MongoDB Cheatsheet

Essential operations for NoSQL database management and document manipulation

This cheatsheet provides a quick reference to fundamental MongoDB operations, syntax, and advanced features, ideal for both beginners and experienced developers for efficient NoSQL database management.

Database Operations
Create and manage databases

Document Queries
Find and filter data

CRUD Operations
Insert, update, and delete documents

Data Aggregation
Process and analyze data

Indexing
Optimize query performance

Database & Collection Management

Show Databases: `show dbs`

Display all databases on the MongoDB server.

```
// Show all databases
show dbs
// Show current database
db
// Get database stats
db.stats()
// Get database help
db.help()
```

Use Database: `use database_name`

Switch to a specific database (creates if doesn't exist).

```
// Switch to myapp database
use myapp
// Create database by inserting data
use newdb
db.users.insertOne({name: "John"})
```

Drop Database: `db.dropDatabase()`

Delete the current database and all its collections.

```
// Drop current database
db.dropDatabase()
// Confirm with database name
use myapp
db.dropDatabase()
```

Show Collections: `show collections`

List all collections in the current database.

```
// Show all collections
show collections
// Alternative method
db.runCommand("listCollections")
```

Create Collection: `db.createCollection()`

Create a new collection with optional configuration.

```
// Create simple collection
db.createCollection("users")
// Create with options
db.createCollection("logs", {
  capped: true,
  size: 1000000,
  max: 1000
})
```

Drop Collection: `db.collection.drop()`

Delete a collection and all its documents.

```
// Drop users collection
db.users.drop()
// Check if collection was dropped
show collections
```

Document Structure & Info

Understand the structure and summary of your collections and documents.

01

Collection Stats: `db.collection.stats()`

Displays comprehensive statistics about a collection including size, document count, and index information.

```
// Collection statistics
db.users.stats()
// Count documents
db.users.countDocuments()
// Estimated count (faster)
db.users.estimatedDocumentCount()
// Check collection indexes
db.users.getIndexes()
```

02

Sample Documents: `db.collection.findOne()`

Retrieve sample documents to understand structure and data types.

```
// Get one document
db.users.findOne()
// Get specific document
db.users.findOne({name: "John"})
// Get document with all fields shown
db.users.findOne({}, {_id: 0})
```

03

Explore Data: `db.collection.find().limit()`

Browse through collection data with pagination and formatting.

```
// First 5 documents
db.users.find().limit(5)
// Skip and limit (pagination)
db.users.find().skip(10).limit(5)
// Pretty format
db.users.find().pretty()
```

Document Insertion (Create)

Insert One: `db.collection.insertOne()`

Add a single document to a collection.

```
// Insert single document
db.users.insertOne({
  name: "John Doe",
  age: 30,
  email: "john@example.com"
})
// Insert with custom _id
db.users.insertOne({
  _id: "custom_id_123",
  name: "Jane Doe",
  status: "active"
})
```

Insert Many: `db.collection.insertMany()`

Add multiple documents in a single operation.

```
// Insert multiple documents
db.users.insertMany([
  {name: "Alice", age: 25},
  {name: "Bob", age: 35},
  {name: "Charlie", age: 28}
])
// Insert with options
db.users.insertMany([
  {name: "Dave", age: 40},
  {name: "Eve", age: 22}
], {ordered: false})
```

Insert with Date: `new Date()`

Add documents with timestamp fields.

```
// Insert with current date
db.posts.insertOne({
  title: "My Blog Post",
  content: "Post content here",
  createdAt: new Date(),
  publishDate: ISODate("2024-01-15")
})
```

Insert Nested Documents

Add documents with embedded objects and arrays.

```
// Insert with nested objects
db.users.insertOne({
  name: "John Doe",
  address: {
    street: "123 Main St",
    city: "New York",
    zip: "10001"
  },
  hobbies: ["reading", "swimming", "coding"]
})
```

Document Querying (Read)

Basic Find: `db.collection.find()`

Retrieve documents based on query conditions.

```
// Find all documents
db.users.find()
// Find with condition
db.users.find({age: 30})
// Find with multiple conditions (AND)
db.users.find({age: 30, status: "active"})
// Find with OR condition
db.users.find({$or: [{age: 25}, {age: 30}]})
```

Projection: `db.collection.find({}, {})`

Control which fields are returned in the results.

```
// Include specific fields
db.users.find({}, {name: 1, age: 1})
// Exclude specific fields
db.users.find({}, {password: 0, _id: 0})
// Nested field projection
db.users.find({}, {"address.city": 1})
```

Query Operators: `$gt`, `$lt`, `$in`, etc.

Use comparison and logical operators for complex queries.

```
// Greater than, less than
db.users.find({age: {$gt: 25, $lt: 40}})
// In array
db.users.find({status: {$in: ["active", "pending"]}})
// Not equal
db.users.find({status: {$ne: "inactive"}})
// Exists
db.users.find({email: {$exists: true}})
```

Text Search: `$text`, `$regex`

Search documents using text and pattern matching.

```
// Text search (requires text index)
db.posts.find({$text: {$search: "mongodb tutorial"}})
// Regex search
db.users.find({name: {$regex: "^John", $options: "i"}})
// Case-insensitive search
db.users.find({email: {$regex: "@gmail.com$"}})
```

Document Updates

Update One: `db.collection.updateOne()`

Modify the first document that matches the query.

```
// Update single field
db.users.updateOne(
  {name: "John Doe"},
  {$set: {age: 31}}
)
// Update multiple fields
db.users.updateOne(
  {_id: ObjectId("...")},
  {$set: {age: 31, status: "updated"}}
)
// Upsert (insert if not found)
db.users.updateOne(
  {name: "New User"},
  {$set: {age: 25}},
  {upsert: true}
)
```

Update Many: `db.collection.updateMany()`

Modify all documents that match the query.

```
// Update multiple documents
db.users.updateMany(
  {status: "inactive"},
  {$set: {status: "archived"}}
)
// Increment values
db.posts.updateMany(
  {category: "tech"},
  {$inc: {views: 1}}
)
```

Update Operators: `$set`, `$unset`, `$push`

Use various operators to modify document fields.

```
// Set and unset fields
db.users.updateOne(
  {name: "John"},
  {$set: {lastLogin: new Date()}, $unset: {temp: ""}}
)
// Push to array
db.users.updateOne(
  {name: "John"},
  {$push: {hobbies: "gaming"}}
)
// Pull from array
db.users.updateOne(
  {name: "John"},
  {$pull: {hobbies: "reading"}}
)
```

Replace Document: `db.collection.replaceOne()`

Replace an entire document except the _id field.

```
// Replace entire document
db.users.replaceOne(
  {name: "John Doe"},
  {
    name: "John Smith",
    age: 35,
    email: "johnsmith@example.com"
  }
)
```

Data Aggregation

Basic Aggregation: `db.collection.aggregate()`

Process data through aggregation pipeline stages.

```
// Group and count
db.users.aggregate([
  {$group: {_id: "$status", count: {$sum: 1}}}
])
// Match and group
db.orders.aggregate([
  {$match: {status: "completed"}},
  {$group: {_id: "$customerId", total: {$sum: "$amount"}}}
])
```

Common Stages: `$match`, `$group`, `$sort`

Use pipeline stages to transform and analyze data.

```
// Complex aggregation pipeline
db.sales.aggregate([
  {$match: {date: {$gte: ISODate("2024-01-01")}}},
  {$group: {
    _id: "$product",
    totalSales: {$sum: "$amount"},
    avgPrice: {$avg: "$price"}
  }},
  {$sort: {totalSales: -1}},
  {$limit: 10}
])
```

Aggregation Operators: `$sum`, `$avg`, `$max`

Calculate statistical values and perform mathematical operations.

```
// Statistical operations
db.products.aggregate([
  {$group: {
    _id: "$category",
    maxPrice: {$max: "$price"},
    minPrice: {$min: "$price"},
    avgPrice: {$avg: "$price"},
    count: {$sum: 1}
  }}
])
```

Projection Stage: `$project`

Transform document structure and create calculated fields.

```
// Project and calculate fields
db.users.aggregate([
  {$project: {
    name: 1,
    age: 1,
    isAdult: {$gte: ["$age", 18]},
    fullName: {$concat: ["$firstName", " ", "$lastName"]}
  }}
])
```

Document Deletion

Remove documents from collections safely and efficiently.

Delete One: `db.collection.deleteOne()`

Remove the first document that matches the query condition.

```
// Delete single document
db.users.deleteOne({name: "John Doe"})
// Delete by ID
db.users.deleteOne({_id: ObjectId("...")})
// Delete with condition
db.posts.deleteOne({status: "draft", author: "unknown"})
```

Delete Many: `db.collection.deleteMany()`

Remove all documents that match the query condition.

```
// Delete multiple documents
db.users.deleteMany({status: "inactive"})
// Delete all documents (be careful!)
db.temp_collection.deleteMany({})
// Delete with date condition
db.logs.deleteMany({
  createdAt: {$lt: new Date("2024-01-01")}
})
```

Find and Delete: `db.collection.findOneAndDelete()`

Find a document and delete it in a single atomic operation.

```
// Find and delete
const deletedDoc = db.users.findOneAndDelete(
  {status: "pending"}
)
// Find and delete with options
db.queue.findOneAndDelete(
  {processed: false},
  {sort: {priority: -1}}
)
```

Indexing & Performance

Create and manage indexes to optimize query performance.

Create Index: `db.collection.createIndex()`

Create indexes on fields to speed up queries.

```
// Single field index
db.users.createIndex({email: 1})
// Compound index
db.users.createIndex({status: 1, createdAt: -1})
// Text index for search
db.posts.createIndex({title: "text", content: "text"})
// Unique index
db.users.createIndex({email: 1}, {unique: true})
```

Index Management: `getIndexes()`, `dropIndex()`

View and manage existing indexes on collections.

```
// List all indexes
db.users.getIndexes()
// Drop specific index
db.users.dropIndex({email: 1})
// Drop index by name
db.users.dropIndex("email_1")
// Drop all indexes except _id
db.users.dropIndexes()
```

Query Performance: `explain()`

Analyze query execution and performance statistics.

```
// Explain query execution
db.users.find({age: {$gt: 25}}).explain("executionStats")
// Check if index is used
db.users.find({email: "john@example.com"}).explain()
// Analyze aggregation performance
db.users.aggregate([
  {$match: {status: "active"}},
  {$group: {_id: "$department", count: {$sum: 1}}}
]).explain("executionStats")
```

Performance Tips

Best practices for optimizing MongoDB queries and operations.

```
// Use projection to limit data transfer
db.users.find({status: "active"}, {name: 1, email: 1})
// Limit results for better performance
db.posts.find().sort({createdAt: -1}).limit(10)
// Use hint to force specific index
db.users.find({age: 25}).hint({age: 1})
```

MongoDB Shell & Connection

Connect to MongoDB and manage shell sessions effectively.

Connect to MongoDB: `mongosh`

Start MongoDB shell and connect to different instances.

```
// Connect to local MongoDB
mongosh
// Connect to specific host and port
mongosh "mongodb://localhost:27017"
// Connect to remote server
mongosh
"mongodb://username:password@host:port/database"
// Connect with options
mongosh --host localhost --port 27017
```

Shell Helpers: `help`, `exit`

Get help information and manage shell sessions.

```
// General help
help
// Database specific help
db.help()
// Collection specific help
db.users.help()
// Exit shell
exit
```

Shell Variables and Settings

Configure shell behavior and use JavaScript variables.

```
// Set variable
var myQuery = {status: "active"}
db.users.find(myQuery)
// Configure display options
db.users.find().pretty()
// Show execution time
db.users.find({age: 25}).explain("executionStats")
// Use JavaScript in shell
var user = db.users.findOne({name: "John"})
print("User age: " + user.age)
```

Data Import & Export

Transfer data between MongoDB and external formats.

Import Data: `mongoimport`

Load data from JSON, CSV, or TSV files into MongoDB.

```
// Import JSON file
mongoimport --db myapp --collection users --file
users.json
// Import CSV file
mongoimport --db myapp --collection products \
  --type csv --headerline --file products.csv
// Import with upsert
mongoimport --db myapp --collection users \
  --file users.json --mode upsert
```

Export Data: `mongoexport`

Export MongoDB data to JSON or CSV format.

```
// Export to JSON
mongoexport --db myapp --collection users \
  --out users.json
// Export to CSV
mongoexport --db myapp --collection users \
  --type csv --fields name,email,age --out users.csv
// Export with query
mongoexport --db myapp --collection users \
  --query '{"status":"active"}' --out active_users.json
```

Backup: `mongodump`

Create binary backups of MongoDB databases.

```
// Backup entire database
mongodump --db myapp --out /backup/
// Backup specific collection
mongodump --db myapp --collection users --out /backup/
// Backup with compression
mongodump --db myapp --gzip --out /backup/
```

Restore: `mongorestore`

Restore MongoDB data from binary backups.

```
// Restore database
mongorestore --db myapp /backup/myapp/
// Restore with drop
mongorestore --db myapp --drop /backup/myapp/
// Restore compressed backup
mongorestore --gzip --db myapp /backup/myapp/
```

MongoDB Installation & Setup

Install and configure MongoDB for your development environment.

MongoDB Community Server

Download and install MongoDB Community Edition.

```
# Ubuntu/Debian
sudo apt-get install -y mongodb-org
# Start MongoDB service
sudo systemctl start mongod
# Enable auto-start
sudo systemctl enable mongod
# Check status
sudo systemctl status mongod
```

Docker Installation

Run MongoDB using Docker containers.

```
# Pull MongoDB image
docker pull mongo
# Run MongoDB container
docker run --name mongodb -d \
  -p 27017:27017 \
  -v mongodb_data:/data/db \
  mongo
# Connect to container
docker exec -it mongodb mongosh
```

MongoDB Compass (GUI)

Install and use MongoDB's official GUI tool.

```
# Download from mongodb.com
# Connect using connection string
mongodb://localhost:27017
# Features available:
# - Visual query builder
# - Schema analysis
# - Performance monitoring
# - Index management
```

Configuration & Security

Configure MongoDB settings and implement security best practices.

Authentication: Create Users

Set up database users with proper roles and permissions.

```
// Create admin user
use admin
db.createUser({
  user: "admin",
  pwd: "securepassword",
  roles: [{role: "root", db: "admin"}]
})
// Create database user
use myapp
db.createUser({
  user: "appuser",
  pwd: "password123",
  roles: [{role: "readWrite", db: "myapp"}]
})
```

Enable Authentication

Configure MongoDB to require authentication.

```
# Edit /etc/mongod.conf
security:
  authorization: enabled
# Restart MongoDB
sudo systemctl restart mongod
# Connect with authentication
mongosh -u admin -p --authenticationDatabase admin
```

Replica Sets: `rs.initiate()`

Set up replica sets for high availability.

```
// Initialize replica set
rs.initiate({
  _id: "myReplicaSet",
  members: [
    {_id: 0, host: "mongodb1:27017"},
    {_id: 1, host: "mongodb2:27017"},
    {_id: 2, host: "mongodb3:27017"}
  ]
})
// Check replica set status
rs.status()
```

Configuration Options

Common MongoDB configuration settings.

```
# mongod.conf example
storage:
  dbPath: /var/lib/mongodb
systemLog:
  destination: file
  path: /var/log/mongodb/mongod.log
net:
  port: 27017
  bindIp: 127.0.0.1
processManagement:
  fork: true
```

Error Handling & Debugging

Diagnose and resolve common MongoDB issues and errors.

Common Errors and Solutions

Identify and fix frequently encountered MongoDB problems.

```
// Connection errors
// Check if MongoDB is running
sudo systemctl status mongod
// Check port availability
netstat -tuln | grep 27017
// Duplicate key error handling
try {
  db.users.insertOne({email: "existing@example.com"})
} catch (e) {
  if (e.code === 11000) {
    print("Email already exists")
  }
}
```

Monitoring: `db.currentOp()`, `db.serverStatus()`

Monitor database operations and server performance.

```
// Check current operations
db.currentOp()
// Kill long-running operation
db.killOp(operationId)
// Server status
db.serverStatus()
// Connection stats
db.runCommand({connPoolStats: 1})
```

Profiling: `db.setProfilingLevel()`

Enable profiling to analyze slow operations.

```
// Enable profiling for slow operations (>100ms)
db.setProfilingLevel(1, {slowms: 100})
// Enable profiling for all operations
db.setProfilingLevel(2)
// View profiler data
db.system.profile.find().sort({ts: -1}).limit(5)
// Disable profiling
db.setProfilingLevel(0)
```

Advanced Operations

Transactions: `session.startTransaction()`

Use multi-document transactions for data consistency.

```
// Start session and transaction
const session = db.getMongo().startSession()
session.startTransaction()
try {
  const users = session.getDatabase("myapp").users
  const accounts =
session.getDatabase("myapp").accounts

  users.insertOne({name: "John", balance: 100})
  accounts.updateOne({userId: "john"}, {$inc: {balance:
-100}})

  session.commitTransaction()
} catch (error) {
  session.abortTransaction()
} finally {
  session.endSession()
}
```

Change Streams: `db.collection.watch()`

Watch for real-time changes in collections.

```
// Watch collection changes
const changeStream = db.users.watch()
changeStream.on('change', (change) => {
  console.log('Change detected:', change)
})
// Watch with filter
const pipeline = [{$match: {operationType: 'insert'}}]
const changeStream = db.users.watch(pipeline)
```

Reference: This cheatsheet covers essential MongoDB commands and modern practices for efficient NoSQL database management and document-based data operations.