# Comprehensive MongoDB Query Guide

## Table of Contents



## Basic CRUD Operations

### Create (Insert)

MongoDB provides multiple methods for inserting documents into a collection.

#### Single Document Insert

```
db.collection.insertOne({
  name: "John Doe",
  email: "john@example.com",
  age: 30,
  created_at: new Date()
})
```

### Multiple Documents Insert

```
db.collection.insertMany([
  {
    name: "Jane Smith",
    email: "jane@example.com",
    age: 25,
    created_at: new Date()
  },
  {
    name: "Bob Johnson",
    email: "bob@example.com",
    age: 35,
    created_at: new Date()
  }
])
```

## Read (Find)

Find operations retrieve documents from collections.

### Find All Documents

```
db.collection.find()
```

### Find with Filtering

```
db.collection.find({ age: { $gt: 25 } })
```

### Find One Document

```
db.collection.findOne({ email: "john@example.com" })
```

### Limit and Skip

```
// Return 10 documents, skip the first 20
db.collection.find().skip(20).limit(10)
```

### Sort Documents

```
// Sort by age in ascending order
db.collection.find().sort({ age: 1 })

// Sort by age in descending order
db.collection.find().sort({ age: -1 })
```

## Update

MongoDB provides several ways to update documents.

### Update One Document

```
db.collection.updateOne(
  { email: "john@example.com" },
  { $set: { age: 31 } }
)
```

### Update Multiple Documents

```
db.collection.updateMany(
  { age: { $lt: 30 } },
  { $set: { status: "young" } }
)
```

### Replace One Document

```
db.collection.replaceOne(
  { _id: ObjectId("6078f24a5df54b0012345678") },
  {
    name: "John Doe Updated",
    email: "john.new@example.com",
    age: 32
  }
)
```

### Upsert - Insert if Not Exists

```
db.collection.updateOne(
  { email: "new.user@example.com" },
  { $set: { name: "New User", age: 40 } },
  { upsert: true }
)
```

## Delete

Remove documents from collections with delete operations.

### Delete One Document

```
db.collection.deleteOne({ email: "john@example.com" })
```

### Delete Multiple Documents

```
db.collection.deleteMany({ age: { $lt: 18 } })
```

### Delete All Documents

```
db.collection.deleteMany({})
```

# Query Operators

## Comparison Operators

MongoDB provides various comparison operators for queries.

| Operator | Description | Example |
|---|---|---|
| $eq | Equal to | { age: { $eq: 30 } } |
| $ne | Not equal to | { age: { $ne: 30 } } |
| $gt | Greater than | { age: { $gt: 30 } } |
| $gte | Greater than or equal | { age: { $gte: 30 } } |
| $lt | Less than | { age: { $lt: 30 } } |
| $lte | Less than or equal | { age: { $lte: 30 } } |
| $in | In an array | { age: { $in: [25, 30, 35] } } |
| $nin | Not in an array | { age: { $nin: [25, 30, 35] } } |

### Examples

```
// Find users with age between 25 and 35
db.users.find({ age: { $gte: 25, $lte: 35 } })

// Find users with skills in JavaScript or Python
```

```
db.users.find({ skills: { $in: ["JavaScript", "Python"] } })

// Find users who are not from US or Canada
db.users.find({ country: { $nin: ["US", "Canada"] } })
```

## Logical Operators

Combine multiple conditions with logical operators.

| Operator | Description | Example |
|---|---|---|
| $and | All conditions true | { $and: [{ age: { $gt: 20 } }, { age: { $lt: 30 } }] } |
| $or | Any condition true | { $or: [{ age: { $lt: 20 } }, { age: { $gt: 50 } }] } |
| $not | Negates a condition | { age: { $not: { $gt: 30 } } } |
| $nor | None of conditions true | { $nor: [{ age: 20 }, { name: "John" }] } |

### Examples

```
// Find users with age less than 20 or greater than 50
db.users.find({
  $or: [
    { age: { $lt: 20 } },
    { age: { $gt: 50 } }
  ]
})

// Find users who are 30+ years old and have JavaScript skills
db.users.find({
  $and: [
    { age: { $gte: 30 } },
    { skills: "JavaScript" }
  ]
})

// Find users who are neither from US nor have "admin" role
```

```
db.users.find({
  $nor: [
    { country: "US" },
    { role: "admin" }
  ]
})
```

## Element Operators

Query based on existence of fields or their types.

| Operator | Description | Example |
| --- | --- | --- |
| `$exists` | Field exists | `{ phone: { $exists: true } }` |
| `$type` | Field is of specific type | `{ age: { $type: "number" } }` |

### Examples

```
// Find users who have a phone field
db.users.find({ phone: { $exists: true } })

// Find users whose age is stored as a string
db.users.find({ age: { $type: "string" } })

// Find users who have no address field
db.users.find({ address: { $exists: false } })
```

## Array Operators

Query for array fields with these operators.

| Operator | Description | Example |
| --- | --- | --- |
| `$all` | Contains all elements | `{ tags: { $all: ["mongodb", "database"] } }` |
| `$elemMatch` | | |

| Operator | Description | Example |
| --- | --- | --- |
| | Element matches criteria | `{ scores: { $elemMatch: { $gt: 80, $lt: 90 } } }` |
| `$size` | Array with specific size | `{ friends: { $size: 3 } }` |

**Examples**

```
// Find documents with tags containing both "mongodb" and
        "database"
db.articles.find({ tags: { $all: ["mongodb", "database"] } })

// Find students with at least one score between 80 and 90
db.students.find({
  scores: { $elemMatch: { $gt: 80, $lt: 90 } }
})

// Find users with exactly 3 friends
db.users.find({ friends: { $size: 3 } })
```

# Aggregation Pipeline

The aggregation pipeline is MongoDB's framework for data processing, transformation, and analysis. It consists of stages that process documents sequentially.

## Stages Overview

Documents pass through each stage in the pipeline, where they can be filtered, grouped, sorted, or transformed.

## Common Stages

### $match

Filters documents like the `find()` method.

```
db.orders.aggregate([
  { $match: { status: "completed" } }
])
```

**$group**

Groups documents by specified fields and can perform aggregation operations.

```
db.orders.aggregate([
  {
    $group: {
      _id: "$customer_id",
      totalSpent: { $sum: "$amount" },
      count: { $sum: 1 }
    }
  }
])
```

**$sort**

Sorts documents based on specified fields.

```
db.orders.aggregate([
  { $sort: { amount: -1 } }
])
```

**$project**

Reshapes documents by including, excluding, or computing fields.

```
db.users.aggregate([
  {
    $project: {
      fullName: { $concat: ["$firstName", " ", "$lastName"] },
      age: 1,
      _id: 0
    }
  }
])
```

**$limit**

Limits the number of documents.

```
db.orders.aggregate([
  { $limit: 10 }
])
```

**$skip**

Skips a specified number of documents.

```
db.orders.aggregate([
  { $skip: 20 }
])
```

**$unwind**

Deconstructs an array field to create a separate document for each element.

```
db.products.aggregate([
  { $unwind: "$categories" }
])
```

**$lookup**

Performs a left outer join with another collection.

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",
      foreignField: "_id",
      as: "customer_info"
    }
  }
])
```

**$addFields**

Adds new fields to documents.

```
db.products.aggregate([
  {
    $addFields: {
      discountedPrice: { $multiply: ["$price", 0.9] }
    }
  }
])
```

## Complex Aggregation Examples

### Sales Analysis by Category and Month

```
db.sales.aggregate([
  // Filter completed orders
  { $match: { status: "completed" } },

  // Extract month and year
  {
    $addFields: {
      month: { $month: "$date" },
      year: { $year: "$date" }
    }
  },

  // Group by category, year, and month
  {
    $group: {
      _id: {
        category: "$product_category",
        year: "$year",
        month: "$month"
      },
      totalSales: { $sum: "$amount" },
      count: { $sum: 1 }
    }
  },

  // Sort by year, month, and total sales
  {
    $sort: {
      "_id.year": 1,
      "_id.month": 1,
      "totalSales": -1
    }
  },

  // Reshape output
  {
    $project: {
      _id: 0,
      category: "$_id.category",
      year: "$_id.year",
```

```
      month: "$_id.month",
      totalSales: 1,
      count: 1
    }
  }
])
```

**Customer Purchase Analysis with Joins**

```
db.orders.aggregate([
  // Match orders from past year
  {
    $match: {
      order_date: {
        $gte: new Date(new Date().setFullYear(new
        Date().getFullYear() - 1))
      }
    }
  },

  // Look up customer information
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",
      foreignField: "_id",
      as: "customer"
    }
  },

  // Unwind customer array (from lookup)
  { $unwind: "$customer" },

  // Look up product details
  {
    $lookup: {
      from: "products",
      localField: "product_id",
      foreignField: "_id",
      as: "product"
    }
  },

  // Unwind product array
```

```
{ $unwind: "$product" },

// Group by customer and category
{
  $group: {
    _id: {
      customer_id: "$customer_id",
      customer_name: "$customer.name",
      category: "$product.category"
    },
    totalSpent: { $sum: { $multiply: ["$quantity",
      "$product.price"] } },
    orderCount: { $sum: 1 },
    avgOrderValue: { $avg: { $multiply: ["$quantity",
      "$product.price"] } }
  }
},

// Add rank within category
{
  $setWindowFields: {
    partitionBy: "$_id.category",
    sortBy: { totalSpent: -1 },
    output: {
      rank: {
        $rank: {}
      }
    }
  }
},

// Format output
{
  $project: {
    _id: 0,
    customer_id: "$_id.customer_id",
    customer_name: "$_id.customer_name",
    category: "$_id.category",
    totalSpent: { $round: ["$totalSpent", 2] },
    orderCount: 1,
    avgOrderValue: { $round: ["$avgOrderValue", 2] },
    categoryRank: "$rank"
  }
},
```

```
  // Sort by category and total spent
  {
    $sort: {
      category: 1,
      totalSpent: -1
    }
  }
])
```

**Complex Data Transformation**

```
db.transactions.aggregate([
  // Match transactions for specific date range
  {
    $match: {
      date: {
        $gte: ISODate("2023-01-01"),
        $lte: ISODate("2023-12-31")
      }
    }
  },

  // Add calculated fields
  {
    $addFields: {
      month: { $month: "$date" },
      dayOfWeek: { $dayOfWeek: "$date" },
      isWeekend: {
        $in: [{ $dayOfWeek: "$date" }, [1, 7]] // 1 is Sunday, 7
        is Saturday
      },
      transactionProfit: {
        $subtract: ["$revenue", "$cost"]
      }
    }
  },

  // Facet to perform multiple aggregations in one pass
  {
    $facet: {
      "monthlyStats": [
        {
          $group: {
```

```
      _id: "$month",
      totalRevenue: { $sum: "$revenue" },
      totalProfit: { $sum: "$transactionProfit" },
      count: { $sum: 1 },
      avgTransactionValue: { $avg: "$revenue" }
    }
  },
  { $sort: { _id: 1 } }
],
"weekdayVsWeekend": [
  {
    $group: {
      _id: "$isWeekend",
      totalRevenue: { $sum: "$revenue" },
      avgTransactionValue: { $avg: "$revenue" },
      count: { $sum: 1 }
    }
  }
],
"topProducts": [
  {
    $group: {
      _id: "$product_id",
      totalRevenue: { $sum: "$revenue" },
      totalProfit: { $sum: "$transactionProfit" },
      count: { $sum: 1 }
    }
  },
  { $sort: { totalProfit: -1 } },
  { $limit: 5 },
  {
    $lookup: {
      from: "products",
      localField: "_id",
      foreignField: "_id",
      as: "productDetails"
    }
  },
  { $unwind: "$productDetails" },
  {
    $project: {
      _id: 0,
      productName: "$productDetails.name",
      totalRevenue: 1,
```

```
            totalProfit: 1,
            count: 1
          }
        }
      ]
    }
  }
])
```

# Indexes

Indexes improve query performance by creating efficient data structures for field access.

## Creating Indexes

```
// Create a single field index
db.users.createIndex({ email: 1 })

// Create a compound index
db.orders.createIndex({ customer_id: 1, date: -1 })

// Create a unique index
db.users.createIndex({ email: 1 }, { unique: true })

// Create a TTL index (documents expire after 3600 seconds)
db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds:
        3600 })

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

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

## Managing Indexes

```
// List all indexes
db.collection.getIndexes()

// Drop an index
db.users.dropIndex("email_1")

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

## Transactions

MongoDB supports multi-document transactions since version 4.0.

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

// Start a transaction
session.startTransaction()

try {
  // Perform operations
  db.accounts.updateOne(
    { _id: fromAccountId },
    { $inc: { balance: -amount } },
    { session }
  )

  db.accounts.updateOne(
    { _id: toAccountId },
    { $inc: { balance: amount } },
    { session }
  )

  db.transfers.insertOne(
    {
      from: fromAccountId,
      to: toAccountId,
      amount: amount,
      date: new Date()
    },
    { session }
  )

  // Commit the transaction
  session.commitTransaction()
} catch (error) {
  // Abort the transaction on error
  session.abortTransaction()
  throw error
} finally {
  // End the session
  session.endSession()
}
```

# Advanced Queries

## Geospatial Queries

MongoDB supports various geospatial query operators to find places within a specified distance or area.

```
// Find places near a point (within 1000 meters)
db.places.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [-73.9667, 40.78]
      },
      $maxDistance: 1000
    }
  }
})

// Find places within a polygon
db.places.find({
  location: {
    $geoWithin: {
      $geometry: {
        type: "Polygon",
        coordinates: [
          [
            [-73.9667, 40.78],
            [-73.95, 40.78],
            [-73.95, 40.79],
            [-73.9667, 40.79],
            [-73.9667, 40.78]
          ]
        ]
      }
    }
  }
})

// Find places that intersect with a line
db.routes.find({
  path: {
    $geoIntersects: {
      $geometry: {
```

```
        type: "LineString",
        coordinates: [
          [-73.9667, 40.78],
          [-73.95, 40.79]
        ]
      }
    }
  }
})
```

### Text Search

Text search allows searching for text content within documents.

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

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

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

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

// Search with negation
db.articles.find({ $text: { $search: "mongodb -sql" } })
```

### Projections

Projections control which fields are returned in query results.

```
// Include only specified fields
db.users.find({ age: { $gt: 30 } }, { name: 1, email: 1 })

// Exclude specified fields
db.users.find({ age: { $gt: 30 } }, { password: 0, secret_key:
        0 })

// Projection with array slicing
db.posts.find(
  { author: "John" },
```

```
  { title: 1, comments: { $slice: 5 } } // Get first 5 comments
        only
)

// Projection with elemMatch
db.students.find(
  { "grades.score": { $gt: 90 } },
  { name: 1, "grades.$": 1 } // Return only the matching grade
)

// Computed projections
db.products.aggregate([
  {
    $project: {
      name: 1,
      price: 1,
      discountedPrice: { $multiply: ["$price", 0.9] },
      priceRange: {
        $switch: {
          branches: [
            { case: { $lt: ["$price", 50] }, then: "Budget" },
            { case: { $lt: ["$price", 100] }, then: "Regular" },
            { case: { $gte: ["$price", 100] }, then: "Premium" }
          ],
          default: "Unknown"
        }
      }
    }
  }
])
```

# Performance Optimization

### Using Explain to Analyze Queries

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

// Analyze an aggregation execution plan
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: { _id: "$customer_id", total: { $sum: "$amount" } } }
], { explain: true })
```

## Query Optimization Tips

1. **Use appropriate indexes**:
   - Create indexes for frequently queried fields
   - Use compound indexes for queries with multiple conditions
   - Put equality conditions before range conditions in compound indexes
2. **Limit result sets**:
   - Use `limit()` when you don't need all results
   - Use projections to return only needed fields
3. **Avoid regex queries without index prefixes**:
   - Bad: `/.*smith/i`
   - Better: `/^smith/i` (can use index)
4. **Use covered queries**:
   - All fields in query and projection should be part of an index
5. **Optimize aggregation pipelines**:
   - Put `$match` and `$limit` stages early in the pipeline
   - Use indexes for fields used in `$match`, `$sort`, and `$group` stages
6. **Read Concerns and Write Concerns**:
   - Adjust read/write concerns based on consistency vs. performance needs
7. **Appropriate shard keys**:
   - Choose shard keys that distribute data evenly and support common queries