Ministry of Higher Education and Scientific Research
University of Abdelhafid Boussouf - Mila
Institute of Mathematics and Computer Science
Department of Computer Science
Master 2 I2A – Big Data
2025/2026

Directed Works TD 8 – Aggregation Framework & Indexing Strategies

Exercise 1: The Pipeline Logic (Mental Compilation)

1. Filtering (Stage 1: `$match`)

- **Discarded:** Document with `_id: 2` (Sara) because `status` is "pending".
- **Kept:** Documents `_id: 1` (Amine) and `_id: 3` (Omar).

2. Explosion (Stage 2: `$unwind`)

- The `$unwind` stage deconstructs the `items` array. Every item becomes a separate document, carrying the original document's data with it.
- **Result (4 Documents in Memory):**
  1. `{ client: "Amine", item: "Laptop", total: 60000 }`
  2. `{ client: "Amine", item: "Mouse", total: 60000 }`
  3. `{ client: "Omar", item: "Laptop", total: 70000 }`
  4. `{ client: "Omar", item: "HDMI", total: 70000 }`

3. Grouping (Stage 3 & 4: `$group` + `$sort`)

- **Group key:** `_id: "$items"` (Group by Item Name).
- **Calculations:**
  - **Laptop:** Count = 2. Avg = (60000 + 70000) / 2 = 65000.
  - **Mouse:** Count = 1. Avg = 60000.
  - **HDMI:** Count = 1. Avg = 70000.
- **Final Output Table (Sorted by count desc):**

_id (Item)	count	avg_order_val
"Laptop"	**2**	**65000**
"Mouse"	1	60000
"HDMI"	1	70000

4. Logical Analysis

- **Does `avg_order_val` mean average item price?**
  - **NO.** It is the average of the *total basket size* where this item appeared.
- **Usefulness:** It answers: *"Do people who buy Laptops usually spend more money in total than people who buy Mice?"* (Market Basket Analysis). It is a very useful metric for cross-selling, even if it doesn't represent the unit price.

Exercise 2: Constructing Advanced Pipelines

1. Monthly Revenue Report

```
db.orders.aggregate([
  {
    $group: {
      _id: {
        year: { $year: "$order_date" },
        month: { $month: "$order_date" }
      },
      total_revenue: { $sum: "$total_amount" }
    }
  },
  { $sort: { "_id.year": 1, "_id.month": 1 } }
])
```

2. Segmentation (Customer Value)

```
db.orders.aggregate([
  {
    $project: {
      total_amount: 1,
      category: {
        $cond: {
          if: { $gt: ["$total_amount", 50000] },
          then: "Premium",
          else: "Standard"
        }
      }
    }
  }
])
```

3. Joins (Avg Spending by City)

```
db.orders.aggregate([
  // 1. Join with Users
  {
    $lookup: {
      from: "users",
      localField: "user_id",
      foreignField: "_id",
      as: "user_info" // Returns an array [ {username:..., address:...} ]
    }
  },
  // 2. Unwind the array (to access fields inside)
  { $unwind: "$user_info" },

  // 3. Group by City
  {
    $group: {
      _id: "$user_info.address.city",
      avg_spending: { $avg: "$total_amount" }
    }
  }
])
```

Exercise 3: Indexing & The ESR Rule

Part A: Selecting the Best Index Query: `find({ category: "Laptops", price: { $gt: 50000 } }).sort({ price: -1 })`

- **Rule:** **E**quality (First) -> **S**ort (Second) -> **R**ange (Last).
- **Analysis:**
  - **Equality:** `category: "Laptops"` -> Index must start with `category`.
  - **Sort:** `price: -1` -> Index should continue with `price`.
  - **Range:** `price: {$gt...}` -> Index ends with `price`.
- **Winner: Index 2** `{ "category": 1, "price": -1 }`
  - MongoDB jumps directly to the "Laptops" section of the index (Equality).
  - Inside that section, the data is *already sorted* by price (Sort).
  - It simply walks down the tree until price > 50000 (Range).
- *Why not Index 3?* `{ "price": -1, "category": 1 }`.
  - It sorts everything by price first. "Laptops" are scattered throughout the index (e.g., a cheap laptop, then a cheap mouse, then an expensive laptop). MongoDB has to scan *all* prices > 50000 and check "Is this a Laptop?" one by one.

Part B: he "Covered Query"

- A query is covered if the Index contains **all** the fields requested, so MongoDB doesn't need to look at the document itself.
- **The Fix:** We must project **only** the fields in the index (`category`, `price`) and explicitly **exclude** `_id` (which is fetched by default).

```
db.products.find(
  { "category": "Laptops", "price": { $gt: 50000 } },
  { "category": 1, "price": 1, "_id": 0 } // Projection
).sort({ "price": -1 })
```

Exercise 4: Schema Design & The 16MB Limit

1. Do the Math

- 16 MB = $16 \times 1024 \times 1024 = 16,777,216$ bytes.
- Max Reviews = $16,777,216 / 500 \approx 33,554$ reviews.
- *Teacher Note:* In reality, it's less because of BSON overhead (field names, types, length headers). Safe estimate is ~30k.

2. Failure Scenario

- **Result:** The **Write Operation Fails**.
- MongoDB throws a `DocumentTooLarge` exception. The application will crash when trying to save the 33,555th review. The database itself stays up, but that specific product becomes "read-only" essentially because no new info can be added.

3. The Fix (Subset Pattern)

- **Solution:** Use a Hybrid Approach.
- **Product Collection:** Store only the **Top 5 Reviews** (Subset) for fast display on the landing page.
- `{ "name": "Laptop", "latest_reviews": [ ...5 items... ] }`
- **Reviews Collection:** Store **ALL** reviews in a separate collection, linked by `product_id`.
- `{ "product_id": "prod_1", "user": "Ali", "text": "..." }`
- **Workflow:**
  - Load Page -> Read Product (Fast, gets top 5 reviews instantly).
  - User clicks "See All" -> Query `db.reviews.find({product_id: "prod_1"})`.