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

Chapter 4

mongoDB®

Presented by: Dr. Brahim Benabderrahmane

Goals

Learning Objectives:

- Understand the principles and architecture of MongoDB.
- Manipulate and query data using MongoDB commands.
- Apply aggregation and indexing for analytics and optimization.
- Integrate MongoDB into a Big Data pipeline with Spark.

Table of Contents

01

Foundations of MongoDB

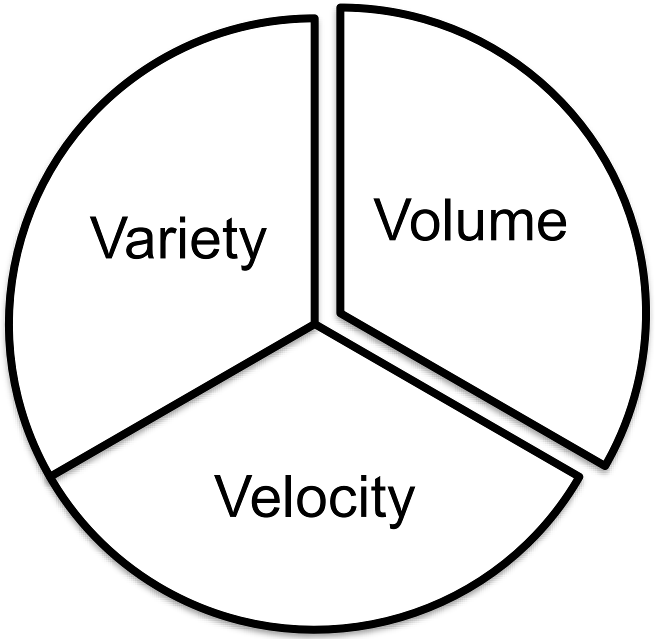
The Data Management Challenge

📈 **Explosion of data**:
- Volume, Variety, Velocity

🧱 **Limitations of traditional RDBMS**
- Rigid schema
- Expensive joins
- Difficult horizontal scaling

📚 **Need for flexible data models**
- Handle unstructured & semi-structured data

🌐 **Rise of NoSQL databases**
- Adapted to modern data workloads



Big Data Stack Recap

Hadoop manages **large-scale storage** and **batch workloads**.

Spark enables **real-time**, **in-memory**, and **iterative** computations.

Big Data architectures still require a database that scales and accepts **JSON-like**, dynamic structures.

MongoDB completes the ecosystem:

- Handles **unstructured / semi-structured** data
- Scales horizontally
- Integrates well with modern analytics stacks

Distributed storage (HDFS) & batch processing (MapReduce)

In-memory analytics, fast computation, unified processing

"Where do we store flexible, semi-structured data?"

Scalable, document-oriented, schema-flexible database

Families of NoSQL Databases

- **Key–Value Stores**
- **Document Stores**
- **Column-Family Stores**
- **Graph Databases**

NoSQL includes **multiple families**, each optimized for specific workloads.

- **Key-Value** → ultra-fast, simple caching/lookup systems.
- **Document** → flexible, JSON-like structures
- **Column-family** → high-throughput distributed storage.
- **Graph** → relationship-centric analytics.

MongoDB is a Document-oriented NoSQL database.

Why MongoDB?

A modern database for flexible, scalable data.

- **Document Model (BSON)**
  Stores data in JSON-like structures → natural for modern apps.
  example: { "name": "Alice", "age": 23 }

- **Flexible Schema**
  No rigid table structure; fields can adapt as data evolves.

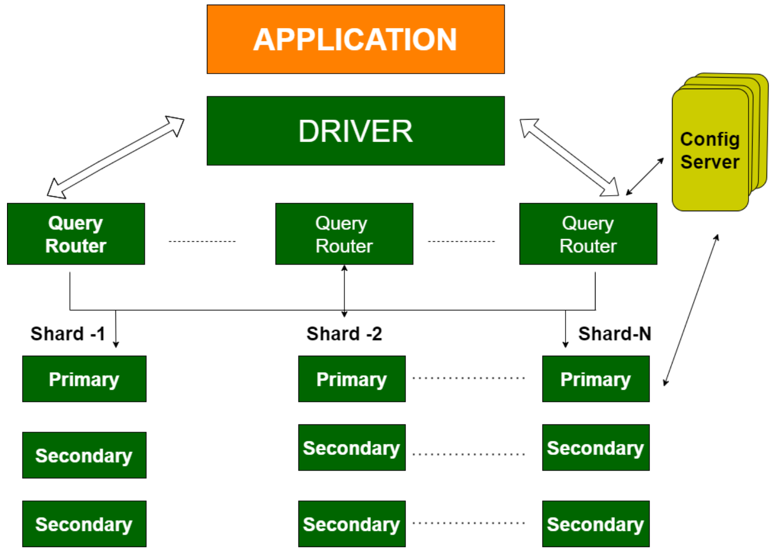
- **Scalable & Highly Available**
  Horizontal sharding + replica sets for fault tolerance.

- **Built for Modern Workloads**
  Great fit for web applications, IoT systems, and analytics pipelines.

MongoDB Architecture

Components:

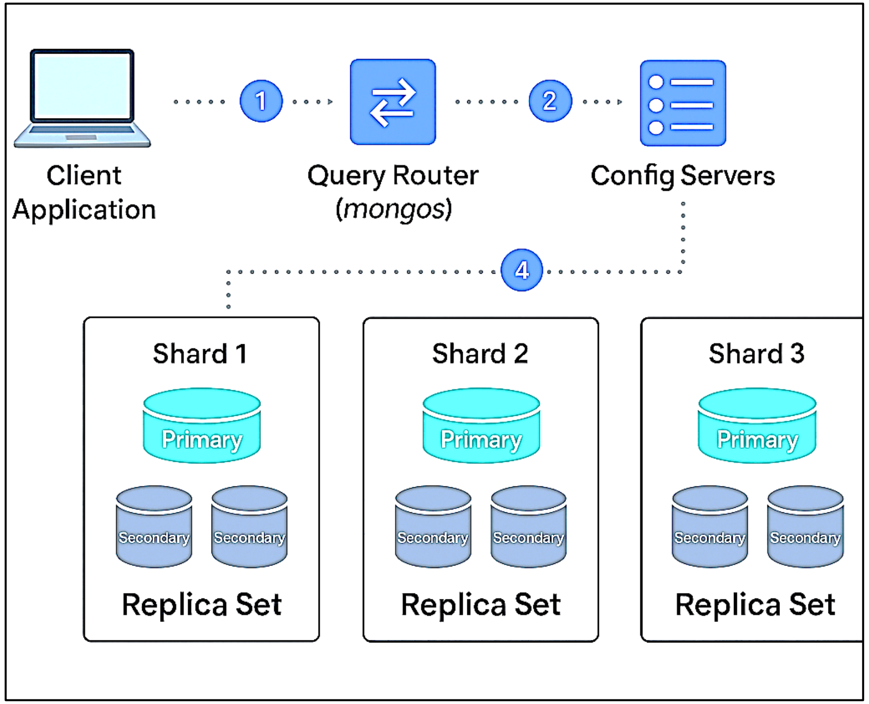
- **Client / Application**
  Sends queries and receives results via drivers.
- **MongoDB Server (mongod)**
  The main database process handling CRUD, indexing, queries.
- **Replica Set**
  Group of mongod processes providing **high availability** (Primary + Secondaries).
- **Config Servers**
  Store metadata and cluster configuration in sharded clusters.
- **Mongos Router**
  Distributes queries across shards; entry point for applications in sharded setups.
- **Shards**
  Each shard = replica set storing a portion of the data.



Sharding in MongoDB

MongoDB Architecture: How It Works

1. Clients send requests through **drivers**.
2. In a sharded cluster:
- Queries first reach **mongos** → routing layer.
- Mongos uses **config servers** to know where data lives.
3. Data is split across shards using the **shard key**.
4. Each shard is a **replica set** for high availability.
5. If the primary fails, a secondary takes over automatically.



BSON and JSON

JSON
Human-readable text format used for data exchange.

BSON (Binary JSON)

- Internal storage format used by MongoDB.
- Faster to parse (binary).
- More data types: **Date**, **32/64-bit integers**, **Decimal128**, **ObjectId**, **Binary**, etc.
- Optimized for **speed**, **indexing**, and **storage efficiency**.

How MongoDB Stores Data:

- Each document is stored internally as **BSON**.
- **Collections** = groups of BSON documents.
- MongoDB allocates storage in **"extents/pages"** and manages them via **WiredTiger.**

BSON enables:

- Fast traversal of documents (field offsets).
- Compact encoding of arrays and nested objects.
- Efficient comparison of typed values (important for indexing).

JSON (Text)

```
{
 "name": "Sara",
 "age": 25,
 "skills": ["Python", "AI"]
}
```

BSON (Concept View)
Binary blocks representation
Field types shown as tags:

- "name" → string
- "age" → int32
- "skills" → array<string>

Data Model: Embedding vs. Referencing

Embedding:

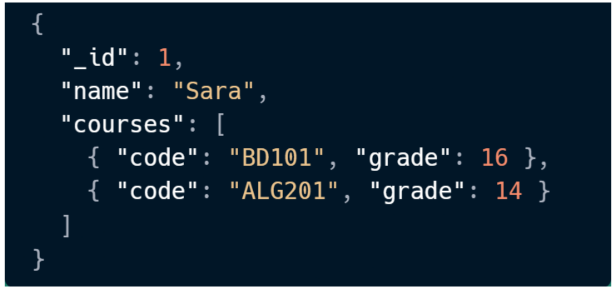
- Store related data **inside the same document**.
- **Fast reads**, fewer queries.
- Ideal when the related data is:
  - Small
  - Accessed together
  - Rarely updated independently

Referencing:

- Store related data in **separate documents**, linked by IDs.
- Reduces duplication, **better for large or frequently updated** subdocuments.
- Good for many-to-many or large sets (e.g., thousands of items).

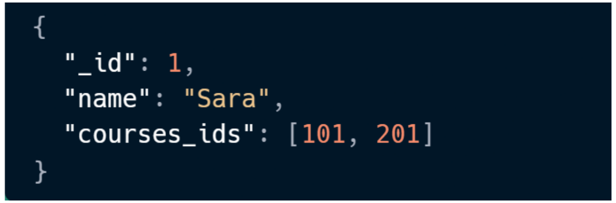
Key Line:
The choice depends on data size, relationship type, and read/write patterns.

Embedded Example:

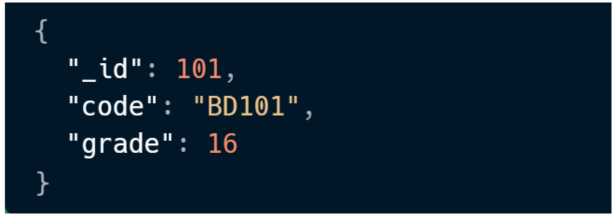


```
{
  "_id": 1,
  "name": "Sara",
  "courses": [
    { "code": "BD101", "grade": 16 },
    { "code": "ALG201", "grade": 14 }
  ]
}
```

Referenced Example:



```
{
  "_id": 1,
  "name": "Sara",
  "courses_ids": [101, 201]
}
```



```
{
  "_id": 101,
  "code": "BD101",
  "grade": 16
}
```

Basic MongoDB Commands

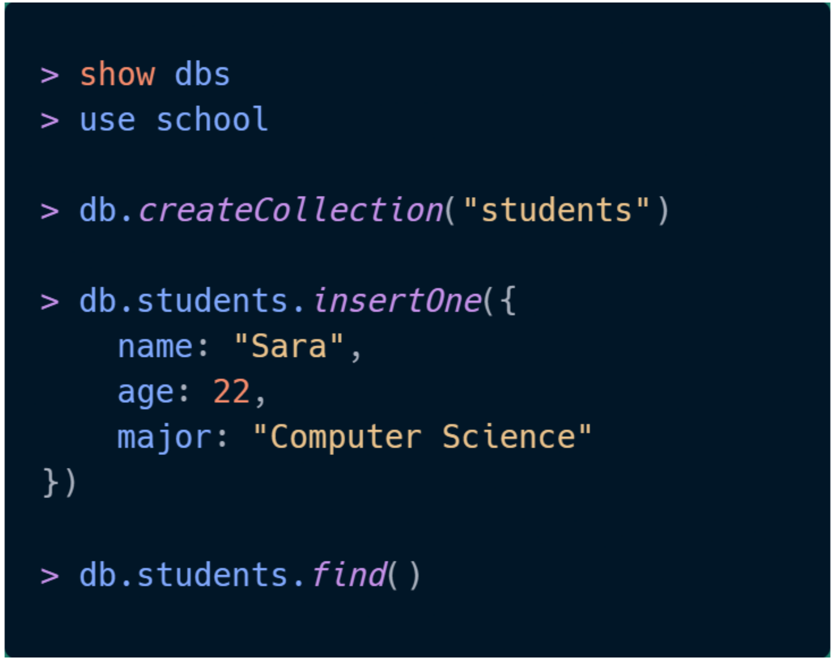
- **Listing databases**
  show dbs
  Displays all existing databases.

- **Switching / Creating a database**
  use school
  Creates school if it doesn't exist and switches to it.

- **Creating a collection**
  db.createCollection("students")
  Creates a new collection inside the current database.

- **Inserting a document**
  db.students.insertOne({ … })
  Adds one JSON/BSON document.

- **Querying data**
  db.students.find()
  Retrieves all documents.



```
> show dbs
> use school

> db.createCollection("students")

> db.students.insertOne({
    name: "Sara",
    age: 22,
    major: "Computer Science"
})

> db.students.find()
```

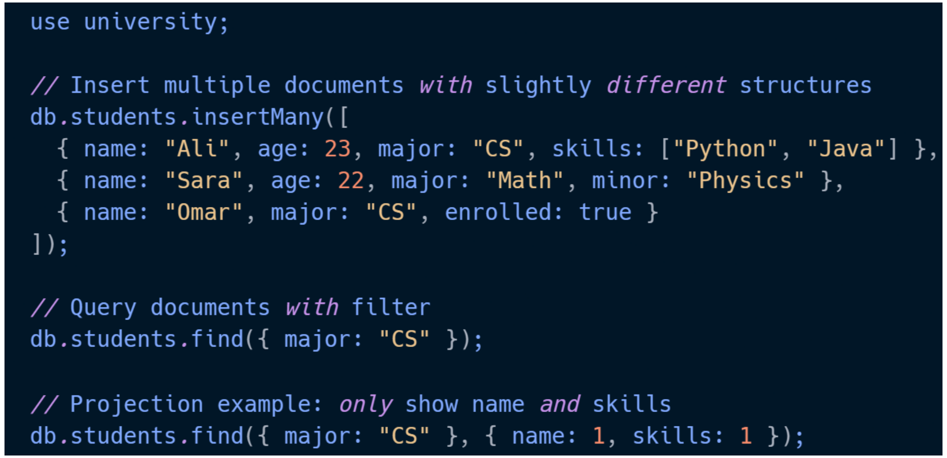
SQL vs. MongoDB Comparison

Relational databases use **tables, rows, and structured schemas**, while MongoDB stores data as **documents** with flexible, nested structures. The way we query and manipulate data reflects these fundamental differences.

Task	**SQL**	**MongoDB**
Create	INSERT INTO	insertOne()
Read	SELECT * FROM	find()
Update	UPDATE ... SET	updateOne()
Delete	DELETE FROM	deleteOne()

- **Schema Flexibility:**
  SQL requires predefined table schemas; MongoDB allows documents to evolve over time.
- **Data Model:**
  SQL uses relations and joins. MongoDB favors embedding or referencing inside JSON-like documents.
- **Query Approach:**
  SQL uses declarative statements (SELECT, INSERT, UPDATE).
  MongoDB uses JavaScript-like function calls (find(), insertOne(), updateOne()).
- **Scaling Philosophy:**
  SQL traditionally scales vertically.
  MongoDB is designed for **horizontal scaling** with sharding.

MongoDB in Action – Practical Overview



```
use university;

// Insert multiple documents with slightly different structures
db.students.insertMany([
  { name: "Ali", age: 23, major: "CS", skills: ["Python", "Java"] },
  { name: "Sara", age: 22, major: "Math", minor: "Physics" },
  { name: "Omar", major: "CS", enrolled: true }
]);

// Query documents with filter
db.students.find({ major: "CS" });

// Projection example: only show name and skills
db.students.find({ major: "CS" }, { name: 1, skills: 1 });
```

- MongoDB allows **different fields for different documents**.
- You can query with filters and retrieve only the fields you need.
- Collections are flexible: no table definition required.

Dynamic Schema in Action

- Each student document can have different fields: *skills, minor, enrolled.*
- MongoDB adapts to evolving data easily.

Simple CRUD Preview

- *insertOne / insertMany* : create
- *find* : read

Real-World Analogy

- Think of a collection as a folder of resumes
- Each resume can have different sections, but you can still filter/search them efficiently.

MongoDB in the Big Data Context

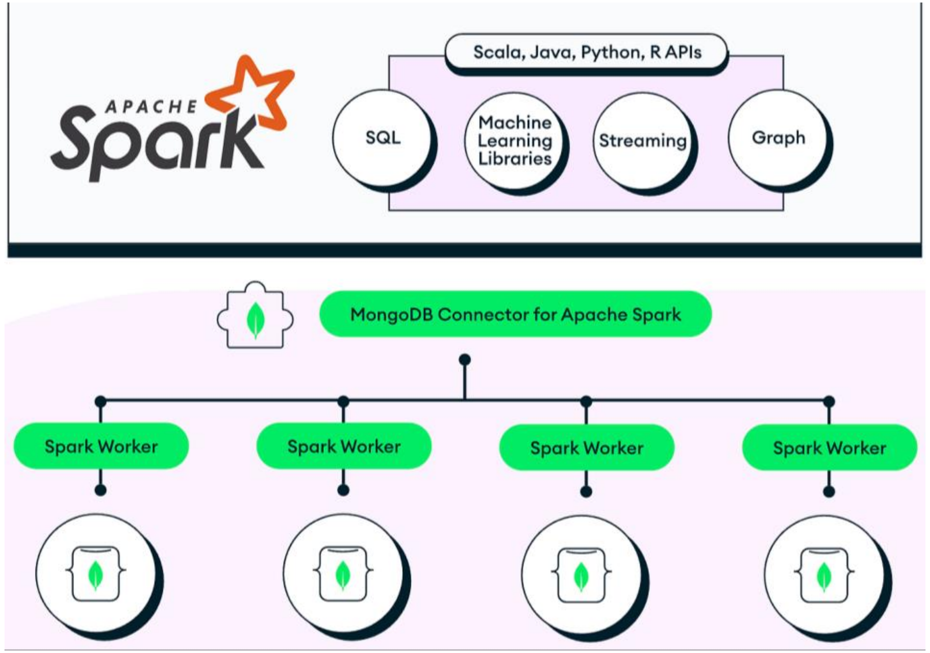
MongoDB complements **Hadoop and Spark** in modern Big Data architectures.

Used for **real-time or semi-structured data storage**.

Spark can read from and write to MongoDB, enabling analytics pipelines.

Part of a **Unified Analytics** ecosystem:

- Hadoop → storage
- Spark → processing
- MongoDB → flexible, schema-less database layer


APACHE Spark
Scala, Java, Python, R APIs
SQL
Machine Learning Libraries
Streaming
Graph
MongoDB Connector for Apache Spark
Spark Worker
Spark Worker
Spark Worker
Spark Worker

02

CRUD Operations

Database Creation

The *"TechStore"* Scenario

Why TechStore?

To explore MongoDB, we will build the backend for an online electronics retailer.

- **High Velocity:** Thousands of orders per minute.
- **Varied Data:** Complex product specs.
- **Scalability:** Needs to grow horizontally.

Modeling Goals

We will move away from rigid SQL tables to flexible JSON documents.

```
// Our Goal:
group_related_data: true,
optimize_for_read: true,
avoid_joins: true
```

The Big Picture: Schema Overview

Users
- Stores customer profiles, embedded addresses, and references to secure documents.

Products
- The inventory catalog using the "One-to-Few" pattern for specifications.

Orders
- Transactional records that use denormalization (snapshots) to preserve history.

Users Collection: 1:1 Relationships

```
  "_id": "user_101",
  "username": "sara_data",

  // Embedded 1:1
  "address": {
   "city": "Algiers",
   "zip": "16000"
  },

  // Referenced 1:1
  "kyc_doc_id": "doc_555"
}
```

Design Decisions

Embedded Address:

- We almost always need the address when loading the profile. Embedding avoids a join and improves read speed.

Referenced KYC:

- Sensitive documents are large and rarely accessed. Keeping them separate improves security and performance.

Products Collection: 1:N Relationships

```
{
  "_id": "prod_500",
  "name": "Laptop Pro",

  // Embedded 1:N (One-to-Few)
  "specs": [
    {"k": "RAM", "v": "32GB"},
    {"k": "CPU", "v": "i9"}
  ],

  // Referenced N:M
  "supplier_ids": [
    "supp_01", "supp_02"
  ]
}
```

The "One-to-Few" Pattern

- A product has a finite set of specifications (~10-20). It is much faster to read them inside the document than to join a separate "Attributes" table.

Referencing

- Suppliers are distinct entities. We reference them to keep data normalized.

Suppliers: Many-to-Many Network

Bidirectional Referencing

In the **suppliers** collection, we also store references to products.

```
{
  "_id": "supp_01",
  "company": "TechDistro",
  "supplied_product_ids": [
    "prod_500",
    "prod_501"
  ]
}
```

This allows us to quickly query:

"Give me all products sold by TechDistro"

without scanning the entire Products collection.

Orders: The Snapshot Pattern

We use **Denormalization** to preserve data integrity over time.

```
"items": [
  {
    "product_id": "prod_500",
    "name": "Laptop Pro",
    "price": 2500.00  ← // FROZEN
  }
]
```

Critical Logic:

We embed product details *as they were at the moment of purchase*. If the product price changes next week, this historic order record remains accurate.

Summary of Relationships

Relationship Type	Implementation	Why?
One-to-One	User Address (Embed)	Speed (read together)
One-to-One	User KYC (Reference)	Security & Performance
One-to-Many	Product Specs (Embed)	Efficiency (small, static list)
One-to-Many	User Orders (Reference)	Scalability (unbounded growth)
Many-to-Many	Order Items (Embed)	**Data Integrity (Snapshot)**

CRUD Operations in Python

Thinking in Dictionaries

We interact with MongoDB using standard Python dictionaries.
*Note the switch to **snake_case**.*

Operation	SQL (Relational)	Python (PyMongo)
Create	INSERT INTO...	db.users.insert_one({...})
Read	SELECT * FROM...	db.users.find({...})
Update	UPDATE...	db.users.update_one({...})
Syntax	SQL Syntax	Python Dictionaries

1. Create: Flexible Schema

In Python, we simply pass a dictionary. Note that **keys must be quoted strings**.

Scenario:
Inserting a "Refurbished" laptop with unique fields.

```
db.products.insert_one({
"name": "Gaming Laptop (Refurbished)",
"price": 1800.00,
"category": "Laptops",
# Python specific: None, True, False
"is_new": False,
"warranty_void": None,
# Unique fields for this item
"refurbished_date": "2024-01-15",
"previous_repairs": ["Screen", "Battery"]
})
```

2. Read: Nested Fields

```
# Find users living in Algiers
 db.users.find({ "address.city“ : "Algiers" })

# Find products with specific specs
# No JOIN needed!
db.products.find({ "specifications.key“ : "RAM", "specifications.value": "32GB" })
```

DOT NOTATION

Drilling Down
Even in Python, when querying **inside** a dictionary or array, we must use the dot string syntax: "address.city".

Iteration
The find() method returns a **Cursor** (an iterator), not a list. To see results, you loop over it or use list(cursor).

Read: Advanced Filtering

Read: Advanced Filtering

Comparison & Logic
We use MongoDB operators (starting with $) inside our Python dictionary.

- **$gt** / **$lt**: Greater/Less Than
- **$in**: Match any value in a list
- **$and** / **$or**: Logical combinations

```
# Find expensive Gaming Laptops
db.products.find({ "$and": [ { "price": { "$gt": 2000 } }, { "tags": "gaming" } ] })
# Find orders with either 0 OR None items
 db.orders.find({ "items": { "$in": [0, None, []] } })
```

3. Update: Atomic Modifiers

```
# 1. Update the price (Set)
# 2. Mark as 'On Sale' (Set)
# 3. Reduce stock by 1 (Increment)
db.products.update_one( { "_id": "prod_500" },  # Filter
{
"$set": { "price": 2200.00, "on_sale": True },
"$inc": { "stock_count": -1 }
} )
```

Precision Updates

Warning: Always use operators like $set.

- update_one: Takes two arguments: the **Filter** and the **Update** dictionary.
- The keys for operators (like $set) must be quoted strings in Python.

Update: Managing Array

Managing lists in MongoDB via Python is powerful. No need to pull the data, edit the list in Python, and save it back.

```
# Add a new supplier to the list
db.products.update_one( { "_id": "prod_500" }, { "$push": { "supplier_ids": "supp_99" } } )

# Add 'gaming' tag ONLY if not already there
db.products.update_one( { "_id": "prod_500" }, { "$addToSet": { "tags": "gaming" } } )
```

- **$push:** Appends item.
- **$addToSet:** Acts like a Python Set (no duplicates).

4. Delete: Removing Data

Targeted Removal
Standard methods: delete_one and delete_many.

Danger Zone: delete_many({}) will wipe the entire collection!

```
# Delete a specific order
db.orders.delete_one({ "_id": "ord_999" })

# Delete all 'Cancelled' orders older than 1 year
db.orders.delete_many({ "status": "cancelled", "order_date": { "$lt": "2023-01-01" } })
```

Summary

Python + MongoDB

Snake Case: Methods are insert_one, find_one, etc.
Types: Use Python natives: None, True, False, datetime.
Quotes: Operator keys like "$set" must be quoted.

03

Aggregation & Indexing

The Analytics Challenge

Why isn't find() enough?

Standard CRUD is for **Transactional** data (fetching a profile, creating an order).

Analytical questions require transforming data:

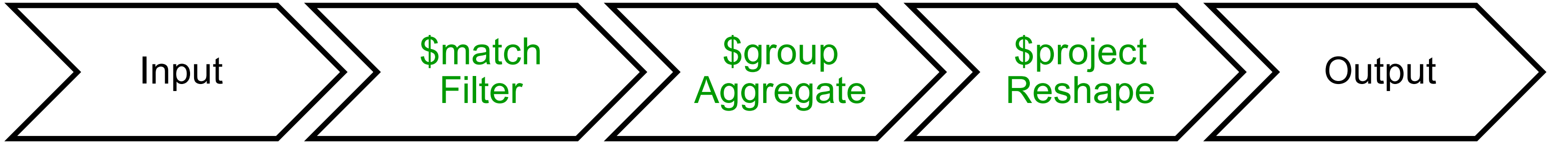
- "What is my monthly revenue?"
- "Who are my top 10 customers by spend?"
- "What is the average order size per category?"

```
# find() output (Raw):
{ "id": 1, "total": 100, "date": "Jan 1" }
{ "id": 2, "total": 200, "date": "Jan 2" }
{ "id": 3, "total": 150, "date": "Feb 1" }

# What we ACTUALLY want:
{ "month": "January", "revenue": 300 }
{ "month": "February", "revenue": 150 }
```

The Pipeline Framework

Aggregation works like a Unix pipe (|). Documents enter, get processed by a stage, and passed to the next.



In Python, this is defined as a list of stage dictionaries.

Stage 1: $match (Filter)

The First Rule of Aggregation

Always filter early.

Using $match at the start of your pipeline reduces the number of documents the next stages have to process.

It takes advantage of **Indexes** (unlike later stages).

```
pipeline = [
  # Keep only 'Completed' orders from 2024
  {
    "$match": {
      "status": "completed",
      "order_date": {
        "$gte": "2024-01-01"
      }
    }
  }
  # ... next stages ...
]
```

Stage 2: $project (Reshape)

```
{
  "$project": {
    "user_id": 1,
    "_id": 0,

    # Rename a field
    "final_cost": "$total_amount",

    # Calculate a new field (Tax 19%)
    "tax_amount": {
      "$multiply": ["$total_amount", 0.19]
    }
  }
}
```

More than just Selecting
$project (or $addFields) allows you to:

- Select specific fields (1 to keep, 0 to hide).
- Rename fields to be more readable.
- Compute new values (Arithmetic, String manipulation).

Stage 3: $group (Accumulate)

The Pivot Table

This stage collapses multiple documents into one.

- **_id:** The "Bucket" (What are we grouping by?).
- **Accumulators:** How do we summarize the data in that bucket?

Common Accumulators: $sum, $avg, $min, $max, $push (create array).

```
{
  "$group": {
    # Bucket: Group by Customer
    "_id": "$user_id",

    # Metric: Total Spent
    "total_revenue": {
      "$sum": "$total_amount"
    },

    # Metric: Average Order Value
    "avg_basket": {
      "$avg": "$total_amount"
    }
  }
}
```

Stage 4: $unwind (Explode)

```
# Document 1 (Before)
{ "order": 1, "items": ["Laptop", "Mouse"] }

# Stage:
{ "$unwind": "$items" }

# Output (Becomes 2 Docs)
{ "order": 1, "items": "Laptop" }
{ "order": 1, "items": "Mouse" }
```

Why is this necessary?

In **TechStore**, items is an array inside orders.
If we want to calculate **"Total Revenue per Product"**, we cannot group by the array itself.

We must first "explode" or "unwind" the array so that each product becomes its own document, then we can group by product.

Stage 5: $lookup (Joins)

Left Outer Join

Even in NoSQL, sometimes we need to join data.
$lookup pulls data from another collection.

Scenario: We have an aggregation of "Top Spenders" (User IDs), but we need their real Names and Emails from the users collection.

```
{
  "$lookup": {
    "from": "users",        # Target Collection
    "localField": "_id",    # Field in Input (User ID)
    "foreignField": "_id", # Field in Target
    "as": "user_details"    # Output Array Field
  }
}
```

Advanced: Date Analytics

In Analytics, we rarely group by exact timestamp. We group by Day, Month, or Year.

```
{
  "$group": {
   "_id": {
     # Extract formatting on the fly
     "month": { "$month": "$order_date" },
     "year": { "$year": "$order_date" }
   },
   "monthly_sales": { "$sum": "$total_amount" }
  }
 }
```

Full Pipeline: Monthly Revenue

```
pipeline = [
 # 1. Match: Only Completed Orders
 { "$match": { "status": "completed" } },

 # 2. Group: By Year-Month
 {
  "$group": {
   "_id": {
    "format": { "$dateToString": { "format": "%Y-%m", "date": "$order_date" } }
   },
   "total": { "$sum": "$total_amount" }
  }
 },

 # 3. Sort: Oldest to Newest
 { "$sort": { "_id.format": 1 } },

 # 4. Out: Save to new collection (Optional)
 { "$out": "monthly_reports" }
]
```

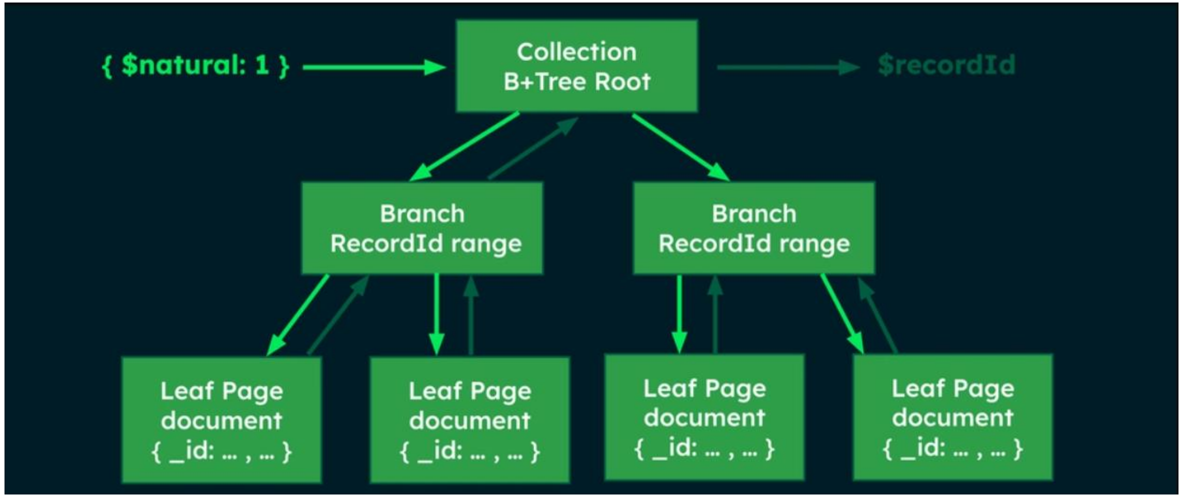
Part2 : Performance & Indexing

Making Queries Fast

Indexing: The Concept

The Collection Scan (Slow)
Without an index, MongoDB scans every document (COLLSCAN). O(N) complexity.

The B-Tree Index (Fast)
An index stores a small portion of the data in an ordered tree structure. MongoDB traverses the tree to find the document reference. O(log N) complexity.


{ $natural: 1 }
Collection B+Tree Root
$recordId
Branch RecordId range
Branch RecordId range
Leaf Page document { _id: ... , ... }
Leaf Page document { _id: ... , ... }
Leaf Page document { _id: ... , ... }
Leaf Page document { _id: ... , ... }

Creating Indexes

```
# Single Field Index (Ascending)
db.users.create_index([("username", 1)])

# Compound Index
db.products.create_index([
  ("category", 1),
  ("price", -1)
])

# Multikey Index (Array)
db.products.create_index([("tags", 1)])
```

Types of Indexes

- **Single Field:** Basic index on one key.
- **Compound:** Index on multiple fields. Critical for queries that filter on A and Sort by B.
- **Multikey:** Automatically created when you index an array (like tags). It creates an index entry for each element in the array.

Strategy: The ESR Rule

When creating Compound Indexes, the order of fields matters immensely.

Follow the **ESR Rule**:

- **E (Equality):** Place fields you run exact matches on first.
  category: "Laptop"

- **S (Sort):** Place fields you sort by next.
  sort({ price: -1 })

- **R (Range):** Place fields you range filter on last.
  price: { $gt: 500 }

```
# Ideal Index for: Find Laptops, Sort by Price, Price > 500
create_index([ ("category", 1), ("price", -1) ])
```

Debugging: explain()

Always check if your query is using the index you created.

```
db.products.find({...}).explain("executionStats")
```

Metric	Value	Verdict
totalDocsExamined	10,000	High number = Bad (Scanning too much)
nReturned	5	Examined 10k to return 5? **Missing Index!**
stage	IXSCAN	**GOOD** (Index Scan)
stage	COLLSCAN	**BAD** (Full Scan)

The Cost: Writes vs. Reads

No Free Lunch!

- Indexes make READS faster, but they make WRITES slower.

Why?

- Every time you insert, update, or delete a document, MongoDB must also update every B-Tree index associated with that collection.

Rule of Thumb:

- Only index patterns you actually query frequently.

Pattern: Computed Pattern

Problem

- Calculating "Total Spend" for a user requires summing up all their Orders every time (CPU intensive).

Solution

- Compute the value on write and store it in the User document.

```
# User Document
{
  "_id": "user_101",
  "username": "ahmed",

  # Pre-calculated field!
  # Updated whenever a new order is
placed.
  "total_spend": 5400.00,
  "orders_count": 12
}
```

Pattern: Subset Pattern

Problem
A product has 1,000 reviews.
Embedding all of them makes the document huge (RAM waste), but referencing them means extra queries.

Solution
Embed the Subset (Top 5 reviews) for fast display, and reference the rest.

```
{
  "product": "Laptop",

  # Only the 5 most recent
  "reviews_subset": [
    { "user": "A", "rating": 5 },
    { "user": "B", "rating": 4 }
  ],

  # Link to the full collection
  "reviews_ref": "prod_101_reviews"
}
```

Lecture Summary

Aggregation

- Pipeline: Match -> Group -> Project.
- Use $unwind for arrays.
- Use $lookup for joins.

Performance

- **ESR Rule:** Equality, Sort, Range.
- **Explain:** Avoid COLLSCAN.
- **Patterns:** Computed & Subset patterns optimize reads.

04

Advanced Topics & Integration

Replication, Sharding, and the Spark Connector

MongoDB in the Ecosystem

In a modern "Polyglot Persistence" architecture, we use the right tool for the job.

Hadoop / S3
Cheap Storage / Archive

Apache Spark
Heavy Processing / ML

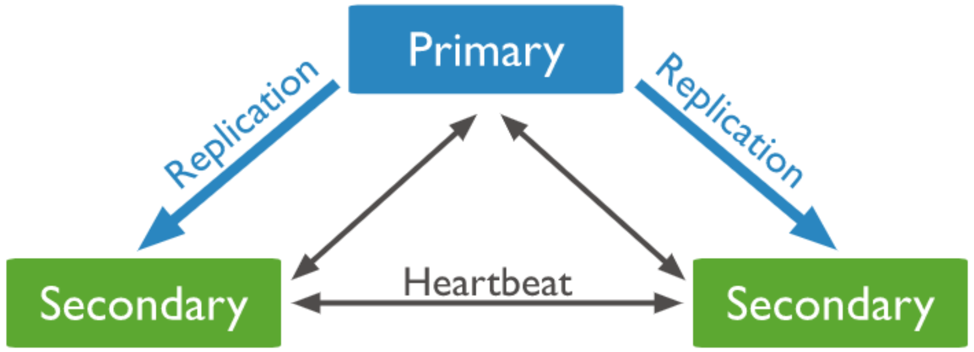
MongoDB
Serving Layer / App DB

Replication: High Availability

SAFETY FIRST!!

Replica Sets:
A group of `mongod` processes maintaining the same dataset.

- **Primary:** Receives all writes.
- **Secondaries:** Replicate operations from the Primary's *Oplog* (Operations Log).
- **Automatic Failover:** If Primary dies, Secondaries hold an election to pick a new leader.

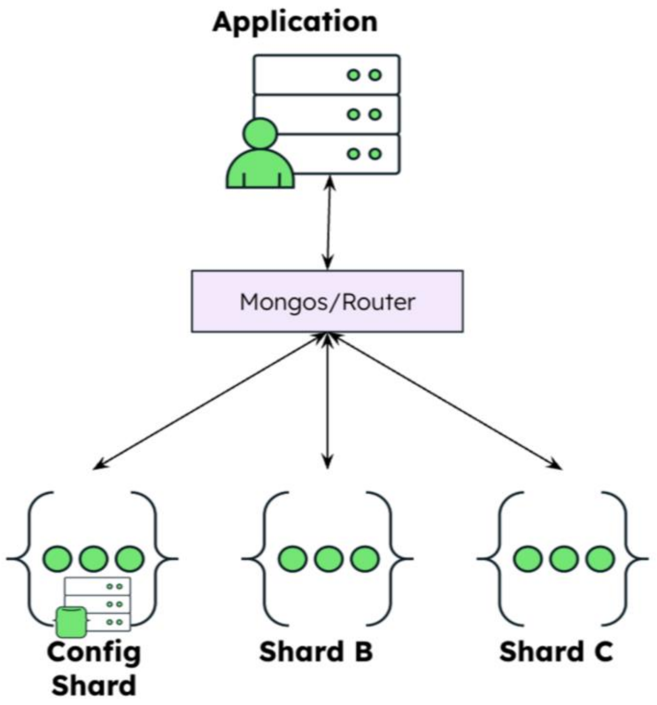


Sharding: Horizontal Scaling

PERFORMANCE FIRST

Partitioning Data:
When data exceeds the storage capacity of a single machine, we split it across multiple servers (Shards).

- **Shard Key:** The field used to distribute data (e.g., `user_id` or `region`).
- **Mongos Router:** Acts as a traffic cop, directing queries to the correct shard based on the key.



ACID Transactions

Multi-Document ACID

Historically, NoSQL only guaranteed atomicity per document. Modern MongoDB (4.0+) supports SQL-style transactions across multiple documents.

Use Case: Moving money between two bank accounts (Debit A, Credit B). Both must succeed, or both fail.

```
session = client.start_session()
with session.start_transaction():
    # 1. Debit Alice
    db.accounts.update_one(
        {"id": "A"},
        {"$inc": {"balance": -100}},
        session=session
    )

    # 2. Credit Bob
    db.accounts.update_one(
        {"id": "B"},
        {"$inc": {"balance": 100}},
        session=session
    )
# Auto-commit on exit
```

Spark Integration

The **MongoDB Spark Connector** allows Spark to treat MongoDB as a Data Source. It is intelligent: it pushes filters down to MongoDB to minimize data transfer.

```
# 1. Reading from MongoDB
df = spark.read.format("mongodb") \
    .option("uri", "mongodb://127.0.0.1/shop.orders") \
    .load()

# 2. Processing in Spark
# The filter is pushed down!
big_orders = df.filter(df["total"] > 1000)

# 3. Writing Back
big_orders.write.format("mongodb") \
    .option("uri",
"mongodb://127.0.0.1/shop.vip_orders") \
    .mode("append") \
    .save()
```

Why use this?

- Use Spark MLlib on data stored in Mongo.
- Offload complex aggregations that exceed Mongo's RAM.
- Move archival data from Mongo to HDFS/S3 (Parquet).

Cloud: MongoDB Atlas

Database as a Service (DBaaS)

Instead of managing servers manually, Atlas provides a fully managed cluster on AWS, Azure, or GCP.

- **Auto-Scaling:** Adds shards automatically when load increases.
- **Global Clusters:** Put data near users (e.g., EU users get data from Frankfurt node).
- **Serverless:** Pay only for operations, not idle time.

Production Readiness

Domain	Mechanism	Best Practice
Authentication	SCRAM-SHA-256 / x.509	Never use `localhost` without auth in prod.
Authorization	Role-Based Access Control (RBAC)	Grant `readWrite` only on specific DBs.
Network	TLS / SSL Encryption	Encrypt data in transit.
Backup	Ops Manager / Cloud Backup	Continuous point-in-time recovery (Oplog).

Chapter Summary

We have covered:

- **Modeling:** Embedding vs. Referencing.
- **Querying:** CRUD & Aggregation Pipelines.
- **Performance:** Indexing & Sharding.
- **Integration:** Using MongoDB with Spark.

Practically:

- **TD Session:** Designing the schema for the "TechStore".
- **TP Session:** Writing Python scripts to connect Spark and MongoDB.

End of Chapter 4