MongoDB Associate Developer (Node.js) Study Guide

Table of Contents

1. MongoDB Overview and Document Model

1.1 BSON Value Types

MongoDB stores data in BSON (Binary JSON) format which supports more data types than JSON.

BSON Supported Types:

- String: `"Hello"`
- Integer: `42`
- Double: `3.14159`
- Boolean: `true` or `false`
- Array: `["item1", "item2"]`
- Object/Document: `{ key: value }`
- ObjectId: `ObjectId("507f1f77bcf86cd799439011")`
- Date: `ISODate("2023-04-01T12:00:00Z")`
- Null: `null`
- Binary data: For storing binary data
- Regular Expression: `/pattern/`
- Timestamp: For internal MongoDB use
- Decimal128: For high-precision decimal values
- JavaScript code: For stored functions
- MinKey/MaxKey: Special types for comparisons

Example in Node.js:

```javascript
const document = {
  name: "John Doe",                                    // String
  age: 30,                                             // Integer
  score: 85.5,                                         // Double
  isActive: true,                                      // Boolean
  tags: ["developer", "mongodb", "javascript"],        // Array
  address: {                                           // Embedded document
    street: "123 Main St",
    city: "San Francisco"
  },
  userId: new ObjectId(),                              // ObjectId
  createdAt: new Date(),                               // Date
  profile: null,                                       // Null
  data: Buffer.from("binary data"),                    // Binary
  searchPattern: /^test/i,                             // Regular Expression
  balance: new Decimal128("123.45"),                   // Decimal128
  lastModified: new Timestamp()                        // Timestamp
};
```

1.2 Document Structure and Collections

MongoDB is schema-less, meaning documents in the same collection can have different structures (fields, types).

Key Concepts:

- Documents with different "shapes" can exist in the same collection
- Document size limit is 16MB
- Field names are case-sensitive
- Field names cannot contain null characters
- The `_id` field is reserved for the primary key

Example of Different Document Shapes in Same Collection:

```javascript
// All three can exist in the same collection
const document1 = {
  _id: ObjectId("507f1f77bcf86cd799439011"),
  name: "John",
  age: 30
};

const document2 = {
  _id: ObjectId("507f1f77bcf86cd799439012"),
  firstName: "Jane",
  lastName: "Smith",
  contact: {
    email: "jane@example.com",
    phone: "123-456-7890"
  }
};

const document3 = {
  _id: ObjectId("507f1f77bcf86cd799439013"),
  items: ["book", "pen", "paper"]
};
```

2. CRUD Operations

2.1 Create (Insert) Operations

Inserting a Single Document:

```javascript
// Insert one document
db.collection('users').insertOne({
  name: "John Doe",
  email: "john@example.com",
  age: 30
});

// With error handling
try {
  const result = await db.collection('users').insertOne({
    name: "John Doe",
    email: "john@example.com",
    age: 30
  });
  console.log(`Document inserted with _id: ${result.insertedId}`);
} catch (error) {
  console.error("Error inserting document:", error);
}
```

Inserting Multiple Documents:

```javascript
// Insert many documents
db.collection('users').insertMany([
  { name: "John", age: 30 },
  { name: "Jane", age: 25 },
  { name: "Bob", age: 35 }
]);

// With options
db.collection('users').insertMany(
  [
    { name: "John", age: 30 },
    { name: "Jane", age: 25 }
  ],
  { ordered: false } // Continue processing even if one insert fails
);
```

2.2 Read (Query) Operations

Finding a Single Document:

```javascript
// Find one document
const user = await db.collection('users').findOne({ name: "John" });

// Find one with projection (include only specific fields)
const userDetails = await db.collection('users').findOne(
  { name: "John" },
  { projection: { name: 1, email: 1, _id: 0 } }
);
```

Finding Multiple Documents:

```javascript
// Find all documents matching criteria
const cursor = db.collection('users').find({ age: { $gt: 25 } });
const users = await cursor.toArray();

// With sort, limit, and skip
const cursor = db.collection('users')
  .find({ age: { $gt: 25 } })
  .sort({ name: 1 }) // Sort by name ascending
  .skip(10)          // Skip first 10 results
  .limit(5);         // Limit to 5 results
```

Query Operators:

1. Comparison Operators:

javascript

```javascript
// Greater than
db.collection('users').find({ age: { $gt: 25 } });

// Less than
db.collection('users').find({ age: { $lt: 30 } });

// Greater than or equal
db.collection('users').find({ age: { $gte: 25 } });

// Less than or equal
db.collection('users').find({ age: { $lte: 30 } });

// Equal
db.collection('users').find({ age: { $eq: 30 } });

// Not equal
db.collection('users').find({ age: { $ne: 30 } });

// In array
db.collection('users').find({ age: { $in: [20, 25, 30] } });

// Not in array
db.collection('users').find({ age: { $nin: [20, 25, 30] } });
```

2. Logical Operators:

javascript

```javascript
// AND - both conditions must be true
db.collection('users').find({
  $and: [{ age: { $gt: 25 } }, { status: "active" }]
});

// OR - either condition can be true
db.collection('users').find({
  $or: [{ age: { $gt: 30 } }, { status: "premium" }]
});

// NOT - negate the condition
db.collection('users').find({
  age: { $not: { $lt: 25 } }
});

// NOR - none of the conditions can be true
db.collection('users').find({
  $nor: [{ age: { $lt: 20 } }, { status: "inactive" }]
});
```

3. Element Operators:

javascript

```javascript
// Field exists
db.collection('users').find({ email: { $exists: true } });

// Field type is string
db.collection('users').find({ name: { $type: "string" } });
```

4. Array Operators:

```javascript
// Array contains element
db.collection('users').find({ tags: "mongodb" });

// Array contains all elements
db.collection('users').find({ tags: { $all: ["mongodb", "nodejs"] } });

// Element matching multiple criteria
db.collection('users').find({
  scores: { $elemMatch: { $gt: 80, $lt: 90 } }
});

// Array size
db.collection('users').find({ tags: { $size: 3 } });
```

2.3 Update Operations

Update a Single Document:

```javascript
// Update one document
await db.collection('users').updateOne(
  { name: "John" },
  { $set: { age: 31, status: "active" } }
);

// Increment a field
await db.collection('users').updateOne(
  { name: "John" },
  { $inc: { age: 1 } }
);
```

Update Multiple Documents:

```javascript
// Update many documents
await db.collection('users').updateMany(
  { age: { $lt: 30 } },
  { $set: { category: "young" } }
);
```

Replace Entire Document:

```javascript
// Replace the entire document (except _id)
await db.collection('users').replaceOne(
  { name: "John" },
  { name: "John Doe", age: 31, email: "john@example.com" }
);
```

Upsert (Update or Insert):

```javascript
// Upsert example (create if doesn't exist)
await db.collection('users').updateOne(
  { name: "Alice" },
  { $set: { age: 25, email: "alice@example.com" } },
  { upsert: true }
);
```

Update Operators:

javascript

```
// $set - set field values
db.collection('users').updateOne(
  { name: "John" },
  { $set: { age: 31, status: "active" } }
);

// $inc - increment field values
db.collection('users').updateOne(
  { name: "John" },
  { $inc: { age: 1, loginCount: 1 } }
);

// $mul - multiply field values
db.collection('products').updateOne(
  { name: "Widget" },
  { $mul: { price: 1.1 } } // Increase price by 10%
);

// $rename - rename fields
db.collection('users').updateMany(
  {},
  { $rename: { "phone": "phoneNumber" } }
);

// $unset - remove fields
db.collection('users').updateOne(
  { name: "John" },
  { $unset: { temporary: "" } }
);

// Array operators
db.collection('users').updateOne(
  { name: "John" },
  { $push: { tags: "mongodb" } } // Add to array
);

db.collection('users').updateOne(
  { name: "John" },
  { $pull: { tags: "beginner" } } // Remove from array
);

db.collection('users').updateOne(
  { name: "John" },
  { $addToSet: { tags: "mongodb" } } // Add if not exists
);
```

FindAndModify Operations:

The `findOneAndUpdate`, `findOneAndReplace`, and `findOneAndDelete` methods allow you to atomically update or delete a document and return either the original document or the updated document.

```javascript
// Find document and update it, returning the updated document
const result = await db.collection('users').findOneAndUpdate(
  { name: "John" },
  { $set: { status: "active" } },
  { returnDocument: "after" } // Return the updated document
);

// Find document and replace it, returning the original document
const result = await db.collection('users').findOneAndReplace(
  { name: "John" },
  { name: "John", age: 31, status: "active" },
  { returnDocument: "before" } // Return the original document
);
```

2.4 Delete Operations

Delete a Single Document:

```javascript
// Delete one document
await db.collection('users').deleteOne({ name: "John" });
```

Delete Multiple Documents:

```javascript
// Delete many documents
await db.collection('users').deleteMany({ age: { $lt: 18 } });
```

Find and Delete:

```javascript
// Find a document and delete it, returning the deleted document
const deletedUser = await db.collection('users').findOneAndDelete({ name: "John" });
```

2.5 Bulk Operations

Bulk operations allow you to perform multiple operations in a single database request:

```javascript
// Initialize a bulk operation
const bulk = db.collection('users').initializeUnorderedBulkOp();

// Add operations to the bulk
bulk.find({ status: "inactive" }).update({ $set: { status: "archived" } });
bulk.find({ age: { $lt: 18 } }).remove();
bulk.insert({ name: "New User", age: 25 });

// Execute the bulk operation
const result = await bulk.execute();
```

2.6 Search Index and Search Queries

MongoDB Atlas provides powerful text search capabilities using search indexes:

Creating a Search Index:

javascript

```javascript
// Create a search index
await db.collection('products').createIndex(
  { description: "text", name: "text" },
  { weights: { name: 10, description: 5 } }
);

// More powerful search index (MongoDB Atlas)
db.runCommand({
  createSearchIndexes: "products",
  indexes: [
    {
      name: "product_search",
      definition: {
        mappings: {
          dynamic: false,
          fields: {
            name: { type: "string" },
            description: { type: "string" },
            category: { type: "string" }
          }
        }
      }
    }
  ]
});
```

Performing a Search:

```javascript
// Basic text search
const results = await db.collection('products').find(
  { $text: { $search: "wireless headphones" } },
  { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } });

// Advanced search (MongoDB Atlas)
const results = await db.collection('products').aggregate([
  {
    $search: {
      index: "product_search",
      text: {
        query: "wireless headphones",
        path: ["name", "description"]
      }
    }
  },
  { $limit: 10 }
]);
```

2.7 Aggregation Framework

The Aggregation Framework is used for data processing and analysis:

Basic Aggregation Pipeline:

```javascript
// Simple aggregation with $match and $group
const results = await db.collection('orders').aggregate([
  // Stage 1: Filter documents
  { $match: { status: "completed" } },

  // Stage 2: Group documents and calculate totals
  { $group: {
      _id: "$customerId",
      totalSpent: { $sum: "$amount" },
      count: { $sum: 1 }
    }
  },

  // Stage 3: Sort by total spent
  { $sort: { totalSpent: -1 } }
]).toArray();
```

Common Aggregation Operators:

1. `$match`: Filter documents

```javascript
{ $match: { age: { $gt: 30 } } }
```

2. `$group`: Group documents by key

```javascript
{ $group: {
    _id: "$country",
    totalUsers: { $sum: 1 },
    avgAge: { $avg: "$age" }
  }
}
```

3. `$project`: Reshape documents

```javascript
{ $project: {
    _id: 0,
    fullName: { $concat: ["$firstName", " ", "$lastName"] },
    age: 1
  }
}
```

4. `$sort`: Sort documents

```javascript
{ $sort: { age: -1 } } // Descending order
```

5. `$limit`: Limit results

```javascript
{ $limit: 5 }
```

6. `$skip`: Skip results

```javascript
{ $skip: 10 }
```

7. `$lookup`: Join with another collection

```javascript
{ $lookup: {
    from: "orders",
    localField: "_id",
    foreignField: "customerId",
    as: "customerOrders"
  }
}
```

8. `$unwind`: Deconstruct array field

```javascript
{ $unwind: "$tags" }
```

9. `$out`: Output to a collection

```javascript
{ $out: "aggregationResults" }
```

Example with $lookup:

```javascript
// Join customers with their orders
const results = await db.collection('customers').aggregate([
  { $match: { status: "active" } },
  { $lookup: {
      from: "orders",
      localField: "_id",
      foreignField: "customerId",
      as: "orders"
    }
  },
  { $project: {
      _id: 1,
      name: 1,
      email: 1,
      orderCount: { $size: "$orders" },
      totalSpent: { $sum: "$orders.amount" }
    }
  }
]).toArray();
```

3. Indexes

3.1 Index Types

Creating a Single Field Index:

```javascript
// Create an ascending index on name
await db.collection('users').createIndex({ name: 1 });

// Create a descending index on age
await db.collection('users').createIndex({ age: -1 });
```

Creating a Compound Index:

```javascript
// Create a compound index on name (ascending) and age (descending)
await db.collection('users').createIndex({ name: 1, age: -1 });
```

Index Types:

1. **Single Field Index**

javascript

```javascript
await db.collection('users').createIndex({ email: 1 });
```

2. **Compound Index**

javascript

```javascript
await db.collection('users').createIndex({ lastName: 1, firstName: 1 });
```

3. **Multikey Index** (automatically created for array fields)

javascript

```javascript
await db.collection('blog').createIndex({ tags: 1 });
```

4. **Text Index**

javascript

```javascript
await db.collection('articles').createIndex({ content: "text", title: "text" });
```

5. **Hashed Index**

javascript

```javascript
await db.collection('users').createIndex({ _id: "hashed" });
```

6. **Geospatial Index**

javascript

```javascript
// 2dsphere index for GeoJSON data
await db.collection('places').createIndex({ location: "2dsphere" });
```

7. **TTL Index** (documents expire after a specified time)

javascript

```javascript
// Documents will be automatically removed 3600 seconds after createdAt
await db.collection('sessions').createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 3600 }
);
```

3.2 Index Options

```javascript
// Create an index with options
await db.collection('users').createIndex(
  { email: 1 },
  {
    unique: true,                // Enforce unique values
    sparse: true,                // Only index docs with the field
    background: true,            // Build in the background
    name: "email_unique_idx",    // Custom name
    partialFilterExpression: {  // Only index certain documents
      status: "active"
    }
  }
);
```

3.3 Index Management

Listing Indexes:

```javascript
// List all indexes on a collection
const indexes = await db.collection('users').indexes();
console.log(indexes);
```

Dropping an Index:

```javascript
// Drop a specific index
await db.collection('users').dropIndex("email_1");

// Drop all indexes (except _id)
await db.collection('users').dropIndexes();
```

3.4 Using Explain Plans

The `explain()` method shows how MongoDB executes a query:

```javascript
// Get execution plan
const explainResult = await db.collection('users')
  .find({ age: { $gt: 25 } })
  .explain("executionStats");

// Check what index was used (if any)
console.log(explainResult.queryPlanner.winningPlan.inputStage);

// Check execution statistics
console.log(explainResult.executionStats);
```

Key Explain Output Fields to Check:

1. **winningPlan.stage**: Look for:
   - `COLLSCAN` (Collection Scan) - No index used, full collection scan (bad for performance)
   - `IXSCAN` (Index Scan) - Using an index (good)
   - `FETCH` - Retrieving documents using an index
   - `SORT` - In-memory sort (can be expensive)
2. **executionStats**:
   - `nReturned`: Number of documents returned
   - `totalKeysExamined`: Number of index keys examined
   - `totalDocsExamined`: Number of documents examined
   - `executionTimeMillis`: Execution time

Example Analysis:
If `totalDocsExamined` is much higher than `nReturned`, the query is inefficient and may benefit from an index.

4. Data Modeling

4.1 Data Modeling Patterns

Embedded Documents Pattern:

```javascript
// User document with embedded address
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  name: "John Doe",
  email: "john@example.com",
  address: {
    street: "123 Main St",
    city: "San Francisco",
    state: "CA",
    zip: "94107"
  }
}
```

Referenced Documents Pattern:

```javascript
// User document with reference to orders
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  name: "John Doe",
  email: "john@example.com"
}

// Order documents with reference to user
{
  _id: ObjectId("507f1f77bcf86cd799439012"),
  userId: ObjectId("507f1f77bcf86cd799439011"),
  product: "MongoDB Course",
  amount: 199.99
}
```

4.2 When to Embed vs. Reference

Embed When:

- One-to-few relationships
- Data is always loaded together
- Data doesn't change frequently
- Embedded data is small
- Data forms a logical unit

Reference When:

- One-to-many or many-to-many relationships
- Data is accessed separately
- Data changes frequently
- Referenced data is large
- Data needs to be shared across documents

4.3 Common Anti-Patterns

1. **Massive Arrays:**

```javascript
// Anti-pattern: Unbounded array that could grow very large
{
  _id: ObjectId(),
  name: "Popular Post",
  comments: [
    // Thousands of comments could make this document exceed 16MB limit
    { user: "user1", text: "Great post!" },
    { user: "user2", text: "Thanks for sharing!" },
    // ...many more comments
  ]
}
```

2. **Deeply Nested Documents:**

```javascript
// Anti-pattern: Too many levels of nesting
{
  _id: ObjectId(),
  level1: {
    level2: {
      level3: {
        level4: {
          // More nesting makes queries complex
        }
      }
    }
  }
}
```

3. **Storing Too Much in a Single Document:**

```javascript
// Anti-pattern: Document trying to do too much
{
  _id: ObjectId(),
  user: {
    // User data
  },
  orders: [
    // Order history
  ],
  cart: {
    // Shopping cart
  },
  wishlist: [
    // Wishlist items
  ],
  // And more unrelated data
}
```

4. **Not Using Proper Data Types:**

```javascript
// Anti-pattern: Using strings for dates or numbers
{
  _id: ObjectId(),
  price: "19.99",      // Should be a number
  createdAt: "2023-04-01"  // Should be a Date object
}
```

5. Tools and Tooling

5.1 MongoDB Atlas

MongoDB Atlas is the cloud-based database service that includes:

- Automated backups
- Scaling
- Performance monitoring
- Security features

Connecting to Atlas from Node.js:

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

// Atlas connection string (replace with your own)
const uri = "mongodb+srv://username:password@cluster0.mongodb.net/mydb?retryWrites=tru

const client = new MongoClient(uri);

async function run() {
  try {
    await client.connect();
    console.log("Connected to MongoDB Atlas!");

    // Access your database
    const database = client.db("sample_mflix");
    const collection = database.collection("movies");

    // Query for a movie
    const movie = await collection.findOne({ title: "The Godfather" });
    console.log(movie);

  } finally {
    await client.close();
  }
}

run().catch(console.dir);
```

5.2 MongoDB Compass

MongoDB Compass is the GUI for MongoDB that allows you to:

- Explore your data visually
- Run queries with an intuitive interface
- Create and analyze indexes
- Optimize performance
- Create and test aggregation pipelines

5.3 MongoDB Shell (mongosh)

MongoDB Shell is the command-line interface for interacting with MongoDB:

bash

```
# Connect to local MongoDB
mongosh

# Connect to Atlas
mongosh "mongodb+srv://username:password@cluster0.mongodb.net/mydb"

# Basic commands
show dbs                 # List databases
use mydb                 # Switch to a database
show collections         # List collections
db.users.find()          # Query a collection
db.users.createIndex()   # Create an index
```

6. Node.js Driver

6.1 Node.js Driver Overview

The official MongoDB Node.js driver provides a high-level API for interacting with MongoDB from Node.js applications.

Installing the Driver:

bash

```
npm install mongodb
```

6.2 Connecting to MongoDB

Connection String URI Components:

```
mongodb+srv://username:password@hostname/database?options
```

- **Protocol**: `mongodb://` or `mongodb+srv://` (DNS seed list for replica sets)
- **Username & Password**: Authentication credentials
- **Hostname**: Server address (or Atlas cluster)
- **Database**: Optional default database
- **Options**: Connection options as query parameters

Basic Connection:

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

// Connection URL
const url = 'mongodb://localhost:27017';

// Database Name
const dbName = 'myproject';

// Create a new MongoClient
const client = new MongoClient(url);

// Connect to the server
async function connect() {
  try {
    await client.connect();
    console.log('Connected successfully to MongoDB');

    const db = client.db(dbName);
    return db;
  } catch (err) {
    console.error('Error connecting to MongoDB:', err);
    throw err;
  }
}
```

Connection with Options:

```javascript
const client = new MongoClient(url, {
  connectTimeoutMS: 5000,          // Connection timeout
  socketTimeoutMS: 30000,          // Socket timeout
  maxPoolSize: 50,                 // Max connections in pool
  minPoolSize: 5,                  // Min connections in pool
  retryWrites: true,               // Retry write operations
  retryReads: true,                // Retry read operations
  w: 'majority',                   // Write concern
  readPreference: 'primaryPreferred', // Read preference
  useUnifiedTopology: true         // Use modern topology engine
});
```

6.3 Connection Pooling

Connection pooling reuses connections to improve performance:

```javascript
const client = new MongoClient(url, {
  maxPoolSize: 50,   // Maximum number of connections in the pool
  minPoolSize: 5,    // Minimum number of connections in the pool
  maxIdleTimeMS: 30000 // Close idle connections after 30 seconds
});
```

Advantages of Connection Pooling:

- Reduces connection overhead
- Improves performance
- Manages connection lifecycle
- Distributes load across connections

6.4 CRUD Operations with the Node.js Driver

Creating Documents:

```javascript
// Insert one document
async function insertOne(db) {
  const collection = db.collection('users');
  const result = await collection.insertOne({
    name: 'John Doe',
    email: 'john@example.com',
    createdAt: new Date()
  });

  console.log(`Inserted document with _id: ${result.insertedId}`);
  return result;
}

// Insert many documents
async function insertMany(db) {
  const collection = db.collection('users');
  const result = await collection.insertMany([
    { name: 'John', email: 'john@example.com' },
    { name: 'Jane', email: 'jane@example.com' }
  ]);

  console.log(`Inserted ${result.insertedCount} documents`);
  console.log(`Inserted IDs: ${Object.values(result.insertedIds)}`);
  return result;
}
```

Reading Documents:

javascript

```
// Find one document
async function findOne(db) {
  const collection = db.collection('users');
  const user = await collection.findOne({ name: 'John' });

  if (user) {
    console.log(`Found user: ${user.name}`);
  } else {
    console.log('No user found');
  }

  return user;
}

// Find many documents
async function findMany(db) {
  const collection = db.collection('users');

  // Find documents that match criteria
  const cursor = collection.find({ age: { $gt: 25 } });

  // Count documents
  const count = await cursor.count();
  console.log(`Found ${count} users`);

  // Get all results
  const users = await cursor.toArray();
  console.log(users);

  // Iterate through results one by one
  // await cursor.forEach(user => {
  //   console.log(`User: ${user.name}`);
  // });

  return users;
}

// Find with options
async function findWithOptions(db) {
  const collection = db.collection('users');

  const options = {
    sort: { name: 1 },          // Sort by name ascending
    projection: { _id: 0, name: 1, email: 1 }, // Include only name and email
    limit: 10,                  // Limit to 10 results
    skip: 20                    // Skip first 20 results
```

```
  };

  const users = await collection.find({ age: { $gt: 25 } }, options).toArray();
  return users;
}
```

Updating Documents:

javascript

```
// Update one document
async function updateOne(db) {
  const collection = db.collection('users');

  const result = await collection.updateOne(
    { name: 'John' },           // Filter
    { $set: { status: 'active', lastLogin: new Date() } } // Update
  );

  console.log(`Matched ${result.matchedCount} document(s)`);
  console.log(`Modified ${result.modifiedCount} document(s)`);

  return result;
}

// Update many documents
async function updateMany(db) {
  const collection = db.collection('users');

  const result = await collection.updateMany(
    { status: 'inactive' },     // Filter
    { $set: { status: 'archived' } } // Update
  );

  console.log(`Matched ${result.matchedCount} document(s)`);
  console.log(`Modified ${result.modifiedCount} document(s)`);

  return result;
}

// Update with upsert
async function upsert(db) {
  const collection = db.collection('users');

  const result = await collection.updateOne(
    { email: 'bob@example.com' },    // Filter
    { $set: { name: 'Bob', age: 40 } },  // Update
    { upsert: true }                 // Create if doesn't exist
  );

  if (result.upsertedCount > 0) {
    console.log(`Document inserted with _id: ${result.upsertedId}`);
  } else {
    console.log(`Matched ${result.matchedCount} document(s)`);
    console.log(`Modified ${result.modifiedCount} document(s)`);
  }
```

```
  return result;
}

// Find one and update
async function findOneAndUpdate(db) {
  const collection = db.collection('users');

  const result = await collection.findOneAndUpdate(
    { name: 'John' },                          // Filter
    { $set: { status: 'active' } },            // Update
    {
      returnDocument: 'after',                 // Return the updated document
      projection: { name: 1, email: 1, status: 1 }  // Projection
    }
  );

  console.log('Updated user:', result.value);
  return result;
}

/**
 * Deleting Documents
 */

// Delete one document
async function deleteOne(db) {
  const collection = db.collection('users');

  const result = await collection.deleteOne({ name: 'John' });

  console.log(`Deleted ${result.deletedCount} document(s)`);
  return result;
}

// Delete many documents
async function deleteMany(db) {
  const collection = db.collection('users');

  const result = await collection.deleteMany({ status: 'inactive' });

  console.log(`Deleted ${result.deletedCount} document(s)`);
  return result;
}

// Find one and delete
async function findOneAndDelete(db) {
```

```
  const collection = db.collection('users');

  const result = await collection.findOneAndDelete(
    { name: 'John' },
    { projection: { name: 1, email: 1 } }  // Only return these fields
  );

  if (result.value) {
    console.log(`Deleted user: ${result.value.name}`);
  } else {
    console.log('No user found to delete');
  }

  return result;
}
```

6.5 Aggregation Pipeline

The Node.js driver provides full support for MongoDB's aggregation framework:

```javascript
async function runAggregation(db) {
  const collection = db.collection('orders');

  const pipeline = [
    // Stage 1: Filter documents
    { $match: { status: 'completed' } },

    // Stage 2: Group by customer
    { $group: {
        _id: '$customerId',
        totalSpent: { $sum: '$amount' },
        count: { $sum: 1 },
        averageOrder: { $avg: '$amount' }
      }
    },

    // Stage 3: Look up customer details
    { $lookup: {
        from: 'customers',
        localField: '_id',
        foreignField: '_id',
        as: 'customerDetails'
      }
    },

    // Stage 4: Unwind the customer details array
```

```
    { $unwind: '$customerDetails' },

    // Stage 5: Project the final shape
    { $project: {
        _id: 0,
        customerId: '$_id',
        name: '$customerDetails.name',
        email: '$customerDetails.email',
        totalSpent: 1,
        orderCount: '$count',
        averageOrder: 1
      }
    },

    // Stage 6: Sort by total spent
    { $sort: { totalSpent: -1 } },

    // Stage 7: Limit to top 10
    { $limit: 10 }
  ];

  const results = await collection.aggregate(pipeline).toArray();
  console.log(`Found ${results.length} results`);
  return results;
}

// Using $out to save results to a collection
async function aggregateWithOut(db) {
  const collection = db.collection('orders');

  const pipeline = [
    { $match: { status: 'completed' } },
    { $group: {
        _id: '$customerId',
        totalSpent: { $sum: '$amount' },
        count: { $sum: 1 }
      }
    },
    // Save results to a new collection
    { $out: 'customerSummaries' }
  ];

  // Run the aggregation - results saved to collection
  await collection.aggregate(pipeline).toArray();

  // Now we can query the new collection
  const results = await db.collection('customerSummaries').find().toArray();
```

```
  console.log(`Generated ${results.length} customer summaries`);
  return results;
}
```

6.6 Working with Indexes

```javascript
// Create a single field index
async function createIndex(db) {
  const collection = db.collection('users');

  const result = await collection.createIndex(
    { email: 1 },  // Field and direction
    { unique: true }  // Options
  );

  console.log(`Index created: ${result}`);
  return result;
}

// Create a compound index
async function createCompoundIndex(db) {
  const collection = db.collection('users');

  const result = await collection.createIndex(
    { lastName: 1, firstName: 1 }  // Compound index
  );

  console.log(`Index created: ${result}`);
  return result;
}

// List all indexes
async function listIndexes(db) {
  const collection = db.collection('users');

  const indexes = await collection.indexes();
  console.log('Indexes:', indexes);
  return indexes;
}

// Drop an index
async function dropIndex(db) {
  const collection = db.collection('users');

  const result = await collection.dropIndex('email_1');
  console.log('Index dropped:', result);
  return result;
}
```

6.7 Error Handling

Proper error handling is essential when working with MongoDB:

```javascript
async function safeOperation(db) {
  const collection = db.collection('users');

  try {
    // Attempt to insert a document
    const result = await collection.insertOne({
      name: 'John',
      email: 'john@example.com'
    });

    console.log(`Document inserted with _id: ${result.insertedId}`);
    return result;
  } catch (err) {
    // Handle different types of errors
    if (err.code === 11000) {
      console.error('Duplicate key error - document already exists');
    } else if (err.name === 'MongoNetworkError') {
      console.error('Network error - check connection');
    } else {
      console.error('Error occurred:', err);
    }

    throw err; // Re-throw or handle as appropriate
  }
}
```

6.8 Transactions

MongoDB supports multi-document transactions:

javascript

```
async function runTransaction(client) {
  // Start a session
  const session = client.startSession();

  try {
    // Start transaction
    session.startTransaction();

    // Get the database and collections
    const db = client.db('mydb');
    const accounts = db.collection('accounts');
    const transfers = db.collection('transfers');

    // Perform multiple operations in the transaction
    await accounts.updateOne(
      { accountId: 'A123' },
      { $inc: { balance: -100 } },
      { session }
    );

    await accounts.updateOne(
      { accountId: 'B456' },
      { $inc: { balance: 100 } },
      { session }
    );

    await transfers.insertOne({
      from: 'A123',
      to: 'B456',
      amount: 100,
      date: new Date()
    }, { session });

    // Commit the transaction
    await session.commitTransaction();
    console.log('Transaction successfully committed.');

  } catch (err) {
    // Abort transaction on error
    await session.abortTransaction();
    console.error('Transaction aborted. Error:', err);
    throw err;
  } finally {
    // End the session
    session.endSession();
```

```
  }
}
```

7. Practice Questions

Below are sample practice questions similar to those you might encounter on the MongoDB Associate Developer exam:

7.1 MongoDB Overview and Document Model

Question 1: Which of the following value types are supported by MongoDB BSON? (Select all that apply)

- A. String
- B. Integer
- C. Array
- D. Binary data
- E. Function pointers

Answer: A, B, C, D

Question 2: Given the following documents, which can co-exist in the same MongoDB collection?

Document 1:

```javascript
{
  _id: ObjectId("507f1f77bcf86cd799439011"),
  name: "John",
  age: 30
}
```

Document 2:

```javascript
{
  _id: ObjectId("507f1f77bcf86cd799439012"),
  firstName: "Jane",
  lastName: "Smith"
}
```

Document 3:

```javascript
{
  _id: ObjectId("507f1f77bcf86cd799439013"),
  products: ["Laptop", "Mouse", "Keyboard"]
}
```

- A. Only Document 1 and Document 2
- B. Only Document 1 and Document 3
- C. Only Document 2 and Document 3
- D. All three documents can co-exist

Answer: D (All three documents can co-exist in the same collection as MongoDB is schema-less)

7.2 CRUD Operations

Question 3: Which of the following is the correct way to insert a single document using the Node.js driver?

- A. `db.collection('users').insert({ name: "John" })`
- B. `db.collection('users').insertOne({ name: "John" })`
- C. `db.users.insert({ name: "John" })`
- D. `db.users.add({ name: "John" })`

Answer: B

Question 4: If you execute the following command on a document with `name: "John"`, how will the document change?

```javascript
db.collection('users').updateOne(
  { name: "John" },
  { $set: { status: "active" } }
);
```

- A. The document will be replaced with `{ status: "active" }`
- B. The document will be updated to have a status field with value "active", while keeping the name field
- C. Nothing will change because $set requires all fields to be specified
- D. The command will throw an error

Answer: B

Question 5: Which operator would you use to increment a numeric field in a document?

- A. `$inc`
- B. `$add`
- C. `$increment`
- D. `$plus`

Answer: A

7.3 Indexes

Question 6: Given a query that frequently searches users by their email address, which of the following indexes would be most appropriate?

- A. `db.collection('users').createIndex({ name: 1 })`
- B. `db.collection('users').createIndex({ email: 1 })`
- C. `db.collection('users').createIndex({ _id: 1 })`
- D. `db.collection('users').createIndex({ email: -1 })`

Answer: B or D (direction doesn't matter for a single-key equality query)

Question 7: You have a collection with queries that sort by lastName and firstName. Which index would best support these queries?

- A. `db.collection('users').createIndex({ firstName: 1, lastName: 1 })`
- B. `db.collection('users').createIndex({ lastName: 1, firstName: 1 })`
- C. Two separate indexes: one on firstName and one on lastName
- D. `db.collection('users').createIndex({ lastName: 1 })`

Answer: B

Question 8: In an explain plan, what stage indicates that no index was used for a query?

- A. `IXSCAN`
- B. `COLLSCAN`
- C. `FETCH`
- D. `SORT`

Answer: B

7.4 Data Modeling

Question 9: Which of the following is typically a good candidate for embedding rather than referencing?

- A. A user's addresses when there could be hundreds of addresses
- B. A blog post's comments when there could be thousands of comments
- C. A user's profile information where they have exactly one profile
- D. Products and their categories where products can belong to multiple categories

Answer: C

Question 10: Which of the following is considered an anti-pattern in MongoDB data modeling?

- A. Embedding one-to-one relationships
- B. Using references for one-to-many relationships
- C. Storing unbounded arrays that could grow very large in a single document
- D. Using compound indexes for frequently used queries

Answer: C

7.5 Drivers

Question 11: What is the correct syntax to connect to MongoDB using the Node.js driver?

- A. `MongoClient.connect(url, function(err, client) { ... })`
- B. `new MongoClient(url).connect()`
- C. `const client = new MongoClient(url); await client.connect()`
- D. `MongoDB.connect(url)`

Answer: C

Question 12: Which part of the MongoDB connection string specifies authentication credentials?

- A. `mongodb+srv://username:password@hostname/database`
- B. `mongodb://hostname/database:username:password`
- C. `mongodb://authentication=username,password@hostname/database`
- D. `mongodb://hostname/database?username=user&password=pass`

Answer: A

Question 13: What is the advantage of connection pooling in the MongoDB Node.js driver?

- A. It automatically encrypts all data
- B. It reduces connection overhead by reusing connections

- C. It provides automatic failover between servers
- D. It compresses all data before sending it to the server

Answer: B

8. Additional Resources

8.1 Documentation

- MongoDB Documentation
- Node.js MongoDB Driver Documentation
- MongoDB University

8.2 Recommended Books

- "MongoDB: The Definitive Guide" by Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow
- "MongoDB in Action" by Kyle Banker, Peter Bakkum, Shaun Verch, Douglas Garrett, and Tim Hawkins

8.3 Online Tools

- MongoDB Atlas - Cloud database service
- MongoDB Compass - GUI for MongoDB
- MongoDB Playground - Online sandbox for testing queries

8.4 Exam Preparation Tips

1. **Practice hands-on**: Create a free MongoDB Atlas cluster and practice all operations
2. **Study all sections**: Make sure to cover all exam domains proportional to their weights
3. **Take practice tests**: Use the official practice questions from MongoDB University
4. **Read documentation**: Especially for areas where you feel less confident
5. **Build a project**: Create a small project using MongoDB and Node.js to apply what you've learned
6. **Join the community**: Participate in the MongoDB community forums to learn from others

Good luck with your MongoDB Associate Developer certification exam!