Aggregation Deep — Data Ka Hisab MongoDB Mein

Data ka hisaab, database ke andar: pipelines of $match, $group, $sum, $avg and $sort — totals, per-city sales and class averages, live.

Introduction

`find` fetches documents. But the dukaan owner does not want documents — he wants *answers*: **total sales today? Which city buys most? Average order per customer?** Fetching a thousand orders into your app and adding them in JavaScript works — slowly, and worse every day. **Aggregation** is MongoDB doing the hisaab *inside* the database: your data travels through a **pipeline** of stages — filter, group, total, sort — and only the small final answer makes the trip to your app. This lesson builds real pipelines on a real dukaan's orders.

(Roman Urdu: Aggregation = assembly line. Data ek sira se daalo, har stage apna kaam kare, doosre sira se sirf jawab nikle.)

Key points

- **A pipeline is stages in order:** `db.orders.aggregate([ stage1, stage2, ... ])` — each stage is a document whose key starts with `$`: `$match`, `$group`, `$sort`, `$limit`, `$project`. Data flows through them like items on a belt; each stage receives the previous stage's output.
- **$match is find, inside the pipeline:** it filters documents with the exact filter language of MG.4. Put it **first** whenever you can — every document it removes is a document no later stage must carry. Match early, carry less.
- **$group is the hisaab stage:** `{ $group: { _id: '$city', total: { $sum: '$amount' } } }` — `_id` names the grouping key (the field whose value separates the buckets — note the `$` before the field name: *the value of* city), and each extra field is a calculation per bucket.
- **The accumulator family:** `$sum` (add up — `$sum: 1` counts documents), `$avg` (average), `$min` / `$max` (the extremes), `$push` (collect values into an array). `_id: null` means *one bucket for everything* — the grand total.
- **Stages after $group see the new shape:** after grouping, documents have the `_id` and the calculated fields — *not* the original fields. Sorting by `total` works; sorting by `amount` after grouping is a letter to a house that moved.
- **$project reshapes the answer:** rename, hide the `_id`, compute new fields — the pipeline's finishing desk. Aggregation is SQL's GROUP BY grown into a full workshop: DB.8's subqueries' cousin, with more tools.

Code examples

The dukaan's `orders` collection used in every example (six orders, three statuses — note the simple numbered `_id`s; MongoDB allows choosing them):

```
db.orders.insertMany([
  { _id: 1, customer: 'Ali Raza', city: 'Sahiwal', amount: 1500, status: 'delivered' },
  { _id: 2, customer: 'Sara Khan', city: 'Lahore', amount: 2300, status: 'delivered' },
  { _id: 3, customer: 'Bilal Ahmed', city: 'Sahiwal', amount: 700, status: 'pending' },
  { _id: 4, customer: 'Hira Shah', city: 'Lahore', amount: 1200, status: 'delivered' },
  { _id: 5, customer: 'Usman Tariq', city: 'Karachi', amount: 950, status: 'delivered' },
  { _id: 6, customer: 'Ali Raza', city: 'Sahiwal', amount: 400, status: 'cancelled' }
])
```

Example 1 — The grand total: $match, then $group

```
db.orders.aggregate([
  { $match: { status: 'delivered' } },
  { $group: { _id: null, totalSales: { $sum: '$amount' } } }
])
```

What it does: Stage one keeps only delivered orders — pending (700) and cancelled (400) never travel further. Stage two puts everything into a single bucket (`_id: null`) and sums the amounts: 1500 + 2300 + 1200 + 950.

What you will see:

```
[ { _id: null, totalSales: 5950 } ]
```

One small document — **5950** — is the whole answer. Six documents went in, the hisaab happened inside the database, and a single number crossed the wire. This is the entire aggregation argument in one result.

Example 2 — Per-city sales, best first

```
db.orders.aggregate([
  { $match: { status: 'delivered' } },
  { $group: { _id: '$city', total: { $sum: '$amount' }, orders: { $sum: 1 } } },
  { $sort: { total: -1 } }
])
```

What it does: The same match, but now the grouping key is each order's city *value* (`'$city'` — the dollar sign means "the value of this field"). Each bucket gets two calculations: the summed `total`, and `orders` counted by adding 1 per document. A final `$sort` ranks the buckets.

What you will see:

```
[
  { _id: 'Lahore', total: 3500, orders: 2 },
  { _id: 'Sahiwal', total: 1500, orders: 1 },
  { _id: 'Karachi', total: 950, orders: 1 }
]
```

Check it by hand: Lahore = Sara 2300 + Hira 1200 = 3500, two orders. Sahiwal's *delivered* money is only Ali's 1500 — Bilal's 700 is still pending and the cancelled 400 never counted; the `$match` earned its place at the front. Karachi: Usman's single 950. The owner asked "which city buys most?" — the pipeline answered, ranked.

Example 3 — Class averages back at school: $avg + count

```
db.students.aggregate([
  { $group: { _id: '$class', avgMarks: { $avg: '$marks' }, students: { $sum: 1 } } },
  { $sort: { avgMarks: -1 } }
])
```

What it does: The same machinery, aimed at the school collection. Group by class, average the marks, count the students, rank by the average. No `$match` this time — every student counts.

What you will see:

```
[
  { _id: '10-A', avgMarks: 90, students: 2 },
  { _id: '10-B', avgMarks: 85.5, students: 2 },
  { _id: '9-A', avgMarks: 64, students: 1 }
]
```

Verify by hand: 10-A = (88 + 92) / 2 = **90**. 10-B = (76 + 95) / 2 = **85.5**. 9-A is Usman alone at 64. Three stages, three different collections' worth of questions across this lesson — one pattern: **filter (if needed) → group → calculate → sort.**

Quick recap

- **Aggregation** = a pipeline of `$` stages; data flows through, only the answer comes out.
- `$match` first (it is `find` in pipeline clothes) — carry less, work faster.
- `$group`: `_id` is the bucket key (`'$field'` = the field's value; `null` = one grand bucket); `$sum`, `$avg`, `$min`, `$max` calculate per bucket; `$sum: 1` counts.
- After `$group`, the document shape is new — later stages see buckets, not originals.

Common mistakes

- **Forgetting the $ before a field name in $group:** `{ _id: 'city' }` groups everything under the literal word "city" — one bucket, no error, wrong answer. `'$city'` (value) versus `'city'` (a label) is the most expensive missing character in MongoDB.
- **Matching late:** running `$group` over the whole collection and filtering the *results* (`$match` on totals) works — but the database carried every cancelled order through the maths for nothing. Filter raw documents early; filter *results* only when the condition is about the result itself (like `total > 1000`).

  (Roman Urdu: Pehle chhano, phir hisaab — ulta karne se jawab to milta hai, waqt zaya hota hai.)
- **Expecting original fields after $group:** asking `$sort: { amount: -1 }` after grouping fails quietly into nonsense order — after the group stage, `amount` no longer exists. Sort by what the stage produced (`total`), not by what it consumed.
- **Doing the hisaab in app code "just this once":** `find()` everything and sum in JavaScript is fine at six orders and a fire at six hundred thousand. Aggregation is not an advanced luxury; it is the normal way to ask *totals questions*.

Practice

1. Write the pipeline for *pending* money: total amount of orders with status 'pending', one bucket. Then check your answer against the dataset by hand.
2. Group orders by *customer* (all statuses) with each customer's total spend and order count, biggest spender first. Who wins — and does the cancelled order change the winner if you add a `$match` first? Explain the difference in one line.
3. On students: build the pipeline that answers "how many students does each *city* have?" — then extend it with a `$match` for marks of 80 or more before grouping. Which cities disappear, and why?

Practice quiz

1. In what order should $match and $group usually sit in a pipeline?

- $group first, then $match on the results
- $match first whenever you can, because every document it removes is one no later stage must carry; match early, carry less
- Order makes no difference
- $match may only come last

Answer: $match first whenever you can, because every document it removes is one no later stage must carry; match early, carry less

2. In { $group: { _id: '$city', total: { $sum: '$amount' } } }, what does '$city' mean, and what would 'city' mean?

- Both mean the same

- '$city' means the value of the city field, the bucket key; 'city' without the dollar would group everything under the literal word city, one bucket, wrong answer
- '$city' is a label, 'city' is the value
- The dollar sign is decoration

Answer: '$city' means the value of the city field, the bucket key; 'city' without the dollar would group everything under the literal word city, one bucket, wrong answer

3. What does _id: null mean in a $group stage?

- It groups by nothing and returns nothing
- One bucket for everything, the grand total
- An error, _id may never be null
- It counts documents only

Answer: One bucket for everything, the grand total

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/mongodb-aggregation · More free tools and notes at https://scopetreasure.com