MongoDB Complete Guide: Zero to Hero

Table of Contents

1. Introduction to MongoDB
2. Installation & Setup
3. MongoDB Architecture
4. CRUD Operations
5. Data Modeling
6. Querying Deep Dive
7. Indexing
8. Aggregation Framework
9. Transactions
10. Replication
11. Sharding
12. Security
13. Performance Optimization
14. Backup & Restore
15. Interview Questions & Answers

---

1. Introduction to MongoDB

What is MongoDB?

MongoDB is a NoSQL document-oriented database. Instead of storing data in tables with rows and columns (like SQL), it stores data in flexible, JSON-like documents.

Key Features:

- Schema-less (flexible structure)
- Horizontally scalable
- High performance
- Rich query language
- Built-in replication and sharding

SQL vs NoSQL Comparison

SQL (MySQL)		MongoDB
Database	→	Database
Table	→	Collection
Row	→	Document
Column	→	Field
Table Join	→	Embedded documents or $lookup
Primary Key	→	_id field (auto-generated)

When to Use MongoDB?

Good for:

- Rapidly changing schemas
- Hierarchical data storage
- Large scale data
- Real-time analytics
- Content management systems
- IoT applications
- Mobile apps

Not ideal for:

- Complex transactions (though now supported)
- Systems requiring complex joins
- Legacy systems built on SQL

---

2. Installation & Setup

Installation

Windows:

bash

```
# Download from MongoDB website
# Install MongoDB Community Server
# Add to PATH: C:\Program Files\MongoDB\Server\7.0\bin
```

Mac:

bash

```
brew tap mongodb/brew
brew install mongodb-community
brew services start mongodb-community
```

Linux (Ubuntu):


bash

```
wget -qO - https://www.mongodb.org/static/pgp/server-7.0.asc | sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/7.0 multiverse" | sudo tee /etc/a
sudo apt-get update
sudo apt-get install -y mongodb-org
sudo systemctl start mongod
```

Starting MongoDB


bash

```
# Start MongoDB service
mongod

# Connect to MongoDB shell
mongosh

# Check MongoDB version
mongod --version
```

Basic Shell Commands


javascript

```
// Show all databases
show dbs

// Create or switch to database
use myDatabase

// Show current database
db

// Show collections
show collections

// Get database stats
db.stats()

// Drop database
db.dropDatabase()
```

---

3. MongoDB Architecture

Document Structure

javascript

```
{
  _id: ObjectId("507f1f77bcf86cd799439011"),  // Unique identifier (auto-generated)
  name: "John Doe",
  age: 30,
  email: "john@example.com",
  address: {                          // Embedded document
    street: "123 Main St",
    city: "New York",
    zip: "10001"
  },
  hobbies: ["reading", "gaming", "coding"],   // Array
  created_at: ISODate("2024-01-15T10:30:00Z")
}
```

Data Types in MongoDB

javascript

```javascript
// String
{ name: "John" }

// Number (Integer, Long, Double)
{ age: 30, price: 99.99 }

// Boolean
{ isActive: true }

// Date
{ createdAt: new Date() }
{ createdAt: ISODate("2024-01-15") }

// Array
{ tags: ["mongodb", "database", "nosql"] }

// Embedded Document (Object)
{ address: { city: "NYC", zip: "10001" } }

// ObjectId (12-byte identifier)
{ _id: ObjectId("507f1f77bcf86cd799439011") }

// Null
{ middleName: null }

// Binary Data
{ file: BinData(0, "base64data") }

// Regular Expression
{ pattern: /^test/i }
```

Collections

Collections are groups of documents. They don't enforce a schema, so documents in the same collection can have different structures.

javascript

```javascript
// Create collection explicitly
db.createCollection("users")

// Create with options
db.createCollection("logs", {
  capped: true,       // Fixed size
  size: 5242880,      // Max size in bytes (5MB)
  max: 5000           // Max documents
})
```

---

4. CRUD Operations

CREATE Operations

insertOne()

javascript

```javascript
// Insert single document
db.users.insertOne({
  name: "Alice",
  age: 28,
  email: "alice@example.com",
  createdAt: new Date()
})

// Response
{
  acknowledged: true,
  insertedId: ObjectId("...")
}
```

insertMany()

javascript

```
// Insert multiple documents
db.users.insertMany([
  { name: "Bob", age: 35, email: "bob@example.com" },
  { name: "Charlie", age: 42, email: "charlie@example.com" },
  { name: "Diana", age: 29, email: "diana@example.com" }
])

// With ordered option (stops on first error if true)
db.users.insertMany([...], { ordered: false })
```

insert() (deprecated but still works)


javascript

```
db.users.insert({ name: "Eve", age: 31 })
```

READ Operations

find()


javascript

```javascript
// Find all documents
db.users.find()

// Find with filter
db.users.find({ age: 28 })

// Find with multiple conditions (AND)
db.users.find({ age: 28, name: "Alice" })

// Find with OR
db.users.find({
  $or: [
    { age: 28 },
    { name: "Bob" }
  ]
})

// Find with AND + OR
db.users.find({
  age: { $gt: 25 },
  $or: [
    { name: "Alice" },
    { name: "Bob" }
  ]
})
```

Comparison Operators


javascript

```javascript
// $eq (equal)
db.users.find({ age: { $eq: 30 } })

// $ne (not equal)
db.users.find({ age: { $ne: 30 } })

// $gt (greater than)
db.users.find({ age: { $gt: 30 } })

// $gte (greater than or equal)
db.users.find({ age: { $gte: 30 } })

// $lt (less than)
db.users.find({ age: { $lt: 30 } })

// $lte (less than or equal)
db.users.find({ age: { $lte: 30 } })

// $in (in array)
db.users.find({ age: { $in: [25, 30, 35] } })

// $nin (not in array)
db.users.find({ age: { $nin: [25, 30, 35] } })
```

Logical Operators


javascript

```javascript
// $and
db.users.find({
  $and: [
    { age: { $gt: 25 } },
    { age: { $lt: 40 } }
  ]
})

// $or
db.users.find({
  $or: [
    { name: "Alice" },
    { age: 35 }
  ]
})

// $not
db.users.find({ age: { $not: { $gt: 30 } } })

// $nor (not or)
db.users.find({
  $nor: [
    { age: { $lt: 25 } },
    { age: { $gt: 40 } }
  ]
})
```

Element Operators

javascript

```javascript
// $exists (check if field exists)
db.users.find({ email: { $exists: true } })

// $type (check field type)
db.users.find({ age: { $type: "number" } })
db.users.find({ age: { $type: 16 } })  // 16 = 32-bit integer
```

Array Operators

javascript

```javascript
// $all (array contains all elements)
db.users.find({ hobbies: { $all: ["reading", "coding"] } })

// $elemMatch (at least one element matches)
db.orders.find({
  items: {
    $elemMatch: { price: { $gt: 50 }, quantity: { $gte: 2 } }
  }
})

// $size (array length)
db.users.find({ hobbies: { $size: 3 } })
```

Query Modifiers

javascript

```javascript
// Projection (select specific fields)
db.users.find({}, { name: 1, email: 1, _id: 0 })

// Sort (1 = ascending, -1 = descending)
db.users.find().sort({ age: -1, name: 1 })

// Limit
db.users.find().limit(5)

// Skip (pagination)
db.users.find().skip(10).limit(5)

// Count
db.users.find({ age: { $gt: 30 } }).count()
db.users.countDocuments({ age: { $gt: 30 } })

// Distinct
db.users.distinct("age")
```

findOne()

javascript

```javascript
// Returns first matching document
db.users.findOne({ name: "Alice" })

// With projection
db.users.findOne({ name: "Alice" }, { email: 1 })
```

Querying Embedded Documents

javascript

```javascript
// Exact match
db.users.find({ address: { city: "NYC", zip: "10001" } })

// Dot notation (recommended)
db.users.find({ "address.city": "NYC" })
db.users.find({ "address.zip": "10001" })

// With operators
db.users.find({ "address.city": { $in: ["NYC", "LA"] } })
```

Querying Arrays

javascript

```javascript
// Match array element
db.users.find({ hobbies: "reading" })

// Match entire array
db.users.find({ hobbies: ["reading", "coding"] })

// Array with condition
db.users.find({ hobbies: { $in: ["reading", "gaming"] } })

// Array element with dot notation
db.orders.find({ "items.0.name": "Laptop" })  // First element
```

UPDATE Operations

updateOne()

javascript

```javascript
// Update first matching document
db.users.updateOne(
  { name: "Alice" },          // Filter
  { $set: { age: 29 } }       // Update
)

// Response
{
  acknowledged: true,
  matchedCount: 1,
  modifiedCount: 1
}
```

updateMany()

javascript

```javascript
// Update all matching documents
db.users.updateMany(
  { age: { $lt: 30 } },
  { $set: { status: "young" } }
)
```

Update Operators


javascript

```
// $set (set field value)
db.users.updateOne(
  { name: "Alice" },
  { $set: { age: 30, email: "newalice@example.com" } }
)

// $unset (remove field)
db.users.updateOne(
  { name: "Alice" },
  { $unset: { status: "" } }
)

// $inc (increment number)
db.users.updateOne(
  { name: "Alice" },
  { $inc: { age: 1 } }
)

// $mul (multiply number)
db.products.updateOne(
  { name: "Laptop" },
  { $mul: { price: 0.9 } }  // 10% discount
)

// $min (update if new value is less)
db.products.updateOne(
  { name: "Laptop" },
  { $min: { price: 800 } }
)

// $max (update if new value is greater)
db.products.updateOne(
  { name: "Laptop" },
  { $max: { price: 1200 } }
)

// $rename (rename field)
db.users.updateMany(
  {},
  { $rename: { "email": "emailAddress" } }
)
```

```javascript
// $currentDate (set to current date)
db.users.updateOne(
  { name: "Alice" },
  { $currentDate: { lastModified: true } }
)
```

Array Update Operators


javascript

```
// $push (add element to array)
db.users.updateOne(
  { name: "Alice" },
  { $push: { hobbies: "swimming" } }
)

// $push with $each (add multiple elements)
db.users.updateOne(
  { name: "Alice" },
  { $push: { hobbies: { $each: ["cooking", "traveling"] } } }
)

// $push with $sort
db.users.updateOne(
  { name: "Alice" },
  {
    $push: {
      scores: {
        $each: [85, 92],
        $sort: -1  // Sort descending
      }
    }
  }
)

// $push with $slice (limit array size)
db.users.updateOne(
  { name: "Alice" },
  {
    $push: {
      hobbies: {
        $each: ["dancing"],
        $slice: -5  // Keep last 5 elements
      }
    }
  }
)

// $addToSet (add if not exists)
db.users.updateOne(
  { name: "Alice" },
```

```
  { $addToSet: { hobbies: "reading" } }
)

// $pop (remove first or last element)
db.users.updateOne(
  { name: "Alice" },
  { $pop: { hobbies: 1 } }  // 1 = last, -1 = first
)

// $pull (remove matching elements)
db.users.updateOne(
  { name: "Alice" },
  { $pull: { hobbies: "gaming" } }
)

// $pull with condition
db.orders.updateOne(
  { _id: ObjectId("...") },
  { $pull: { items: { price: { $lt: 10 } } } }
)

// $pullAll (remove multiple values)
db.users.updateOne(
  { name: "Alice" },
  { $pullAll: { hobbies: ["gaming", "cooking"] } }
)

// $ (positional operator - update first match)
db.users.updateOne(
  { name: "Alice", "hobbies": "reading" },
  { $set: { "hobbies.$": "READING" } }
)

// $[] (update all array elements)
db.orders.updateOne(
  { _id: ObjectId("...") },
  { $set: { "items.$[].shipped": true } }
)

// $[element] (update filtered array elements)
db.orders.updateOne(
```

```
  { _id: ObjectId("...") },
  { $set: { "items.$[elem].discount": 10 } },
  { arrayFilters: [{ "elem.price": { $gt: 100 } }] }
)
```

replaceOne()

javascript

```
// Replace entire document
db.users.replaceOne(
  { name: "Alice" },
  {
    name: "Alice Johnson",
    age: 29,
    email: "alice.j@example.com"
  }
)
// Note: _id is preserved, all other fields are replaced
```

findOneAndUpdate()

javascript

```
// Update and return the document
db.users.findOneAndUpdate(
  { name: "Alice" },
  { $inc: { age: 1 } },
  { returnNewDocument: true }  // Return updated document
)
```

Upsert (Update or Insert)

javascript

```javascript
// Insert if not exists, update if exists
db.users.updateOne(
  { name: "Frank" },
  { $set: { age: 40, email: "frank@example.com" } },
  { upsert: true }
)
```

DELETE Operations

deleteOne()

javascript

```javascript
// Delete first matching document
db.users.deleteOne({ name: "Alice" })

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

deleteMany()

javascript

```javascript
// Delete all matching documents
db.users.deleteMany({ age: { $lt: 25 } })

// Delete all documents
db.users.deleteMany({})
```

findOneAndDelete()

javascript

```javascript
// Delete and return the document
db.users.findOneAndDelete({ name: "Alice" })
```

Drop Collection


javascript

```javascript
// Remove entire collection
db.users.drop()
```

---

5. Data Modeling

Embedded Documents (Denormalization)

When to use: One-to-One or One-to-Few relationships


javascript

```
// User with embedded address
{
  _id: ObjectId("..."),
  name: "John Doe",
  email: "john@example.com",
  address: {
    street: "123 Main St",
    city: "New York",
    state: "NY",
    zip: "10001"
  },
  phones: [
    { type: "home", number: "555-1234" },
    { type: "work", number: "555-5678" }
  ]
}

// Blog post with embedded comments
{
  _id: ObjectId("..."),
  title: "MongoDB Guide",
  content: "...",
  author: "John",
  comments: [
    { user: "Alice", text: "Great post!", date: ISODate("...") },
    { user: "Bob", text: "Very helpful", date: ISODate("...") }
  ]
}
```

Advantages:

- Better read performance (single query)
- Atomic updates
- Data locality

Disadvantages:

- Document size limit (16MB)
- Data duplication
- Harder to query embedded data

References (Normalization)

When to use: One-to-Many or Many-to-Many relationships


javascript

```javascript
// Users collection
{
  _id: ObjectId("user1"),
  name: "John Doe",
  email: "john@example.com"
}

// Orders collection (referencing user)
{
  _id: ObjectId("order1"),
  user_id: ObjectId("user1"),  // Reference
  total: 299.99,
  items: [...]
}

// Query with reference
db.orders.find({ user_id: ObjectId("user1") })
```

Many-to-Many Example:


javascript

```javascript
// Students collection
{ _id: 1, name: "Alice", course_ids: [101, 102, 103] }
{ _id: 2, name: "Bob", course_ids: [101, 104] }

// Courses collection
{ _id: 101, name: "MongoDB Basics", student_ids: [1, 2] }
{ _id: 102, name: "Node.js", student_ids: [1] }
```

Hybrid Approach


javascript

```javascript
// Store frequently accessed data embedded, reference for full details
{
  _id: ObjectId("order1"),
  user: {
    id: ObjectId("user1"),
    name: "John Doe",        // Embedded for quick access
    email: "john@example.com"
  },
  items: [
    {
      product_id: ObjectId("prod1"),
      name: "Laptop",        // Embedded
      price: 999,
      // Full product details in products collection
    }
  ]
}
```

Design Patterns

1. Attribute Pattern

javascript

```javascript
// Instead of creating fields for each attribute
{
  product: "Laptop",
  color: "Silver",
  size: "15-inch",
  weight: "4.5 lbs"
}

// Use attribute array for flexible attributes
{
  product: "Laptop",
  attributes: [
    { k: "color", v: "Silver" },
    { k: "size", v: "15-inch" },
    { k: "weight", v: "4.5 lbs" }
  ]
}
// Create index on attributes.k and attributes.v
```

2. Bucket Pattern

javascript

```
// Instead of one document per measurement
{ sensor_id: 123, temp: 20.5, timestamp: ISODate("2024-01-01T10:00:00Z") }
{ sensor_id: 123, temp: 20.7, timestamp: ISODate("2024-01-01T10:01:00Z") }

// Group measurements into buckets
{
  sensor_id: 123,
  date: ISODate("2024-01-01"),
  measurements: [
    { temp: 20.5, timestamp: ISODate("2024-01-01T10:00:00Z") },
    { temp: 20.7, timestamp: ISODate("2024-01-01T10:01:00Z") },
    // ... more measurements
  ],
  count: 60,  // Number of measurements
  avg_temp: 20.6
}
```

3. Outlier Pattern

javascript

```javascript
// Most products have few reviews, but some have thousands
// Normal product
{
  _id: ObjectId("prod1"),
  name: "Mouse",
  reviews: [
    { user: "Alice", rating: 5, text: "Great!" },
    { user: "Bob", rating: 4, text: "Good" }
  ]
}

// Popular product (use reference)
{
  _id: ObjectId("prod2"),
  name: "iPhone",
  review_count: 5000,
  has_overflow: true  // Flag for outlier
}

// Separate reviews collection for outliers
{
  product_id: ObjectId("prod2"),
  user: "Charlie",
  rating: 5,
  text: "Amazing!"
}
```

4. Extended Reference Pattern

javascript

```
// Instead of just storing reference
{
  order_id: ObjectId("..."),
  customer_id: ObjectId("cust1")
}

// Store frequently accessed fields
{
  order_id: ObjectId("..."),
  customer: {
    id: ObjectId("cust1"),
    name: "John Doe",          // Duplicate for performance
    email: "john@example.com"  // Duplicate for performance
  }
}
```

Schema Validation

javascript

```
// Create collection with validation rules
db.createCollection("users", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "email", "age"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        email: {
          bsonType: "string",
          pattern: "^.+@.+$",
          description: "must be a valid email"
        },
        age: {
          bsonType: "int",
          minimum: 0,
          maximum: 150,
          description: "must be an integer between 0 and 150"
        },
        status: {
          enum: ["active", "inactive", "pending"],
          description: "can only be one of the enum values"
        }
      }
    }
  },
  validationLevel: "strict",    // or "moderate"
  validationAction: "error"     // or "warn"
})

// Add validation to existing collection
db.runCommand({
  collMod: "users",
  validator: { ... }
})
```

---

6. Querying Deep Dive

Regular Expressions

javascript

```javascript
// Case-insensitive search
db.users.find({ name: { $regex: /john/i } })
db.users.find({ name: { $regex: "john", $options: "i" } })

// Starts with
db.users.find({ name: { $regex: /^john/i } })

// Ends with
db.users.find({ name: { $regex: /doe$/i } })

// Contains
db.users.find({ name: { $regex: /john/i } })
```

Text Search

javascript

```javascript
// Create text index
db.articles.createIndex({ title: "text", content: "text" })

// Search for text
db.articles.find({ $text: { $search: "mongodb database" } })

// Search phrase
db.articles.find({ $text: { $search: "\"mongodb guide\"" } })

// Exclude words
db.articles.find({ $text: { $search: "mongodb -sql" } })

// Get text score
db.articles.find(
  { $text: { $search: "mongodb" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } })
```

Cursor Methods

javascript

```javascript
// Get cursor
var cursor = db.users.find()

// Iterate cursor
cursor.forEach(doc => print(doc.name))

// Convert to array
var users = db.users.find().toArray()

// Check if cursor has next
cursor.hasNext()

// Get next document
cursor.next()

// Cursor count
cursor.count()

// Explain query execution
db.users.find({ age: { $gt: 30 } }).explain("executionStats")
```

Geospatial Queries

javascript

```
// Create 2dsphere index
db.places.createIndex({ location: "2dsphere" })

// Insert location data
db.places.insertOne({
  name: "Central Park",
  location: {
    type: "Point",
    coordinates: [-73.968285, 40.785091]  // [longitude, latitude]
  }
})

// Find near a point
db.places.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [-73.97, 40.78]
      },
      $maxDistance: 1000  // meters
    }
  }
})

// Find within area
db.places.find({
  location: {
    $geoWithin: {
      $centerSphere: [[-73.97, 40.78], 10 / 3963.2]  // 10 miles radius
    }
  }
})

// Polygon search
db.places.find({
  location: {
    $geoWithin: {
      $geometry: {
        type: "Polygon",
        coordinates: [[
```

```
        [-73.99, 40.75],
        [-73.98, 40.75],
        [-73.98, 40.76],
        [-73.99, 40.76],
        [-73.99, 40.75]
      ]]
    }
  }
 }
})
```

---

7. Indexing

Why Indexes Matter

Without index: MongoDB scans every document (COLLSCAN) With index: MongoDB uses index tree structure (IXSCAN)

javascript

```javascript
// Without index - slow on large collections
db.users.find({ email: "alice@example.com" })  // Scans all documents

// Create index
db.users.createIndex({ email: 1 })

// With index - fast
db.users.find({ email: "alice@example.com" })  // Uses index
```

Single Field Index

javascript

```javascript
// Ascending index
db.users.createIndex({ age: 1 })

// Descending index
db.users.createIndex({ age: -1 })

// Check existing indexes
db.users.getIndexes()

// Drop index
db.users.dropIndex({ age: 1 })
db.users.dropIndex("age_1")  // By name

// Drop all indexes (except _id)
db.users.dropIndexes()
```

Compound Index

javascript

```javascript
// Index on multiple fields
db.users.createIndex({ age: 1, name: 1 })

// Order matters!
// Good for: { age: 30, name: "John" }
// Good for: { age: 30 }
// NOT optimized for: { name: "John" }

// Best practice: Equality, Sort, Range (ESR)
db.orders.createIndex({
  status: 1,      // Equality
  created_at: -1, // Sort
  total: 1        // Range
})
```

Index Types

1. Unique Index

javascript

```javascript
// Prevent duplicate values
db.users.createIndex({ email: 1 }, { unique: true })

// Compound unique
db.users.createIndex({ email: 1, phone: 1 }, { unique: true })
```

2. Sparse Index

javascript

```javascript
// Index only documents that have the field
db.users.createIndex({ phone: 1 }, { sparse: true })
// Documents without 'phone' field won't be in index
```

3. TTL Index (Time To Live)

javascript

```javascript
// Auto-delete documents after specified time
db.sessions.createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 3600 }  // Delete after 1 hour
)

// Insert with createdAt
db.sessions.insertOne({
  user_id: "user123",
  createdAt: new Date()
})
```

4. Partial Index

javascript

```javascript
// Index only documents matching filter
db.orders.createIndex(
  { customer_id: 1, total: 1 },
  { partialFilterExpression: { status: "active" } }
)
// Only active orders are indexed
```

5. Text Index

javascript

```javascript
// For text search
db.articles.createIndex({ content: "text" })

// Multiple fields
db.articles.createIndex({
  title: "text",
  content: "text"
})

// With weights (importance)
db.articles.createIndex(
  {
    title: "text",
    content: "text"
  },
  {
    weights: {
      title: 10,
      content: 5
    }
  }
)
```

6. Wildcard Index

javascript

```javascript
// Index all fields
db.products.createIndex({ "$**": 1 })

// Index all subfields of specific field
db.products.createIndex({ "attributes.$**": 1 })
```

7. Hashed Index

```javascript
// For sharding
db.users.createIndex({ _id: "hashed" })
```

8. 2dsphere Index (Geospatial)

```javascript
db.places.createIndex({ location: "2dsphere" })
```

Index Properties

```javascript
```

```javascript
// Background building (don't block operations)
db.users.createIndex({ age: 1 }, { background: true })

// Unique
db.users.createIndex({ email: 1 }, { unique: true })

// Sparse
db.users.createIndex({ phone: 1 }, { sparse: true })

// Name
db.users.createIndex({ age: 1 }, { name: "age_ascending" })

// Partial
db.orders.createIndex(
  { total: 1 },
  { partialFilterExpression: { total: { $gt: 100 } } }
)

// TTL
db.logs.createIndex({ timestamp: 1 }, { expireAfterSeconds: 86400 })

// Case insensitive (collation)
db.users.createIndex(
  { name: 1 },
  { collation: { locale: "en", strength: 2 } }
)
```

Index Analysis

javascript

```javascript
// Explain query plan
db.users.find({ age: 30 }).explain("executionStats")

// Key fields in explain output:
// - executionTimeMillis: Time taken
// - totalDocsExamined: Documents scanned
// - totalKeysExamined: Index entries scanned
// - stage: IXSCAN (index) or COLLSCAN (collection)

// Index statistics
db.users.aggregate([{ $indexStats: {} }])

// Get index sizes
db.users.stats().indexSizes
```

Covered Queries

javascript

```
// Query covered entirely by index (no document lookup)
db.users.createIndex({ name: 1, age: 1 })

// This query is covered (returns only indexed fields)
db.users.find(
  { name: "John" },
  { name: 1, age: 1, _id: 0 }
)
// executionStats shows totalDocsExamined: 0
```

Index Best Practices

1. **Create indexes for frequently queried fields**
2. **Use compound indexes wisely (ESR rule)**
3. **Limit number of indexes (write performance impact)**
4. **Use covered queries when possible**
5. **Monitor index usage with $indexStats**
6. **Delete unused indexes**
7. **Consider partial indexes for large collections**

---

8. Aggregation Framework

The aggregation pipeline processes documents through stages, each transforming the data.

Basic Pipeline

javascript

```javascript
db.collection.aggregate([
  { $match: { ... } },      // Stage 1: Filter
  { $group: { ... } },      // Stage 2: Group
  { $sort: { ... } },       // Stage 3: Sort
  { $project: { ... } }     // Stage 4: Shape output
])
```

Common Aggregation Stages

$match (Filter)

javascript

```javascript
// Filter documents (like find)
db.orders.aggregate([
  { $match: { status: "completed" } }
])

// With operators
db.orders.aggregate([
  { $match: {
    total: { $gt: 100 },
    status: "completed"
  }}
])
```

$project (Shape Output)

javascript

```javascript
// Select/exclude fields
db.users.aggregate([
  { $project: {
    name: 1,
    email: 1,
    _id: 0
  }}
])

// Compute new fields
db.orders.aggregate([
  { $project: {
    total: 1,
    tax: { $multiply: ["$total", 0.1] },
    totalWithTax: { $add: ["$total", { $multiply: ["$total", 0.1] }] }
  }}
])

// String operations
db.users.aggregate([
  { $project: {
    fullName: { $concat: ["$firstName", " ", "$lastName"] },
    upperName: { $toUpper: "$name" },
    emailDomain: { $arrayElemAt: [{ $split: ["$email", "@"] }, 1] }
  }}
])
```

$group (Aggregation)

javascript

```
// Count documents
db.orders.aggregate([
  { $group: {
    _id: null,
    count: { $sum: 1 }
  }}
])

// Group by field
db.orders.aggregate([
  { $group: {
    _id: "$status",
    count: { $sum: 1 }
  }}
])

// Multiple grouping fields
db.orders.aggregate([
  { $group: {
    _id: {
      status: "$status",
      year: { $year: "$created_at" }
    },
    count: { $sum: 1 }
  }}
])

// Aggregation operators
db.orders.aggregate([
  { $group: {
    _id: "$customer_id",
    totalSpent: { $sum: "$total" },      // Sum
    avgOrder: { $avg: "$total" },        // Average
    maxOrder: { $max: "$total" },        // Maximum
    minOrder: { $min: "$total" },        // Minimum
    firstOrder: { $first: "$created_at" }, // First value
    lastOrder: { $last: "$created_at" },   // Last value
    orders: { $push: "$_id" },           // Array of values
    uniqueStatuses: { $addToSet: "$status" } // Unique values
```

```
  }}
])
```

$sort

javascript

```
// Sort results
db.orders.aggregate([
  { $group: {
    _id: "$customer_id",
    totalSpent: { $sum: "$total" }
  }},
  { $sort: { totalSpent: -1 } }  // Descending
])
```

$limit and $skip

javascript

```
// Pagination
db.orders.aggregate([
  { $sort: { created_at: -1 } },
  { $skip: 20 },
  { $limit: 10 }
])
```

$unwind (Deconstruct Array)

javascript

```
// Deconstruct array field
db.orders.aggregate([
  { $unwind: "$items" }
])

// Before: { _id: 1, items: ["A", "B", "C"] }
// After:  { _id: 1, items: "A" }
//         { _id: 1, items: "B" }
//         { _id: 1, items: "C" }

// Preserve empty arrays
db.orders.aggregate([
  { $unwind: {
    path: "$items",
    preserveNullAndEmptyArrays: true
  }}
])
```

$lookup (Join)

javascript

```
// Left outer join
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",           // Collection to join
      localField: "customer_id",   // Field from orders
      foreignField: "_id",         // Field from customers
      as: "customerInfo"           // Output array field
    }
  }
])

// Unwind result
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",
      foreignField: "_id",
      as: "customer"
    }
  },
  { $unwind: "$customer" }
])

// Complex lookup with pipeline
db.orders.aggregate([
  {
    $lookup: {
      from: "products",
      let: { order_items: "$items" },
      pipeline: [
        { $match: {
          $expr: { $in: ["$_id", "$order_items"] }
        }},
        { $project: { name: 1, price: 1 } }
      ],
      as: "productDetails"
    }
```

```
  }
])
```

$addFields

javascript

```
// Add new fields without removing existing ones
db.orders.aggregate([
  { $addFields: {
    totalWithTax: { $multiply: ["$total", 1.1] },
    year: { $year: "$created_at" }
  }}
])
```

$replaceRoot

javascript

```
// Replace document with embedded document
db.orders.aggregate([
  { $replaceRoot: { newRoot: "$customer" } }
])
```

$out

javascript

```
// Write results to new collection
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: {
    _id: "$customer_id",
    total: { $sum: "$total" }
  }},
  { $out: "customer_totals" }  // Creates new collection
])
```

$merge

javascript

```
// Merge results into existing collection
db.orders.aggregate([
  { $group: {
    _id: "$customer_id",
    totalSpent: { $sum: "$total" }
  }},
  { $merge: {
    into: "customers",
    on: "_id",
    whenMatched: "merge",  // or "replace", "keepExisting", "fail"
    whenNotMatched: "discard"  // or "insert"
  }}
])
```

$bucket

javascript

```javascript
// Group by ranges
db.users.aggregate([
  {
    $bucket: {
      groupBy: "$age",
      boundaries: [0, 18, 30, 50, 100],
      default: "Other",
      output: {
        count: { $sum: 1 },
        users: { $push: "$name" }
      }
    }
  }
])
```

$facet


javascript

```
// Multiple parallel pipelines
db.products.aggregate([
  {
    $facet: {
      categoryCounts: [
        { $group: { _id: "$category", count: { $sum: 1 } } }
      ],
      priceStats: [
        { $group: {
          _id: null,
          avgPrice: { $avg: "$price" },
          maxPrice: { $max: "$price" }
        }}
      ],
      topProducts: [
        { $sort: { sales: -1 } },
        { $limit: 5 }
      ]
    }
  }
])
```

Real-World Examples

Example 1: Sales Report

javascript

```
db.orders.aggregate([
  // Filter completed orders
  { $match: {
    status: "completed",
    created_at: { $gte: ISODate("2024-01-01") }
  }},

  // Unwind items array
  { $unwind: "$items" },

  // Group by product
  { $group: {
    _id: "$items.product_id",
    totalQuantity: { $sum: "$items.quantity" },
    totalRevenue: { $sum: { $multiply: ["$items.quantity", "$items.price"] } },
    orderCount: { $sum: 1 }
  }},

  // Lookup product details
  { $lookup: {
    from: "products",
    localField: "_id",
    foreignField: "_id",
    as: "product"
  }},
  { $unwind: "$product" },

  // Shape output
  { $project: {
    _id: 0,
    productName: "$product.name",
    totalQuantity: 1,
    totalRevenue: 1,
    orderCount: 1,
    avgOrderValue: { $divide: ["$totalRevenue", "$orderCount"] }
  }},

  // Sort by revenue
  { $sort: { totalRevenue: -1 } },

  // Top 10
```

```
  { $limit: 10 }
])
```

Example 2: Customer Analysis

javascript

```
db.orders.aggregate([
  // Group by customer
  { $group: {
    _id: "$customer_id",
    totalOrders: { $sum: 1 },
    totalSpent: { $sum: "$total" },
    avgOrderValue: { $avg: "$total" },
    firstOrder: { $min: "$created_at" },
    lastOrder: { $max: "$created_at" }
  }},

  // Calculate days since first order
  { $addFields: {
    daysSinceFirst: {
      $divide: [
        { $subtract: [new Date(), "$firstOrder"] },
        1000 * 60 * 60 * 24
      ]
    }
  }},

  // Customer lifetime value per day
  { $addFields: {
    clvPerDay: { $divide: ["$totalSpent", "$daysSinceFirst"] }
  }},

  // Join with customer details
  { $lookup: {
    from: "customers",
    localField: "_id",
    foreignField: "_id",
    as: "customer"
  }},
  { $unwind: "$customer" },

  // Filter high-value customers
  { $match: { totalSpent: { $gt: 1000 } } },

  // Sort by CLV
```

```
  { $sort: { clvPerDay: -1 } }
])
```

Example 3: Time Series Analysis

javascript

```
db.orders.aggregate([
  // Group by year and month
  { $group: {
    _id: {
      year: { $year: "$created_at" },
      month: { $month: "$created_at" }
    },
    revenue: { $sum: "$total" },
    orders: { $sum: 1 },
    avgOrderValue: { $avg: "$total" }
  }},

  // Sort chronologically
  { $sort: { "_id.year": 1, "_id.month": 1 } },

  // Calculate month-over-month growth
  { $setWindowFields: {
    sortBy: { "_id.year": 1, "_id.month": 1 },
    output: {
      previousRevenue: {
        $shift: {
          output: "$revenue",
          by: -1
        }
      }
    }
  }},

  { $addFields: {
    growthPercent: {
      $multiply: [
        { $divide: [
          { $subtract: ["$revenue", "$previousRevenue"] },
          "$previousRevenue"
        ]},
        100
      ]
    }
  }},

  // Format output
```

```
  { $project: {
    _id: 0,
    period: { $concat: [
      { $toString: "$_id.year" },
      "-",
      { $toString: "$_id.month" }
    ]},
    revenue: { $round: ["$revenue", 2] },
    orders: 1,
    avgOrderValue: { $round: ["$avgOrderValue", 2] },
    growthPercent: { $round: ["$growthPercent", 2] }
  }}
])
```

---

9. Transactions

MongoDB supports ACID transactions for multi-document operations.

Single Document Transactions

javascript

```
// Single document operations are ALWAYS atomic
db.accounts.updateOne(
  { _id: "account1" },
  { $inc: { balance: -100 } }
)
// This is atomic - either completes fully or not at all
```

Multi-Document Transactions

javascript

```
// Start a session
const session = db.getMongo().startSession()

// Start transaction
session.startTransaction()

try {
  const accounts = session.getDatabase("bank").accounts

  // Debit from account1
  accounts.updateOne(
    { _id: "account1" },
    { $inc: { balance: -100 } },
    { session }
  )

  // Credit to account2
  accounts.updateOne(
    { _id: "account2" },
    { $inc: { balance: 100 } },
    { session }
  )

  // Commit transaction
  session.commitTransaction()
  print("Transaction committed")

} catch (error) {
  // Abort transaction on error
  session.abortTransaction()
  print("Transaction aborted: " + error)
} finally {
  session.endSession()
}
```

Transaction with Node.js Driver

javascript

```
const { MongoClient } = require('mongodb')

async function transferMoney() {
  const client = new MongoClient('mongodb://localhost:27017')
  await client.connect()

  const session = client.startSession()

  try {
    await session.withTransaction(async () => {
      const accounts = client.db('bank').collection('accounts')

      await accounts.updateOne(
        { _id: 'account1' },
        { $inc: { balance: -100 } },
        { session }
      )

      await accounts.updateOne(
        { _id: 'account2' },
        { $inc: { balance: 100 } },
        { session }
      )
    })

    console.log('Transaction successful')
  } catch (error) {
    console.log('Transaction failed:', error)
  } finally {
    await session.endSession()
    await client.close()
  }
}
```

Transaction Best Practices

1. **Keep transactions short** (ideally < 1 second)
2. **Limit operations per transaction** (< 1000 documents)
3. **Use appropriate read/write concerns**
4. **Handle retries** (transient errors)
5. **Consider using single-document operations when possible**

---

10. Replication

Replication provides redundancy and high availability.

Replica Set Architecture

```
Primary Node (Read/Write)
    |
    |-- Secondary Node (Read only)
    |-- Secondary Node (Read only)
    |-- Arbiter (Voting only, no data)
```

Setting Up Replica Set

bash

```bash
# Start three mongod instances
mongod --replSet rs0 --port 27017 --dbpath /data/db1
mongod --replSet rs0 --port 27018 --dbpath /data/db2
mongod --replSet rs0 --port 27019 --dbpath /data/db3
```

javascript

```javascript
// Connect to first instance
mongosh --port 27017

// Initiate replica set
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "localhost:27017" },
    { _id: 1, host: "localhost:27018" },
    { _id: 2, host: "localhost:27019" }
  ]
})

// Check replica set status
rs.status()

// Check replica set configuration
rs.conf()

// Add member
rs.add("localhost:27020")

// Remove member
rs.remove("localhost:27020")
```

Replica Set Member Types

javascript

```javascript
// Priority (likelihood of becoming primary)
rs.reconfig({
  _id: "rs0",
  members: [
    { _id: 0, host: "localhost:27017", priority: 2 },  // Preferred primary
    { _id: 1, host: "localhost:27018", priority: 1 },
    { _id: 2, host: "localhost:27019", priority: 0 }   // Never primary
  ]
})

// Hidden member (doesn't receive client reads)
{ _id: 3, host: "localhost:27020", hidden: true, priority: 0 }

// Delayed member (for backup/disaster recovery)
{ _id: 4, host: "localhost:27021", slaveDelay: 3600, priority: 0 }  // 1 hour delay

// Arbiter (voting only, no data)
rs.addArb("localhost:27022")
```

Read Preferences

javascript

```javascript
// Read from primary (default)
db.users.find().readPref("primary")

// Read from primary preferred (primary if available, else secondary)
db.users.find().readPref("primaryPreferred")

// Read from secondary
db.users.find().readPref("secondary")

// Read from secondary preferred
db.users.find().readPref("secondaryPreferred")

// Read from nearest (lowest latency)
db.users.find().readPref("nearest")
```

Write Concerns

javascript

```javascript
// Write acknowledged by primary only (default)
db.users.insertOne(
  { name: "John" },
  { writeConcern: { w: 1 } }
)

// Write acknowledged by majority
db.users.insertOne(
  { name: "John" },
  { writeConcern: { w: "majority" } }
)

// Write acknowledged by specific number
db.users.insertOne(
  { name: "John" },
  { writeConcern: { w: 2 } }
)

// Write with timeout
db.users.insertOne(
  { name: "John" },
  { writeConcern: { w: "majority", wtimeout: 5000 } }
)

// Journaled write
db.users.insertOne(
  { name: "John" },
  { writeConcern: { w: 1, j: true } }
)
```

Read Concerns

javascript

```
// Local (default - reads latest data)
db.users.find().readConcern("local")

// Available (no guarantee of durability)
db.users.find().readConcern("available")

// Majority (reads data acknowledged by majority)
db.users.find().readConcern("majority")

// Linearizable (reads reflect all prior majority writes)
db.users.find().readConcern("linearizable")

// Snapshot (for transactions)
session.startTransaction({ readConcern: { level: "snapshot" } })
```

---

11. Sharding

Sharding distributes data across multiple machines for horizontal scaling.

Sharding Architecture

```
Client Application
    |
mongos (Router)
    |
Config Servers (Metadata)
    |
Shard 1 (Replica Set) | Shard 2 (Replica Set) | Shard 3 (Replica Set)
```

Setting Up Sharding

1. Start Config Servers

bash

```
mongod --configsvr --replSet configRS --port 27019 --dbpath /data/configdb
```

2. Start Shards

bash

```
mongod --shardsvr --replSet shard1RS --port 27018 --dbpath /data/shard1
mongod --shardsvr --replSet shard2RS --port 27020 --dbpath /data/shard2
```

3. Start mongos

bash

```
mongos --configdb configRS/localhost:27019 --port 27017
```

4. Add Shards

javascript

```
// Connect to mongos
mongosh --port 27017

// Add shards
sh.addShard("shard1RS/localhost:27018")
sh.addShard("shard2RS/localhost:27020")

// Check status
sh.status()
```

5. Enable Sharding

javascript

```javascript
// Enable sharding on database
sh.enableSharding("myDatabase")

// Shard collection
sh.shardCollection("myDatabase.users", { user_id: 1 })

// Check sharding status
db.users.getShardDistribution()
```

Shard Key Selection

javascript

```javascript
// Hashed shard key (even distribution)
sh.shardCollection("myDatabase.users", { _id: "hashed" })

// Range-based shard key
sh.shardCollection("myDatabase.orders", { customer_id: 1, created_at: 1 })

// Compound shard key
sh.shardCollection("myDatabase.products", { category: 1, product_id: 1 })
```

Shard Key Best Practices

Good Shard Keys:

- High cardinality (many unique values)
- Even distribution
- Query isolation (most queries include shard key)

Bad Shard Keys:

- Monotonically increasing (e.g., timestamp, auto-increment)
- Low cardinality (e.g., country, status)
- Single-value dominated

Example:

javascript

```
// BAD: Monotonically increasing
sh.shardCollection("logs", { timestamp: 1 })  // All writes go to one shard

// BETTER: Hashed timestamp
sh.shardCollection("logs", { timestamp: "hashed" })

// BEST: Compound key
sh.shardCollection("logs", { user_id: 1, timestamp: 1 })
```

Chunk Management

javascript

```
// Check chunk distribution
db.chunks.find({ ns: "myDatabase.users" }).count()

// Split chunk
sh.splitAt("myDatabase.users", { user_id: 5000 })

// Move chunk
sh.moveChunk("myDatabase.users", { user_id: 1000 }, "shard2RS")

// Enable/disable balancer
sh.enableBalancing("myDatabase.users")
sh.disableBalancing("myDatabase.users")

// Check balancer status
sh.getBalancerState()
```

Targeted vs Broadcast Operations

javascript

```javascript
// Targeted query (includes shard key)
db.users.find({ user_id: 12345 })  // Goes to specific shard

// Broadcast query (no shard key)
db.users.find({ email: "john@example.com" })  // Goes to all shards

// Targeted update
db.users.updateOne(
  { user_id: 12345 },  // Includes shard key
  { $set: { status: "active" } }
)

// Broadcast update
db.users.updateMany(
  { status: "inactive" },  // No shard key
  { $set: { archived: true } }
)
```

---

12. Security

Authentication

Enable Authentication


javascript

```javascript
// Create admin user
use admin
db.createUser({
  user: "admin",
  pwd: "securePassword123",
  roles: ["root"]
})
```


bash

```
# Restart mongod with authentication
mongod --auth --port 27017 --dbpath /data/db

# Connect with authentication
mongosh -u admin -p securePassword123 --authenticationDatabase admin
```

Create Database Users

javascript

```
// Create read-only user
use myDatabase
db.createUser({
  user: "reader",
  pwd: "password123",
  roles: [{ role: "read", db: "myDatabase" }]
})

// Create read-write user
db.createUser({
  user: "writer",
  pwd: "password123",
  roles: [{ role: "readWrite", db: "myDatabase" }]
})

// Custom roles
db.createUser({
  user: "customUser",
  pwd: "password123",
  roles: [
    { role: "read", db: "myDatabase" },
    { role: "readWrite", db: "analytics" }
  ]
})
```

Built-in Roles

javascript

```
// Database roles
- read: Read data
- readWrite: Read and write data
- dbAdmin: Database administration
- dbOwner: Database owner (all privileges)

// Cluster roles
- clusterAdmin: Cluster administration
- clusterManager: Monitor and manage cluster
- clusterMonitor: Read-only access to monitoring tools

// Backup/Restore roles
- backup: Backup data
- restore: Restore data

// All-database roles
- readAnyDatabase: Read all databases
- readWriteAnyDatabase: Read/write all databases
- dbAdminAnyDatabase: Admin all databases
- userAdminAnyDatabase: Manage users in all databases

// Superuser role
- root: Full access to all resources
```

User Management

javascript

```
// List users
db.getUsers()

// View user
db.getUser("username")

// Update user password
db.changeUserPassword("username", "newPassword")

// Grant role to user
db.grantRolesToUser("username", [{ role: "readWrite", db: "myDatabase" }])

// Revoke role from user
db.revokeRolesFromUser("username", [{ role: "read", db: "myDatabase" }])

// Drop user
db.dropUser("username")
```

Network Security

bash

```
# Bind to specific IP
mongod --bind_ip localhost,192.168.1.100

# Enable TLS/SSL
mongod --tlsMode requireTLS \
       --tlsCertificateKeyFile /path/to/mongodb.pem \
       --tlsCAFile /path/to/ca.pem
```

Encryption

Encryption at Rest

bash

```
mongod --enableEncryption \
    --encryptionKeyFile /path/to/keyfile
```

Encryption in Transit (TLS)


bash

```
mongod --tlsMode requireTLS \
    --tlsCertificateKeyFile /path/to/cert.pem
```

Field-Level Encryption


javascript

```
const { MongoClient, ClientEncryption } = require('mongodb')

// Configure encryption
const client = new MongoClient(uri, {
  autoEncryption: {
    keyVaultNamespace: 'encryption.__keyVault',
    kmsProviders: {
      local: {
        key: Buffer.from(localMasterKey, 'base64')
      }
    }
  }
})

// Insert with automatic encryption
await client.db('test').collection('users').insertOne({
  name: 'John',
  ssn: '123-45-6789'  // Automatically encrypted
})
```

Auditing

bash

```
# Enable auditing
mongod --auditDestination file \
    --auditFormat JSON \
    --auditPath /var/log/mongodb/audit.json \
    --auditFilter '{ atype: { $in: ["authenticate", "createUser", "dropDatabase"] } }'
```

javascript

```
// Audit filter examples
// Log all authentication attempts
{ atype: "authenticate" }

// Log all write operations
{ atype: { $in: ["insert", "update", "delete"] } }

// Log operations on specific database
{ "param.ns": /^myDatabase\./ }
```

---

13. Performance Optimization

Query Optimization

1. Use Indexes

javascript

```
// Before (slow)
db.users.find({ email: "john@example.com" })  // COLLSCAN

// Create index
db.users.createIndex({ email: 1 })

// After (fast)
db.users.find({ email: "john@example.com" })  // IXSCAN
```

2. Use Projection


javascript

```javascript
// Bad (retrieves all fields)
db.users.find({ age: { $gt: 25 } })

// Good (retrieves only needed fields)
db.users.find(
  { age: { $gt: 25 } },
  { name: 1, email: 1, _id: 0 }
)
```

3. Avoid Regex Without Index


javascript

```javascript
// Very slow
db.users.find({ name: { $regex: /john/i } })

// Better: Use text index
db.users.createIndex({ name: "text" })
db.users.find({ $text: { $search: "john" } })

// Or anchor regex
db.users.find({ name: { $regex: /^john/i } })  // Can use index
```

4. Use Covered Queries


javascript

```javascript
db.users.createIndex({ name: 1, age: 1 })

// Covered query (no document access needed)
db.users.find(
  { name: "John" },
  { name: 1, age: 1, _id: 0 }
)
```

5. Limit Results

javascript

```
// Bad
db.users.find()  // Returns all

// Good
db.users.find().limit(100)
```

6. Use $in Efficiently

javascript

```
// Bad (multiple queries)
db.users.find({ id: 1 })
db.users.find({ id: 2 })
db.users.find({ id: 3 })

// Good (single query)
db.users.find({ id: { $in: [1, 2, 3] } })

// But limit size of $in array (< 1000 elements)
```

Schema Design for Performance

1. Embed for Read Performance

javascript

```javascript
// One query to get everything
{
  _id: 1,
  name: "John",
  address: {
    street: "123 Main St",
    city: "NYC"
  },
  orders: [
    { id: 101, total: 50 },
    { id: 102, total: 75 }
  ]
}
```

2. Reference for Write Performance

javascript

```javascript
// Users collection
{ _id: 1, name: "John" }

// Orders collection (many writes don't affect user doc)
{ _id: 101, user_id: 1, total: 50 }
{ _id: 102, user_id: 1, total: 75 }
```

3. Use Appropriate Data Types

javascript

```javascript
// Bad
{ age: "25" }  // String

// Good
{ age: 25 }  // Number

// Bad
{ created_at: "2024-01-15" }  // String
```