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 7 – NoSQL Data Modeling & MongoDB (Solution)

Exercise 1: Deciphering MQL (Code Interpretation)

1. **Command A:** `db.users.insertOne({...})`

Answer: "Insert a single new document into the `users` collection containing the username 'amine_23', the role 'student', and set the account to active."

2. **Command B:** `db.products.find({ "category": "Laptops", ... }, { "name": 1, ... })`

Answer: "Find all documents in the `products` collection where the category is 'Laptops' AND the price is less than 50,000. For the results, **display only** the `name` and `price` fields, and **hide** the `_id` field." *Teaching Point: Explain that `1` means include (Project) and `0` means exclude.*

3. **Command C:** `db.products.updateMany({...}, { "$inc": ... })`

Answer: "Find ALL documents where the category is 'Smartphones' and increase their `stock` value by 5." *Teaching Point: Differentiate `$inc` (adds to existing value) vs `$set` (overwrites value).*

4. **Command D:** `db.logs.deleteMany({})`

Answer: "Delete **every single document** in the `logs` collection." *Reason it's dangerous:* The filter `{}` matches everything. It is equivalent to a `TRUNCATE TABLE` in SQL but cannot be undone.

Exercise 2: Thinking in JSON (Syntax & Structure)

Goal: Catch syntax errors that crash Python/Node.js scripts.

Document A (Corrections):

```
{
    "name": "Gaming Mouse",      // Keys should be quoted (Strict JSON)
    "price": 50.00,              // ERROR 1: Use dot (.) for decimals, not comma (,)
    "tags": ["gaming", "accessory"], // ERROR 2: Remove trailing comma
    "is_available": true         // ERROR 3: Lowercase 'true' (Python uses True, JSON uses true)
}
```

Document B (Corrections):

```
{
    "_id": "user_123",
    "email": "test@gmail.com",
    "address": {                 // ERROR: Changed [] to {}
        "street": "12 Rue de la Liberté",
        "city": "Algiers"
    }
}
```

Explanation: An array `[]` is a list of values. An object `{}` is a collection of Key-Value pairs. Since we have keys ("street", "city"), we must use curly braces `{}`.

Exercise 3: Schema Design (The TechStore Scenario)

Goal: Validate the choice between Embedding and Referencing.

1. User ↔ Address (Embedding)

Decision: Embed the address inside the User document.

Justification: Access Pattern A states we *always* need the address when loading the user. Embedding allows us to fetch all data in a **single read operation** (Speed). Splitting it would require a secondary lookup (Join).

2. User ↔ Orders (Referencing)

Decision: Reference (store `user_id` inside the Order document).

Justification:
Unbounded Growth: A user can have infinite orders. If we embed them, the User document could exceed the **16MB limit**.
Access Pattern: We rarely show *all* history at once. Referencing allows us to query only the "last 5 orders" easily (`db.orders.find({user_id: X}).limit(5)`).

3. Product ↔ Specs (Embedding / Attribute Pattern)
Decision: Embed as an Array of Objects (The "Attribute Pattern").
Proposed Structure:

```
{
  "name": "Laptop",
  "specs": [
    { "k": "RAM", "v": "16GB" },
    { "k": "Screen", "v": "15 inch" }
  ]
}
```

Justification: The list is small (max 20) and static. Embedding is efficient. Using the Key-Value array format `{"k":..., "v":...}` allows for easy indexing if we want to search by specific specs later.

Exercise 4: Writing Queries (MQL)

1. Select (Computers < 1500)

```
db.products.find({
    "category": "Computers",
    "price": { "$lt": 1500 }
})
```

2. Logical (Gaming OR > 3000)

```
db.products.find({
    "$or": [
        { "tags": "gaming" },
        { "price": { "$gt": 3000 } }
    ]
})
```

Note: `tags: "gaming"` works even if tags is an array. MongoDB automatically searches inside the array.

3. Embedded (RAM is 16GB)

```
db.products.find({
    "specs.ram": "16GB"  // Dot notation requires quotes
})
```

4. Atomic Update (Increase Stock)

```
db.products.updateOne(
    { "_id": 1 },
    { "$inc": { "stock": 10 } }
)
```

5. Array Update (Add Tag if Unique)

```
db.products.updateOne(
    { "_id": 1 },
    { "$addToSet": { "tags": "bestseller" } }
)
```

Teaching Point: If we used `$push`, we might get duplicates (e.g., `["office", "bestseller", "bestseller"]`). `$addToSet` prevents this.

Exercise 5: Modeling Relationships (Many-to-Many)

1. Schema Proposal To find "All products sold by Supplier X" efficiently, we must store references to the products **inside the Supplier document**.

Schema:

```
// Collection: suppliers
{
  "_id": "supp_01",
  "name": "TechDistro DZ",
  // The Solution: An array of Product IDs here
  "product_ids": [ 101, 102, 105 ]
}
```

Query: `db.suppliers.findOne({_id: "supp_01"})` -> This gives us the list of product IDs immediately.

2. Denormalization

- **Definition:** Denormalization in NoSQL means deliberately **duplicating data** across multiple collections to avoid expensive Joins.
- **Context:** In an Order, instead of just saving `product_id: 101`, we save a copy of the name and price: `{ id: 101, name: "Laptop", price: 1200 }`.
- **Why?**
  1. **Read Performance:** We don't need to query the `products` collection to print an invoice.
  2. **History Preservation:** If the product price changes to 1400 tomorrow, the historical order must still show 1200.