**MongoDB Tutorial — Clear & Simple Guide**

**✅ 1. INSERT OPERATIONS**

**1.1 Create a Database**

use firstdb

✔️ Switches to or creates a database called **firstdb**.

**1.2 Insert One Document**

db.students.insertOne({name: "kim", age: 12, standard: "seventh"})

✔️ Adds one student record into the **students** collection.

**1.3 Create a Collection**

db.createCollection("teachers")

✔️ Creates a collection named **teachers**.

**1.4 Insert Many Documents**

db.teachers.insertMany([

{ name: "roger", subject: "maths" },

{ name: "anita", subject: "chemistry" }

])

✔️ Inserts multiple teacher records into **teachers** collection.

**1.5 View Documents**

db.teachers.find()

db.teachers.find().forEach(printjson)

✔️ Shows all documents. printjson prints nicely formatted output.

**1.6 Find Specific Document**

db.teachers.find({name: "roger"})

✔️ Finds only the teacher named "roger".

**1.7 Create a Capped Collection**

db.createCollection("cap1", { capped: true, size: 10000 })

✔️ Creates a fixed-size collection. When full, oldest data gets removed automatically.

**1.8 Check if Collection is Capped**

db.cap1.isCapped()

✔️ Returns **true** if collection is capped.

**1.9 Capped Collection with Document Limit**

db.createCollection("cap2", { capped: true, size: 100000, max: 5 })

✔️ Capped collection limited to 5 documents maximum.

**1.10 Insert into Capped Collection**

db.cap2.insertMany([

{\_id: 10, name: "neena"},

{\_id: 11, name: "lalitha"},

// ...more documents

])

✔️ Inserts documents but only the latest 5 will be kept, oldest get removed.

**1.11 Convert Normal Collection to Capped**

db.createCollection("convertcap1")

db.runCommand({ "convertToCapped": "convertcap1", size : 10000 })

✔️ Converts an existing collection into a capped collection.

**1.12 Bulk Insert Operations**

**Ordered Bulk (Stops on first error)**

var bulkOrdered = db.bulk1.initializeOrderedBulkOp()

bulkOrdered.insert({ name: "suresh" })

bulkOrdered.insert({ \_id: 1, name: "ravi" })

bulkOrdered.insert({ \_id: 1, name: "pavan" }) // duplicate \_id, error here

bulkOrdered.insert({ name: "pinky" }) // NOT executed due to error

bulkOrdered.execute()

✔️ Inserts in order, stops when it hits an error.

**Unordered Bulk (Continues despite errors)**

var bulkUnOrdered = db.bulk2.initializeUnorderedBulkOp()

bulkUnOrdered.insert({ name: "suresh" })

bulkUnOrdered.insert({ \_id: 1, name: "ravi" })

bulkUnOrdered.insert({ \_id: 1, name: "pavan" }) // duplicate \_id, error

bulkUnOrdered.insert({ name: "pinky" }) // executed regardless of error

bulkUnOrdered.execute()

✔️ Executes all inserts, errors don’t stop the process.

**❌ 2. DELETE OPERATIONS**

**2.1 Insert Documents for Deletion Practice**

db.deleteplist.insertMany([

{ product: "register", quantity: 25, size: {...}, status: "A" },

// ...more documents

])

✔️ Adds sample data to practice deletion.

**2.2 Delete by Field**

db.deleteplist.deleteOne({ product: "sheet" })

✔️ Deletes one document where product equals "sheet".

**2.3 Delete by Embedded Document (Exact Match)**

db.deleteplist.deleteOne({ size: { height: 14, weight: 21, unit: "cm" } })

✔️ Deletes one document with exact embedded object match.

**2.4 Delete by Field Inside Embedded Document**

db.deleteplist.deleteOne({ "size.unit": "in" })

✔️ Deletes one document where embedded field unit equals "in".

**2.5 Delete by Comparison Operator**

db.deleteplist.deleteOne({ "size.height": { $lt: 15 } })

✔️ Deletes one document where height is less than 15.

**2.6 Delete by Exact Array Match**

db.deleteplist.deleteOne({ tags: ["red", "blank"] })

✔️ Deletes if the array matches exactly.

**2.7 Delete if Array Contains a Value**

db.deleteplist.deleteMany({ tags: "red" })

✔️ Deletes all documents where tags array contains "red".

**2.8 Delete by Array of Embedded Documents**

db.deleteplist.deleteMany({ "instock": { warehouse: "A", quantity: 5 } })

✔️ Deletes documents where instock array has an object with matching fields.

**2.9 Remove Method (Deprecated)**

db.deleteplist.remove({ ... })

❌ Avoid using remove(), it is deprecated.

**📊 3. AGGREGATE OPERATIONS**

**3.1 Insert Documents for Aggregation**

db.aggregateproduct.insertMany([

{ product: "register", quantity: 25, size: {...}, status: "A" },

// ...more documents

])

✔️ Sample data for aggregation practice.

**3.2 Types of Aggregation**

* **Aggregation Pipeline:** Most common, processes data step-by-step (filter, group, sort, etc.).
* **Map Reduce:** Older, complex method (now mostly replaced).
* **Single Purpose Methods:** Simple commands like count(), distinct().

**Step 1: Aggregation Pipeline Example — Match & Group**

db.aggregateproduct.aggregate([

{ $match: { status: "A" } },

{ $group: { \_id: "$product", total: { $sum: "$quantity" } } }

])

* $match filters documents where status is "A".
* $group groups by product and sums quantity.

**Step 2: Adding Sorting**

db.aggregateproduct.aggregate([

{ $match: { status: "A" } },

{ $group: { \_id: "$product", total: { $sum: "$quantity" } } },

{ $sort: { total: 1 } } // ascending order

])

**Step 3: Using $addFields (Computed Fields)**

db.aggregateproduct.aggregate([

{ $match: { status: "A" } },

{ $addFields: {

maxWeight: { $max: "$size.weight" },

minSizeHeight: { $min: "$size.height" }

}

}

])

Note: $max and $min here apply per document, not across documents (for cross-document use, group first).

**Step 4: Project Specific Fields Before Grouping**

db.aggregateproduct.aggregate([

{ $project: { product: 1, quantity: 1 } },

{ $group: { \_id: "$product", total: { $sum: "$quantity" } } },

{ $sort: { total: 1 } }

])

Sample output:

[

{ "\_id": "register", "total": 75 },

{ "\_id": "envelop", "total": 90 },

{ "\_id": "book", "total": 150 },

{ "\_id": "timetable", "total": 150 },

{ "\_id": "sheet", "total": 200 }

]

**Step 5: Common Aggregation Pipeline Stages**

* $project — Select or reshape fields.
* $unset — Remove fields.
* $addFields / $set — Add or modify fields.
* $match — Filter documents.
* $group — Group and aggregate data.
* $sort — Sort results.
* $limit — Limit number of results.
* $skip — Skip documents.

Many operators exist for math, array, string, and date operations inside these stages.

Official docs:  
<https://docs.mongodb.com/manual/reference/operator/aggregation/project/>

**Step 6: Map-Reduce (Legacy) Example**

db.aggregateproduct.mapReduce(

function() { emit(this.product, 1); }, // Map: emit product & count 1

function(key, values) { return Array.sum(values); }, // Reduce: sum counts

{

query: { status: "A" }, // Filter on status "A"

out: { merge: "map\_reduce\_result" } // Output collection

}

)

Then check results:

db.map\_reduce\_result.find()

Notes:

* Map-Reduce is mostly deprecated in favor of Aggregation Pipeline.
* Not supported in MongoDB Atlas free or serverless tiers.

**Step 7: Aggregation Pipeline Memory Limits**

* Default memory limit: 100MB RAM.
* Use { allowDiskUse: true } option for large data:

db.aggregateproduct.aggregate(pipeline, { allowDiskUse

: true })

---

### Step 8: Distinct Values Example

```js

db.aggregateproduct.distinct("product", { status: "A" })

✔️ Returns all unique product values with status = "A".

**Summary**

* Use insertOne() and insertMany() to add data.
* Delete with deleteOne() and deleteMany() carefully using filters.
* Use Aggregation Pipeline for powerful querying, grouping, sorting, and transforming data.
* Avoid using deprecated methods like remove().
* For bulk inserts, choose between ordered (stop on error) or unordered (continue despite errors).
* Use capped collections for fixed-size, high-speed log-like data.