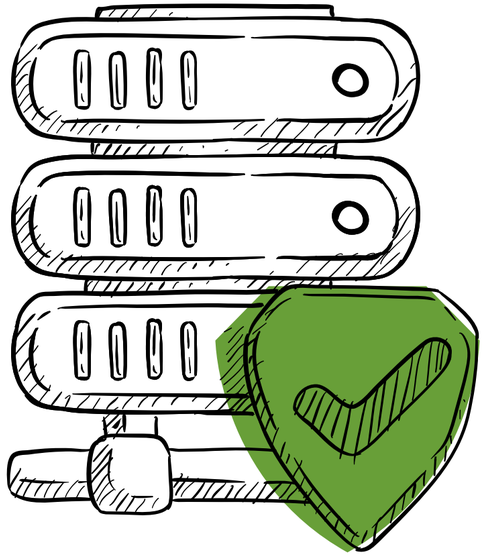
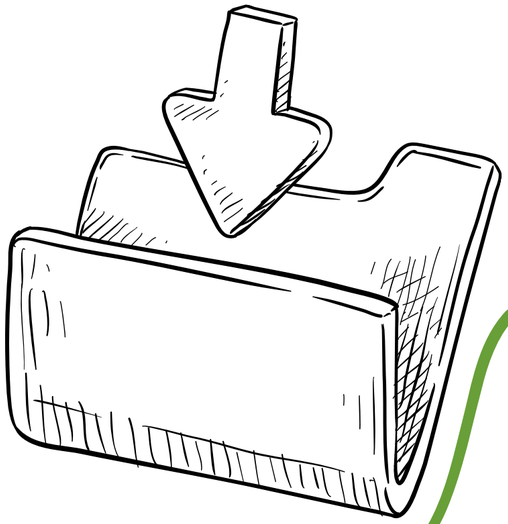
MONGO DATABASE

DAY03

Prepared by
Noha Shehab

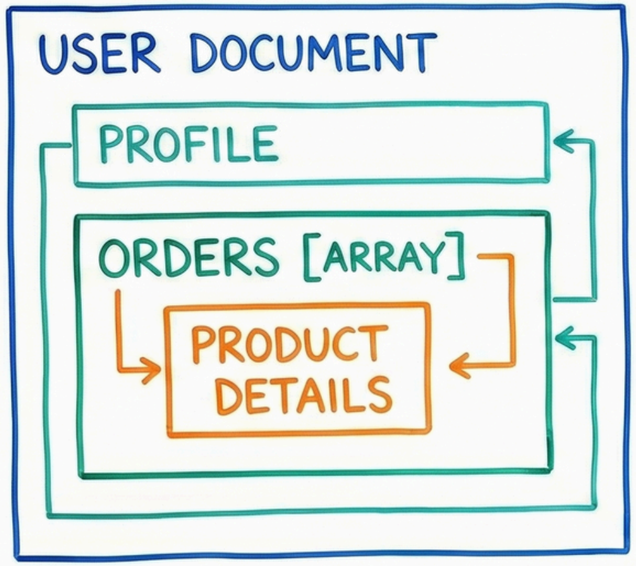




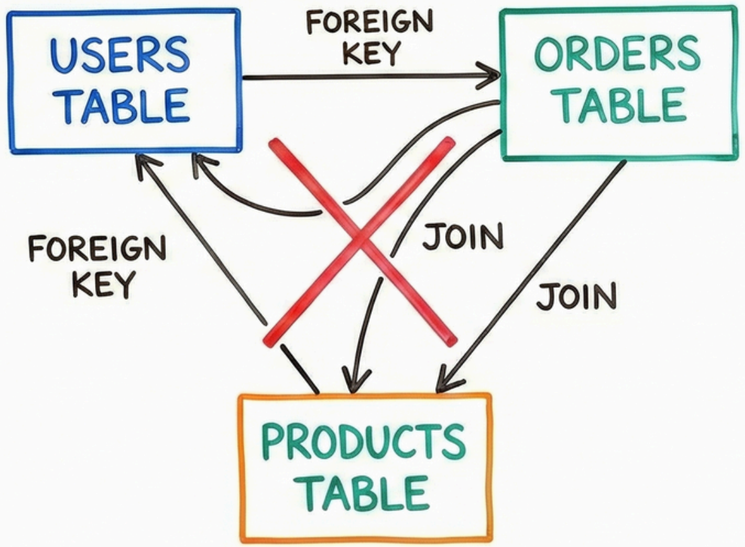
DATA MODELING

DATA MODELING PRINCIPLES: DOCUMENT vs. RELATIONAL (GRAPH)

MONGODB DOCUMENT MODEL (NO GRAPH)



RELATIONAL / GRAPH MODEL



DATA MODELING

- Data modeling in MongoDB is fundamentally different from relational database modeling.
- In a relational database, you model based on the data's structure, **prioritizing normalization to eliminate redundancy**.
- In MongoDB, you model based on your application's access patterns.
- The primary goal is to structure your data in a way that your application can retrieve all the information it needs for a specific task in a single, efficient query.

Core principles of data modeling

- The Golden Rule: Store Data Together That Is Accessed Together
- Design for Your Application's Query Patterns
- Prefer Embedding for Relationships
- Use Referencing for Specific Scenario
- Strategic Denormalization is Encouraged

EMBEDDING

Prefer Embedding for Relationships

MongoDB's document model allows you to nest documents and arrays within a parent document. This is called embedding and is the default choice for modeling relationships.

- When to Embed:
  - **"Contains" relationships**: A blog post contains comments. An order contains line items.
    - **One-to-One (1:1)** relationships: A user has one profile record.
    - **One-to-Few (1:N)** relationships: A person has a few addresses or phone numbers. The "few" side is small and bounded.

Advantages of Embedding:

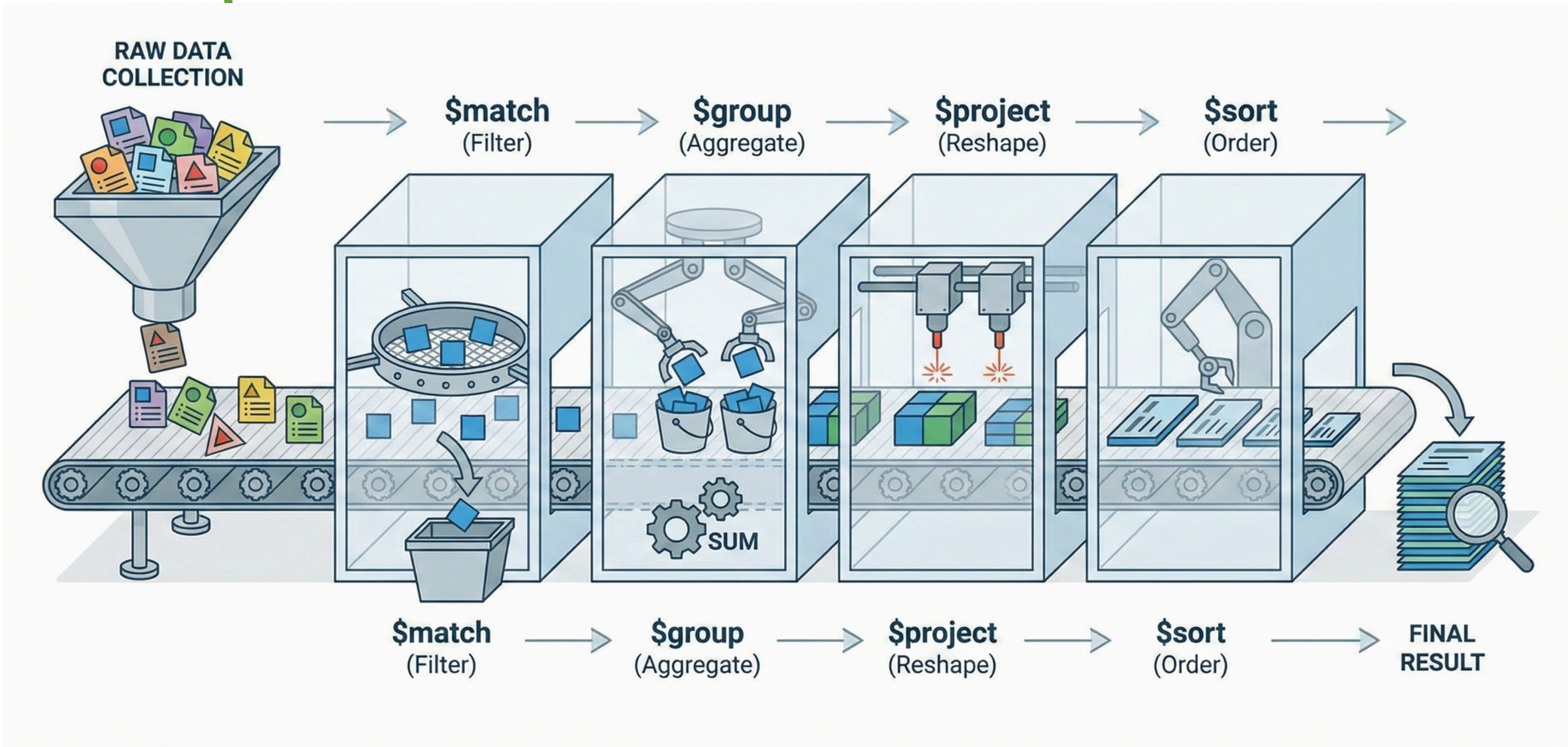
- Performance: You get all related data in a single read operation.
- Atomicity: MongoDB provides atomic writes at the document level. By embedding related data, you can update a parent and its children in a single, all-or-nothing operation without needing complex multi-document transactions.

REFERENCING

Use Referencing for Specific Scenarios

- When to Reference:
  - **One-to-Many (1:N)** with unbounded growth: A popular product could have millions of reviews. Embedding all reviews would make the product document massive and exceed MongoDB's 16MB document size limit.
  - **Many-to-Many (N:M)** relationships: Students enroll in many courses, and courses have many students. Referencing is the standard way to model this.
  - **Data is accessed independently:**
    - If you frequently need to query the "child" data without the "parent" data, referencing might be better.
  - **Data duplication is a major concern:**
    - If an entity (like a product category) is used in thousands of documents and changes frequently, embedding it would mean updating thousands of documents for a single change. Referencing avoids this.

AGGREGATION



AGGREGATION

What is Aggregation?

- Aggregation is the process of taking many distinct pieces of data (documents) and processing them together to produce a computed result.
- While a standard db.collection.find() query is used to fetch exact copies of documents that match criteria, aggregation is used to transform, summarize, and analyze data.
- Think of it like spreadsheet operations:
  - find() query: Filtering a spreadsheet to show only rows where "Status = 'Active'".
  - Aggregation: Taking that filtered data and creating a **Pivot Table** to show the total sales per region, ordered by highest revenue.
- **Common use cases for aggregation:**
  - Grouping data (e.g., "How many users signed up last month by country?")
  - Performing calculations (e.g., "What is the average order value?")
  - Reshaping documents (e.g., joining data from two collections, or flattening arrays).
  - Filtering and sorting based on calculated fields.

AGGREGATION-PIPELINE

The Aggregation Pipeline

- The Aggregation Pipeline is the primary framework MongoDB uses to perform aggregation tasks.
- The concept is based on a data processing "pipe" or an assembly line.
    - Documents enter the pipeline and pass through a series of stages.
- An aggregation pipeline is an array [] containing stage objects {}. Every stage starts with a $ character.

aggregate.js

```
db.orders.aggregate([
    // Stage 1
    {
    },
    // Stage 2
    {
    },
    // Stage 3
    {
    }
])
```

AGGREGATION-PIPELINE

Key Principles of the Pipeline:

1. **Sequential Processing**:
    a. Documents flow through stages in the specific order you define.
2. **Transformation**:
    a. Each stage performs a specific operation on the input documents and outputs transformed documents.
3. **Input/Output**:
    a. The output of one stage becomes the input for the next stage.
4. **No Persistent Changes**: The pipeline runs in memory and returns results to the client. It does not modify the actual data in the collection (unless you specifically use an output stage like $out or $merge as the very last step).

AGGREGATE OPERATORS

Stage	Description	SQL Equivalent
$match	Filters the documents to pass only those that match specified condition(s) to the next stage. **Ideally used as early as possible to reduce workload.**	WHERE
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group.	GROUP BY
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. Like a "SELECT" statement specifying which columns you want.	SELECT
$sort	Sorts all input documents and returns them to the pipeline in sorted order.	ORDER BY
$limit	Limits the number of documents passed to the next stage.	LIMIT
$skip	Skips over the specified number of documents and passes the rest along.	OFFSET
$unwind	Deconstructs an array field from the input documents to output a document for *each* element. (Explodes one document into many).	No direct SQL equiv. (sometimes CROSS JOIN)
$lookup	Performs a left outer join to another collection in the *same* database to filter in documents from the "joined" collection for processing.	LEFT JOIN

$MATCH

- **Purpose**:
  - Filters documents.
    - It decides which documents are allowed to move to the next stage of the pipeline.
  - If an item doesn't meet the criteria (e.g., it's damaged, or it doesn't have the right security badge), it gets tossed off the line immediately. Only good items continue.

aggregate.js

```
db.instructors.aggregate([
{
    $match : {age: {$gt:21}, lastName:{$exists:true}}
} , /// first stage

])
```

$SORT

- **Purpose**:
  - Reorders the documents in the stream based on specific fields.

aggregate.js

```
db.instructors.aggregate([
{
    $match : {age: {$gt:21}, lastName:{$exists:true}}
} ,
{
        $sort : {firstName:1 , lastName:1}
 }
])
```

$PROJECT

- **Purpose**:
  - Reshapes documents. You use it to select specific fields, rename fields, remove fields, or calculate entirely new fields based on existing data.

aggregate.js

```
db.instructors.aggregate([
    {
        $match: { age: { $gt: 21 }, lastName: { $exists: true } }
    },
    {
        $sort: { firstName: 1, lastName: 1 }
    },
    {

        $project: {
            fullname: { $concat: ["$firstName", ' ', "$lastName"] },
            age: 1,
            sal: "$salary",
            netsalary: { $multiply: ["$salary", .8] },
            //              _id:0 /// don't do this
        }
    }
])
```

$OUT

- **Purpose**:
  - Reorders the documents in the stream based on specific fields.

aggregate.js

```
db.instructors.aggregate([
    {
        $match: { age: { $gt: 21 }, lastName: { $exists: true } }
    }, /// first stage
    {
        $sort: { firstName: 1, lastName: 1 }

    },
    {
        $out: "instructors_info"

    }

])
```

$GROUP

- **Purpose**:
  - The $group stage collapses multiple documents into a single document based on a common value (an identifier). For each group, you can perform calculations like counting, summing, or averaging.

aggregate.js

```
db.instructors.aggregate(
    [
        {
            $match: { age: { $gt: 20 } }
        },
        {

            $group: {
                _id: "$age",  /// sepecify field of grouping
                total: { $sum: 1 },   /// $sum  ⟶  value 1 ⟶ count
                total_Ages: { $sum: "$age" },  /// $sum → age
                total_salary: { $sum: "$salary" },
                min_salary: { $min: "$salary" },
                max_salary: { $max: "$salary" },
                avg_salary: { $avg: "$salary" }
            }

        } // stage ⟶ no of instructor per age
    ])
```

$LOOKUP

- **Purpose**:
  - Performs a "Join" with another collection in the same database. It brings in related data from a different "table".

aggregate.js

```
db.students.aggregate([
    {
        $lookup: {
            from: "departments", /// collection name I need dept name from
            localField: "department",
            foreignField: "_id",
            as: "dept_info"
        }

    } /// lookup operator ⟶ return array

])
```

IMPORT-DATABASES

mongorestore is a command-line utility used to import database backups created by mongodump into a MongoDB instance.
It recreates the databases and collections from binary BSON files.

MONGORESTORE --DB <DATABASE_NAME> <PATH_TO_BACKUP_FOLDER>

Flag	Description
--uri="<connection_string>"	Used for remote or Atlas clusters (includes username/password).
--db <name>	Specifies which database to restore into.
--collection <name>	Specifies a single collection to restore (requires a .bson file).
--drop	**Important:** Drops every collection from the target database *before* restoring from the backup.
--nsInclude	Restores only specific namespaces (e.g., dbName.* or dbName.collectionName).
--gzip	Required if the original backup was compressed using the --gzip flag in mongodump.

INDEXING

Indexing is the most effective way to improve the performance of your MongoDB queries.

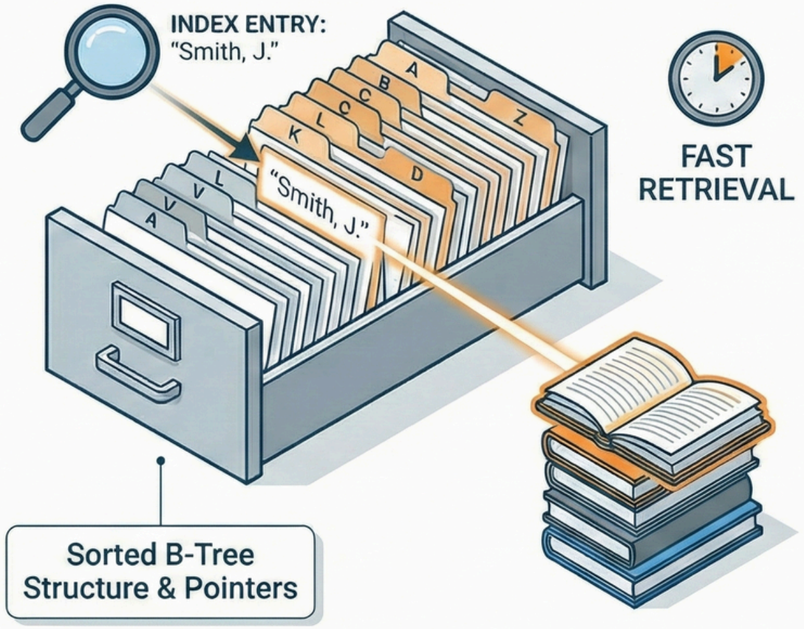
- Without indexes, MongoDB must perform a collection scan, meaning it has to look through every single document in a collection to find the ones that match your query.
- With an index, MongoDB can quickly narrow down the data by looking through a sorted list of values, much like an index in the back of a textbook.
- When you create an index, MongoDB builds a specialized data structure (a B-tree) that stores a small portion of the collection's data in a searchable, sorted form.
- The Trade-off:
  - While indexes make reads (queries) much faster,
  - they slightly slow down writes (inserts/updates/deletes)
  - This because MongoDB has to update the index every time the data changes.
  - They also consume disk space and RAM

INDEXING

MONGODB INDEX: The Textbook Analogy

WITHOUT INDEX
(Collection Scan)

WITH INDEX
(Index Scan)



INDEXING

Index Type	Purpose	Syntax Example
Single Field	The most basic index on a single key.	db.users.createIndex({ "email": 1 })
Compound	Indexing multiple fields together (order matters!).	db.users.createIndex({ "lastName": 1, "age": -1 })
Multikey	Used for indexing fields that contain arrays.	db.posts.createIndex({ "tags": 1 })
Text	Supports search for string content (search engine style).	db.articles.createIndex({ "content": "text" })
Geospatial	For location-based queries (coordinates).	db.places.createIndex({ "location": "2dsphere" })
Hashed	Used primarily for Sharding to ensure even data distribution.	db.users.createIndex({ "user_id": "hashed" })

The values 1 and -1 specify the sort order (Ascending and Descending). For single-field indexes, the direction doesn't usually matter, but for compound indexes, it can impact performance

INDEXING

Feature	Without Index	With Index
Query Method	Collection Scan (Slow)	Index Scan (Fast)
Data Access	Checks every document sequentially	Jumps directly to relevant data pointers
Sorting	Requires expensive in-memory operation	Uses pre-sorted index order automatically
Write Speed	Faster	Slower (must update index on every change)
Disk/RAM Usage	Lower	Higher

restore-index.js

```
db.product.createIndex({brand_name:1 })


db.product.dropIndex({brand_name:1})
```