MongoDB Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Create a compound index for optimizing range and equality queries

```
db.orders.createIndex(
  {status: 1, orderDate: -1, customerId: 1},
  {background: true}
)
```

2. Write a query to perform a faceted search with multiple bucket ranges

```
db.products.aggregate([
  {$facet: {
    priceRanges: [
      {$bucket: {
        groupBy: '$price',
        boundaries: [0, 100, 500, 1000],
        default: 'Other'
      }}
    ]
  }}
])
```

3. Implement a MongoDB update to modify nested array elements conditionally

```
db.users.updateMany(
  {'orders.status': 'PENDING'},
  {$set: {'orders.$.status': 'PROCESSING'},
   $currentDate: {'orders.$.lastModified': true}}
)
```

4. Write a query to perform text search with language-specific stemming

```
db.articles.find(
  {$text: {$search: 'programming databases',
   $language: 'english',
   $caseSensitive: false}},
  {score: {$meta: 'textScore'}}
)
```

5. Implement a MongoDB aggregation to perform window operations

```
db.sales.aggregate([
  {$setWindowFields: {
    partitionBy: '$region',
    sortBy: {date: 1},
    output: {movingAvg: {$avg: '$amount', window: {documents: [-2, 0]}}}
  }}
])
```

6. Create a query to find documents with geospatial intersection

```
db.locations.find({
  area: {$geoIntersects: {
    $geometry: {
      type: 'Polygon',
      coordinates: [[[0,0], [3,6], [6,1], [0,0]]]
    }
  }}
})
```

7. Write an update operation to handle schema migration

```
db.users.updateMany(
  {newField: {$exists: false}},
  [{$set: {
    newField: {$concat: ['$firstName', ' ', '$lastName']},
    schemaVersion: 2
  }}]
)
```

8. Implement a MongoDB aggregation pipeline to calculate the average order value per customer, excluding cancelled orders

```
db.orders.aggregate([
  {$match: {status: {$ne: 'CANCELLED'}}},
  {$group: {
    _id: '$customerId',
    avgOrderValue: {$avg: '$totalAmount'}
  }},
  {$sort: {avgOrderValue: -1}}
])
```

9. Write a query to find documents with array elements matching multiple criteria

```
db.products.find({
  categories: {
    $all: ['electronics', 'mobile'],
    $size: 2
  },
  price: {$gt: 500}
})
```

10. Implement a MongoDB update to atomically increment a counter and update a timestamp

```
db.counters.findAndModify({
  query: {_id: 'userId'},
  update: {
    $inc: {seq: 1},
    $set: {lastUpdated: new Date()}
  },
  upsert: true
})
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How would you implement a basic LRU cache using MongoDB collections?

Implementation Approach:

An LRU cache in MongoDB can be implemented using a collection with TTL index:

```
db.createCollection('lruCache')
db.lruCache.createIndex({ lastAccessed: 1 }, { expireAfterSeconds: 3600 })
db.lruCache.createIndex({ key: 1 }, { unique: true })
```

Key aspects:

- Use TTL index for automatic expiration
- Update lastAccessed timestamp on each read
- Maintain size limit using maxSize parameter

2. Design a MongoDB schema for implementing a hierarchical category tree structure

Solution:

```
const categorySchema = {
  _id: ObjectId,
  name: String,
  parentId: ObjectId,
  path: Array,
  level: Number
}
```

Key features:

- path array stores ancestors for efficient traversal
- level field enables quick depth queries
- parentId enables direct parent access

Time complexity: O(1) for retrievals, O(log n) for updates

3. How would you implement a sliding window analysis on time-series data in MongoDB?

Implementation:

```
db.timeSeries.aggregate([
  { $match: { timestamp: { $gte: windowStart, $lte: windowEnd } } },
  { $bucket: { groupBy: '$timestamp', boundaries: timeWindows } },
  { $project: { avgValue: { $avg: '$value' } } }
])
```

Performance considerations:

- Index on timestamp field
- Choose appropriate window size
- Use proper date arithmetic

4. Implement a MongoDB-based rate limiter using atomic operations

Implementation:

```
db.rateLimits.updateOne(
  { userId: id, window: currentWindow },
  { $inc: { count: 1 } },
  { upsert: true, returnDocument: 'after' }
)
```

Key components:

- Atomic increment operation
- TTL index for automatic cleanup
- Window-based counting

Time complexity: O(1) for both checks and updates

5. Design a schema for implementing a graph database structure in MongoDB

Schema Design:

```
const nodeSchema = {
  _id: ObjectId,
  data: Object,
  edges: [{ nodeId: ObjectId, weight: Number }]
}
```

Advantages:

- Efficient traversal with embedded edges
- Supports weighted edges
- Easy to implement graph algorithms

Best for: Small to medium graphs with frequent traversals

6. Implement a distributed counter using MongoDB

Implementation:

```
db.counters.findAndModify({
  query: { _id: counterId },
  update: { $inc: { sequence: incrementBy } },
  new: true, upsert: true
})
```

Features:

- Atomic operations for consistency
- Supports sharding
- Handles concurrent updates

Performance: O(1) for increments

7. Design a schema for implementing a priority queue in MongoDB

Schema Design:

```
const queueSchema = {
  _id: ObjectId,
  priority: Number,
  payload: Object,
  addedAt: Date
}
```

Operations:

- Compound index on (priority, addedAt)
- findAndModify for atomic dequeue
- Bulk operations for batch processing

Time complexity: O(log n) for enqueue/dequeue

8. Implement a MongoDB-based session store with automatic cleanup

Implementation:

```
db.sessions.createIndex({ expireAt: 1 }, { expireAfterSeconds: 0 })
db.sessions.insertOne({
  sessionId: id,
  data: sessionData,
  expireAt: new Date(Date.now() + ttl)
})
```

Features:

- TTL index for automatic cleanup
- Configurable session duration
- Atomic updates for session data

9. Design a schema for implementing a leaderboard system in MongoDB

Schema Design:

```
const scoreSchema = {
  _id: ObjectId,
  userId: ObjectId,
  score: Number,
  timestamp: Date
}
```

Key operations:

- Compound index on (score, timestamp)
- Use $slice for top-N queries
- Periodic aggregation for performance

Time complexity: O(log n) for updates

10. Implement a document versioning system using MongoDB

Implementation:

```
const versionSchema = {
  _id: ObjectId,
  documentId: ObjectId,
  version: Number,
  data: Object,
  timestamp: Date
}
```

Features:

- Immutable version history
- Point-in-time recovery
- Efficient latest version queries

Space complexity: O(n) where n is version count

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service using MongoDB

Key Components:

- **Data Model:** Collection storing original URL, short code, creation date, click metrics
- **Sharding Strategy:** Hash-based sharding on short code
- **Caching Layer:** Redis for frequently accessed URLs

Sample Schema:

{ shortCode: String (indexed), originalUrl: String, createdAt: Date, clicks: Number, userId: ObjectId }

Architecture:

- Load balancer distributing requests
- Application servers for URL generation/redirection
- MongoDB cluster with replica sets
- Counter collection for sequential IDs

2. How would you design a social media feed system with MongoDB?

Core Components:

- **Collections:** Posts, Users, Followers, Interactions
- **Feed Generation:** Fan-out on write vs fan-out on read
- **Caching Strategy:** Cache hot users' feeds

Post Schema:

{ _id: ObjectId, userId: ObjectId, content: String, media: [String], likes: Number, comments: [{   userId: ObjectId,   text: String }]}

Performance Optimizations:

- Denormalization for frequent reads
- Time-based sharding
- Materialized views for trending posts

3. Design a real-time chat application using MongoDB

Architecture Components:

- **WebSocket Server:** For real-time message delivery
- **MongoDB Change Streams:** For message notifications
- **Message Queue:** For offline message handling

Message Schema:

{ chatId: ObjectId, senderId: ObjectId, content: String, timestamp: Date, status: String, readBy: [ObjectId]}

Optimization Strategies:

- Time-based message archival
- Separate collections for active vs archived chats
- Indexing on chatId and timestamp

4. Design a distributed rate limiter using MongoDB

Implementation Approach:

- **Token Bucket Algorithm** implementation
- **Distributed Counter** using atomic operations
- **TTL Index** for automatic cleanup

Rate Limit Document:

{ key: String, tokens: Number, lastRefill: Date, expiresAt: Date}

Key Features:

- Atomic findAndModify operations
- Sliding window implementation
- Cluster-wide synchronization
- Auto-scaling support

5. How would you implement a job queue system with MongoDB?

Core Components:

- **Job Collection:** Stores job definitions and states
- **Worker Process:** Claims and processes jobs
- **Monitoring System:** Tracks job progress

Job Document Schema:

{ _id: ObjectId, type: String, data: Object, status: String, priority: Number, attempts: Number, lockedAt: Date}

Implementation Features:

- Atomic job claiming
- Dead letter queue
- Job retry mechanism
- Priority queuing

6. Design a notification system using MongoDB

System Components:

- **Notification Service:** Handles message generation
- **Delivery Worker:** Processes notifications
- **Template Engine:** Manages notification templates

Notification Schema:

{ userId: ObjectId, type: String, content: Object, status: String, scheduledAt: Date, channels: [String]}

Key Features:

- Multi-channel support
- Batch processing
- Delivery tracking
- Rate limiting per user

7. Design a product catalog system with MongoDB

Data Model Components:

- **Products Collection:** Core product data
- **Categories Collection:** Hierarchical categories
- **Inventory Collection:** Stock management

Product Schema:

{ _id: ObjectId, sku: String, name: String, price: Decimal128, attributes: Object, categories: [ObjectId]}

Optimization Strategies:

- Denormalization for category paths
- Compound indexes for search
- Caching for popular products

8. How would you implement a leaderboard system using MongoDB?

Implementation Approaches:

- **Sorted Set:** For real-time ranking
- **Batch Updates:** For periodic recalculation
- **Materialized Views:** For complex rankings

Score Document Schema:

{ userId: ObjectId, score: Number, rank: Number, timestamp: Date, category: String}

Performance Features:

- Compound indexes for efficient querying
- Time-based partitioning
- Caching of top N rankings
- Incremental updates

9. Design a content management system using MongoDB

System Components:

- **Content Repository:** Versioned documents
- **Asset Management:** Media files handling
- **Workflow Engine:** Content lifecycle

Content Schema:

{ _id: ObjectId, type: String, title: String, content: Object, version: Number, status: String, metadata: Object}

Key Features:

- Version control
- Content scheduling
- Hierarchical categories
- Full-text search integration

10. Design a distributed session management system using MongoDB

Architecture Components:

- **Session Store:** MongoDB collection
- **Cache Layer:** Redis for active sessions
- **Cleanup Worker:** Expired session removal

Session Schema:

{ sessionId: String, userId: ObjectId, data: Object, createdAt: Date, expiresAt: Date, lastAccessed: Date}

Implementation Features:

- TTL indexes for auto-expiration
- Atomic updates for session data
- Session clustering
- Security features (encryption)

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Implement a MongoDB aggregation pipeline to calculate running totals of transactions per user, grouped by month.

Solution:

```
db.transactions.aggregate([
  { $group: {
    _id: { userId: '$userId', month: { $month: '$date' } },
    total: { $sum: '$amount' }
  }}
])
```

Components:

- Date aggregation operators
- Grouping by multiple fields
- Running total calculation

2. Write a MongoDB query to find all documents where the 'age' field is between 25 and 35, sorted by name ascending.

Solution:

```
db.users.find({ age: { $gte: 25, $lte: 35 } }).sort({ name: 1 })
```

Key points:

- $gte and $lte for range queries
- sort() with 1 for ascending order
- Compound query using multiple conditions

3. Create a MongoDB aggregation pipeline to group users by country and calculate the average age for each country, showing only countries with more than 100 users.

Solution:

```
db.users.aggregate([
  { $group: { _id: '$country', avgAge: { $avg: '$age' }, count: { $sum: 1 } } },
  { $match: { count: { $gt: 100 } } },
  { $sort: { avgAge: -1 } }
])
```

Explanation:

- $group to aggregate by country
- $avg operator for mean calculation
- $match to filter results
- $sort for ordering

4. Write a MongoDB update query to increment the 'views' field by 1 and add a timestamp to 'lastViewed' array, ensuring the array doesn't exceed 5 elements.

Solution:

```
db.articles.update(
  { _id: articleId },
  { $inc: { views: 1 },
    $push: { lastViewed: { $each: [new Date()], $slice: -5 } } }
)
```

Key concepts:

- $inc for atomic increments
- $push with $each and $slice for array management
- Maintains fixed-size array

5. Implement a MongoDB query to find documents with array field 'tags' containing all elements from a given array, handling case-insensitive matching.

Solution:

```
db.posts.find({
  tags: {
    $all: searchTags.map(tag => new RegExp('^' + tag + '$', 'i'))
  }
})
```

Important aspects:

- $all operator for array matching
- RegExp for case-insensitive comparison
- Exact matching with ^ and $

6. Create a MongoDB query to perform a full-text search on 'title' and 'description' fields, with boosted relevance for title matches.

Solution:

```
db.articles.find(
  { $text: { $search: searchTerm } },
  { score: { $meta: 'textScore' } }
).sort({ score: { $meta: 'textScore' } })
```

Prerequisites:

- Text index creation required
- Weights can be specified in index
- Automatic language detection

7. Write a MongoDB update operation to modify nested array elements matching specific criteria while preventing array expansion.

Solution:

```
db.orders.update(
  { 'items.productId': productId },
  { $set: { 'items.$.quantity': newQuantity } },
  { arrayFilters: [{ 'item.productId': productId }] }
)
```

Key features:

- Positional $ operator
- arrayFilters for nested arrays
- Atomic update operation

8. Create a MongoDB query to find documents where a field exists but is not null, empty array, or empty object.

Solution:

```
db.collection.find({
  field: { $exists: true, $ne: null, $not: { $size: 0 } },
  $where: 'Object.keys(this.field).length > 0'
})
```

Validation checks:

- Existence check
- Null check
- Empty array/object check

9. Write a MongoDB update operation to atomically push elements to an array only if they don't already exist.

Solution:

```
db.collection.update(
  { _id: documentId },
  { $addToSet: { tags: { $each: newTags } } }
)
```

Features:

- $addToSet for uniqueness
- $each for multiple values
- Atomic operation guarantee

10. Implement a MongoDB query to find documents with geometric locations within a specified radius, sorted by distance.

Solution:

```
db.places.find({
  location: { $nearSphere: { $geometry: { type: 'Point', coordinates: [lng, lat] }, $maxDistance: 5000 } }
})
```

Requirements:

- 2dsphere index on location field
- GeoJSON format for coordinates
- Distance in meters