took)
- totalKeysExamined and totalDocsExamined

*If you see COLLSCAN, it means a collection scan = No index used!

Index Tradeoffs

1. Slower Write Performance:
    - Every insert, update, or delete must update the corresponding indexes.
    - In write-heavy systems, this becomes a performance bottleneck.

2. Increased Storage Usage:
    - Indexes take up disk space.
    - Compound and multi-key indexes especially can grow large.
3. Complex Maintenance:
    - As data changes, MongoDB must keep indexes updated.
    - This involves CPU and memory usage, particularly with large datasets.
4. Index Selection & Query Planner Conflicts:
    - MongoDB's query planner must choose the best index.
    - Sometimes, multiple indexes lead to suboptimal choices if not managed well.

Use Case Decision:

- If the system is read-heavy, use indexes generously.
- If write-heavy, minimize indexes or index only high-value queries.

Best Practices of Indexes:

- Create indexes on frequently queried fields.
- Avoid indexing fields that change frequently.
- Don't over-index — assess query patterns and usage.
- Use compound indexes carefully — order of fields matters.
- Regularly run .explain("executionStats") to analyze performance.