MONGO DB

CONTENTS

01

MongoDB Fundamentals & CRUD’s

SQL (Relational) Databases

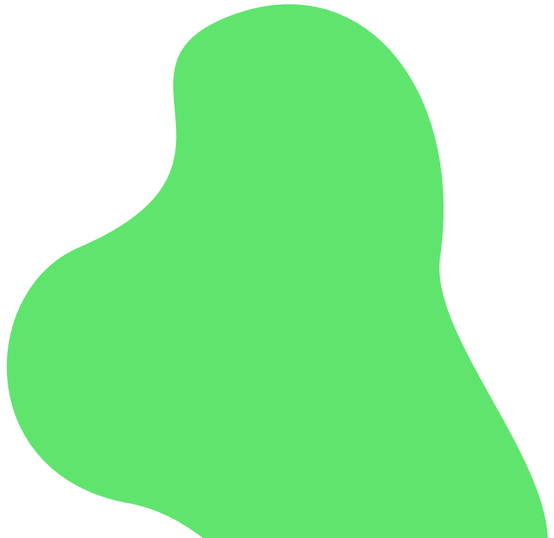
Users		
id	email	name
1		
2		
3		
4		

Blog Posts		
id	title	author
1		
2		
3		
4		

SQL (Relational) Databases

SELECT * FROM authors

query all records from the authors table

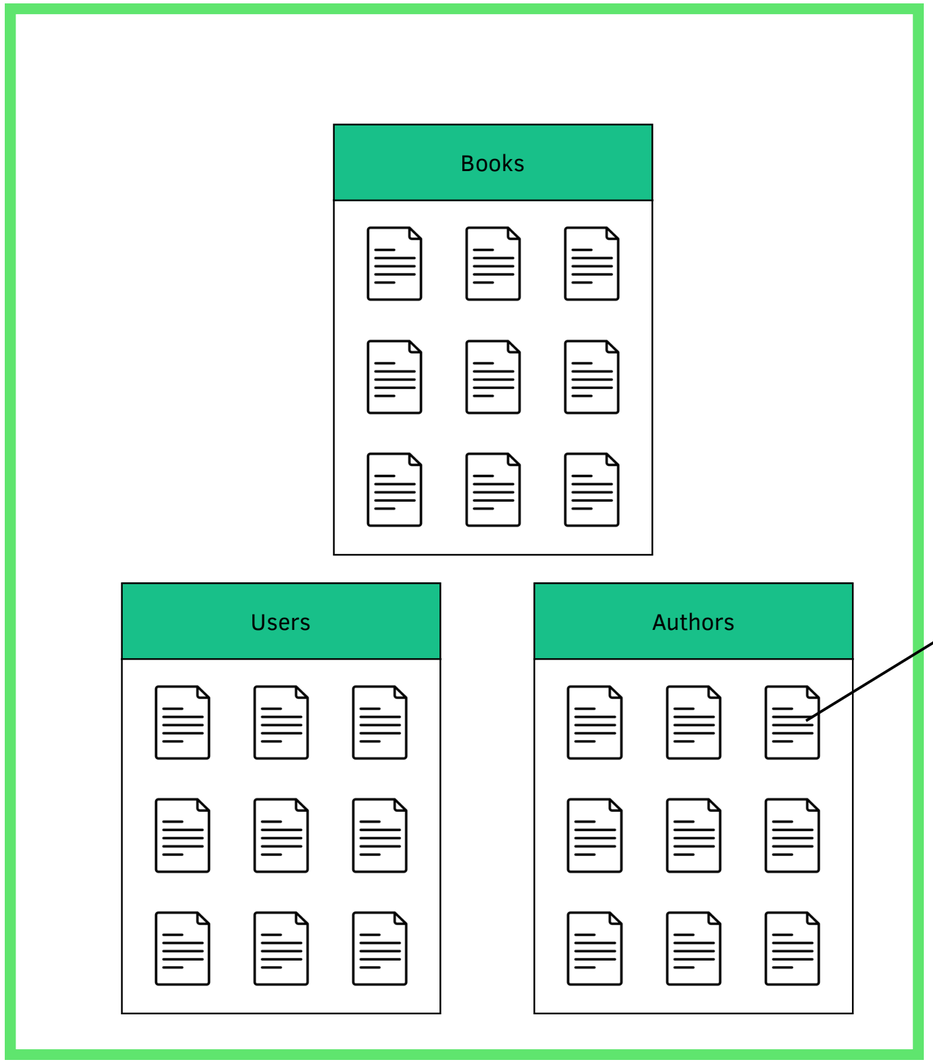


NoSQL Database (MongoDB)

NoSQL Database (MongoDB)

NoSQL Database (MongoDB)

NoSQL Database (MongoDB)




```
{
    "author": {
        "firstName": "John",
        "lastName": "Doe",
        "email": "john.doe@example.com",
        "bio": "A passionate writer",
        "publications": [
            {
                "title": "The Art of Writing",
                "year": 2020
            }
        ],
        "genres": [
            "Fiction",
            "Poetry"
        ]
    }
}
```

NoSQL vs. SQL

Feature	SQL (Relational DB)	NoSQL (Non-Relational DB)
Data Model	Structured, tabular	Flexible, document-oriented
Schema	Predefined schema	Dynamic schema
Scalability	Vertical (Scaling up)	Horizontal (Scaling out)
Querying	SQL queries (JOINs)	Varies (Key-value, document-based)
Use Cases	Banking, ERP	Real-time analytics, social media

How MongoDB Stores Data (BSON vs JSON)

What is JSON?

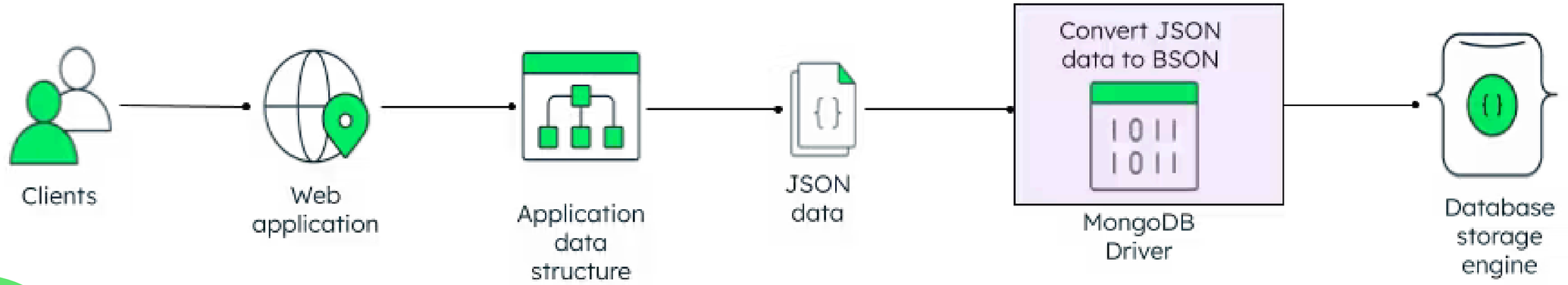
- Text-based, human-readable data format.
- Used for data exchange in web APIs.
- Limited data types: Strings, Numbers, Arrays, Booleans, Null, Objects.

What is BSON (Binary JSON - Used in MongoDB)?

- Binary format for efficient storage & retrieval.
- Supports more data types than JSON.
- Indexes & query optimization make BSON faster.

Data Type Differences: JSON vs BSON

Feature	JSON	BSON (MongoDB)
Format	Text-based	Binary
Speed	Slower processing	Faster queries & storage
Size	Larger due to text overhead	More compact
Supports Dates?	Stored as a string	Native `ISODate` type
Supports Object ID?	No	`ObjectId` (unique identifier)
Supports Binary Data?	No	Yes (`BinData` for images/files)
Supports GeoJSON?	No	Yes (for geospatial data)

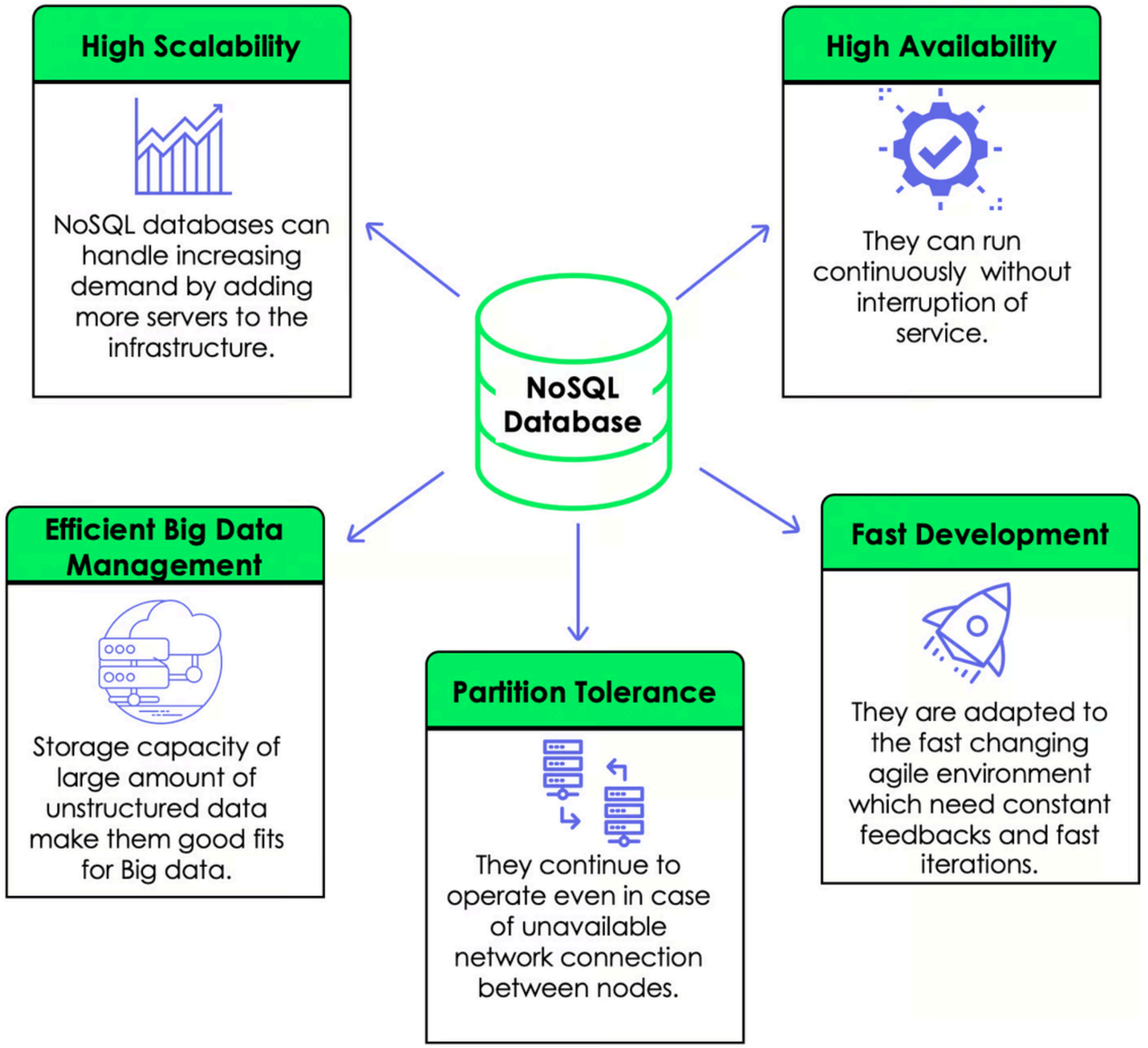

Clients
Web application
Application data structure
JSON data
Convert JSON data to BSON
MongoDB Driver
Database storage engine

NoSQL

What is NoSQL?

- NoSQL stands for "Not Only SQL."
- A non-relational database designed for flexibility, scalability, and high performance.
- Optimized for handling large volumes of unstructured or semi-structured data.

Why NoSQL?


High Scalability
NoSQL databases can handle increasing demand by adding more servers to the infrastructure.
High Availability
They can run continuously without interruption of service.
NoSQL Database
Efficient Big Data Management
Storage capacity of large amount of unstructured data make them good fits for Big data.
Partition Tolerance
They continue to operate even in case of unavailable network connection between nodes.
Fast Development
They are adapted to the fast changing agile environment which need constant feedbacks and fast iterations.

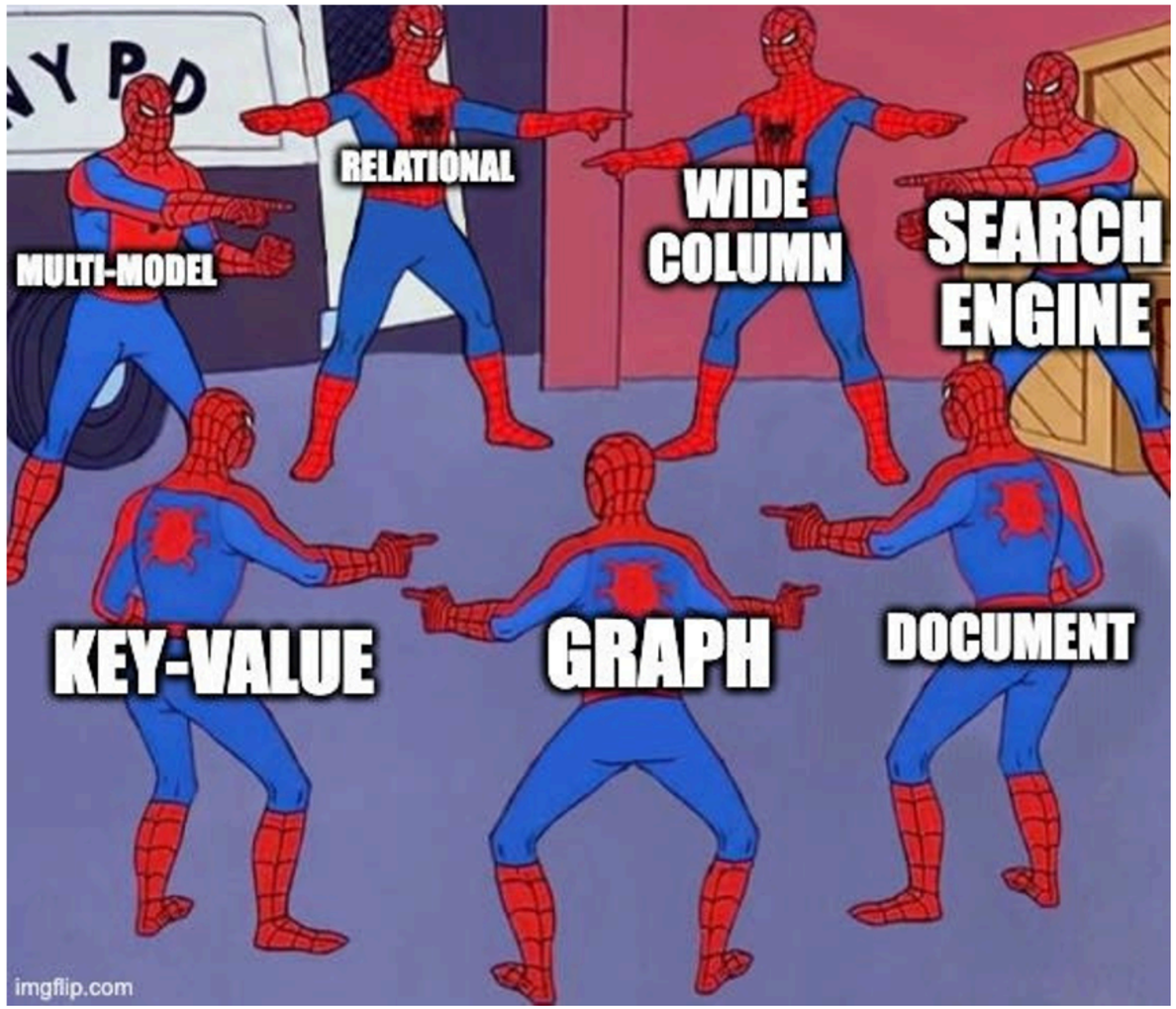
Types of NoSQL Databases

There are many types of noSQL databases , some of them are :

1. **Document-Based**: Stores JSON-like documents (e.g., MongoDB, CouchDB).
2. **Key-Value Stores**: Fast key-based lookups (e.g., Redis, DynamoDB).
3. **Column-Family Stores**: Stores data in columns instead of rows (e.g., Apache Cassandra, HBase).
4. **Graph Databases**: Stores relationships efficiently (e.g., Neo4j, ArangoDB).


KEY
VALUE
KEY
VALUE
KEY
VALUE
KEY
VALUE
KEY-VALUE DATABASE
DOCUMENT DATABASE
NoSQL
COLUMN-ORIENTED DATABASE
GRAPH DATABASE

Types of NoSQL Databases



DB vs DBMS vs ORM/ODM

- Database (DB)

- A structured collection of data.
- Stores and organizes information for easy access.
- Examples: MongoDB, MySQL, PostgreSQL

- Database Management System (DBMS)

- Software that interacts with the database to manage data.
- Provides tools for CRUD operations (Create, Read, Update, Delete).
- Ensures security, performance, and integrity of data.
- Examples: MongoDB Compass, MySQL Server, PostgreSQL DBMS

- ORM/ODM

ORMs and ODMs simplify database interactions by allowing developers to use object-oriented code instead of raw queries, improving readability, maintainability, and reducing errors.

ORM vs ODM

Feature	ORM (Object-Relational Mapper)	ODM (Object-Document Mapper)
Works With	**Relational Databases (SQL)**	**NoSQL Databases (MongoDB)**
Data Model	Tables (Rows & Columns)	Documents (JSON-like format)
Query Language	SQL Queries	MongoDB Query Language
Libraries	Sequelize (Node.js), TypeORM	Mongoose (MongoDB ODM)

MongoDB Compass

MongoDB Compass is a GUI tool for interacting with MongoDB visually.

- Allows users to explore and analyze data without writing complex queries.
- Features:
  - Connect to local or cloud MongoDB instances.
  - Browse collections and documents.
  - Perform CRUD operations with a user-friendly interface.
  - Monitor database performance and visualize query execution.

CRUD Operations

CRUD stands for Create, Read, Update, Delete – the four basic operations in MongoDB.

1. **Create** – Insert documents into a collection.

```
db.users.insertOne({"name": "John", "age": 30})
```

2. **Read** – Retrieve documents from a collection.

```
db.users.find({"name": "John"})
```

3. **Update** – Modify existing documents.

```
db.users.updateOne({"name": "John"}, {"$set": {"age": 31}})
```

4. **Delete** – Remove documents from a collection.

```
db.users.deleteOne({"name": "John"})
```

let’s dive in

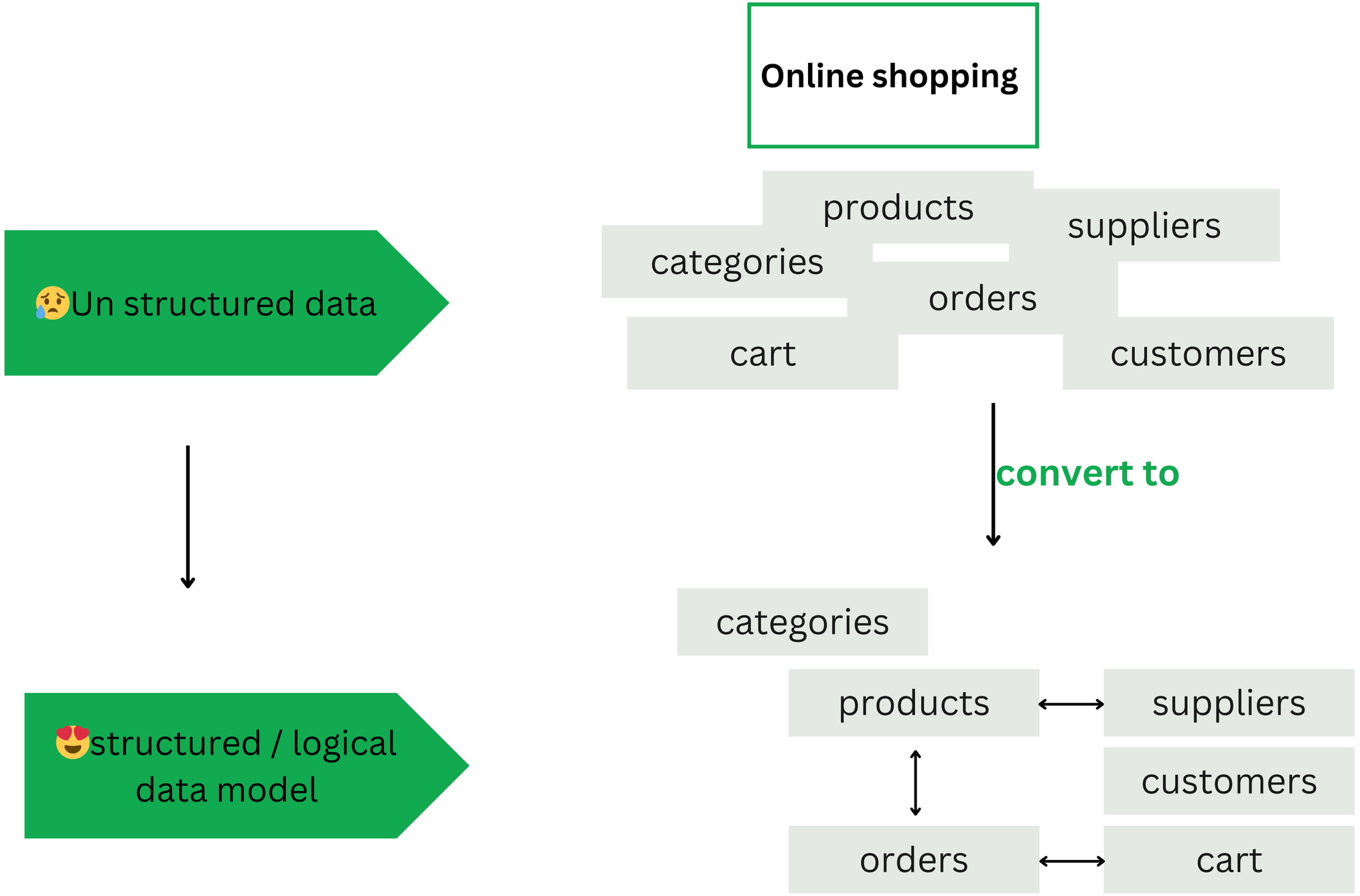
Questions ?

02

Data modeling & Indexes

Data Modeling

Data Modeling is the process of designing the structure of data storage in a way that optimizes retrieval and improves performance. In MongoDB, data is stored in Documents within Collections, unlike relational databases (SQL), which use tables.


Online shopping
products
suppliers
categories
orders
cart
customers
😢Un structured data
convert to
categories
products
suppliers
customers
orders
cart
😍structured / logical data model

Let’s break it into 4 steps:

1-Different types of relationships between data.

2-Referencing (normalization) VS Embedding (denormalization).

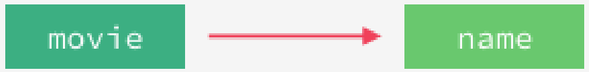
3-Embedding or referencing other documents.

4-Types of referencing.

✋The way we design / model our data can make or break our application.

1-Different types of relationships between data.

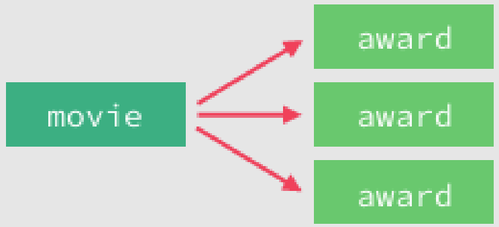
1:1



*(**1** movie can only have **1** name)*

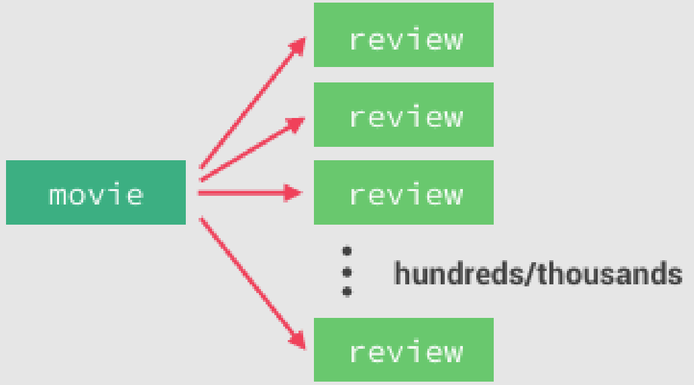
1:MANY

👉 1:FEW

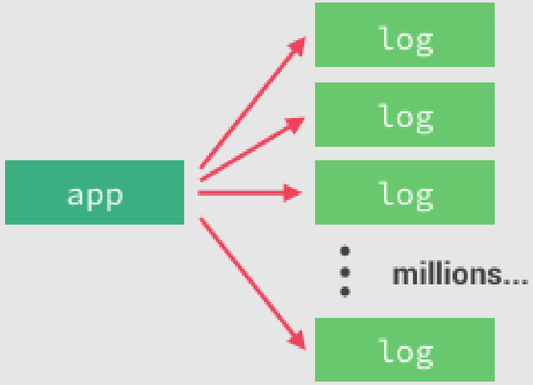


*(**1** movie can win **many** awards)*

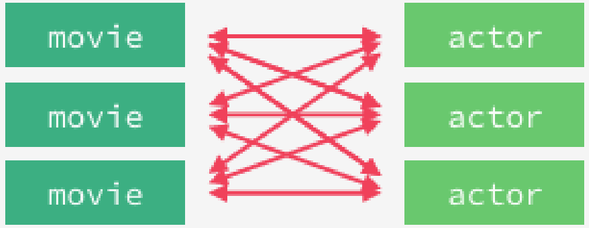
👉 1:MANY



👉 1:TON

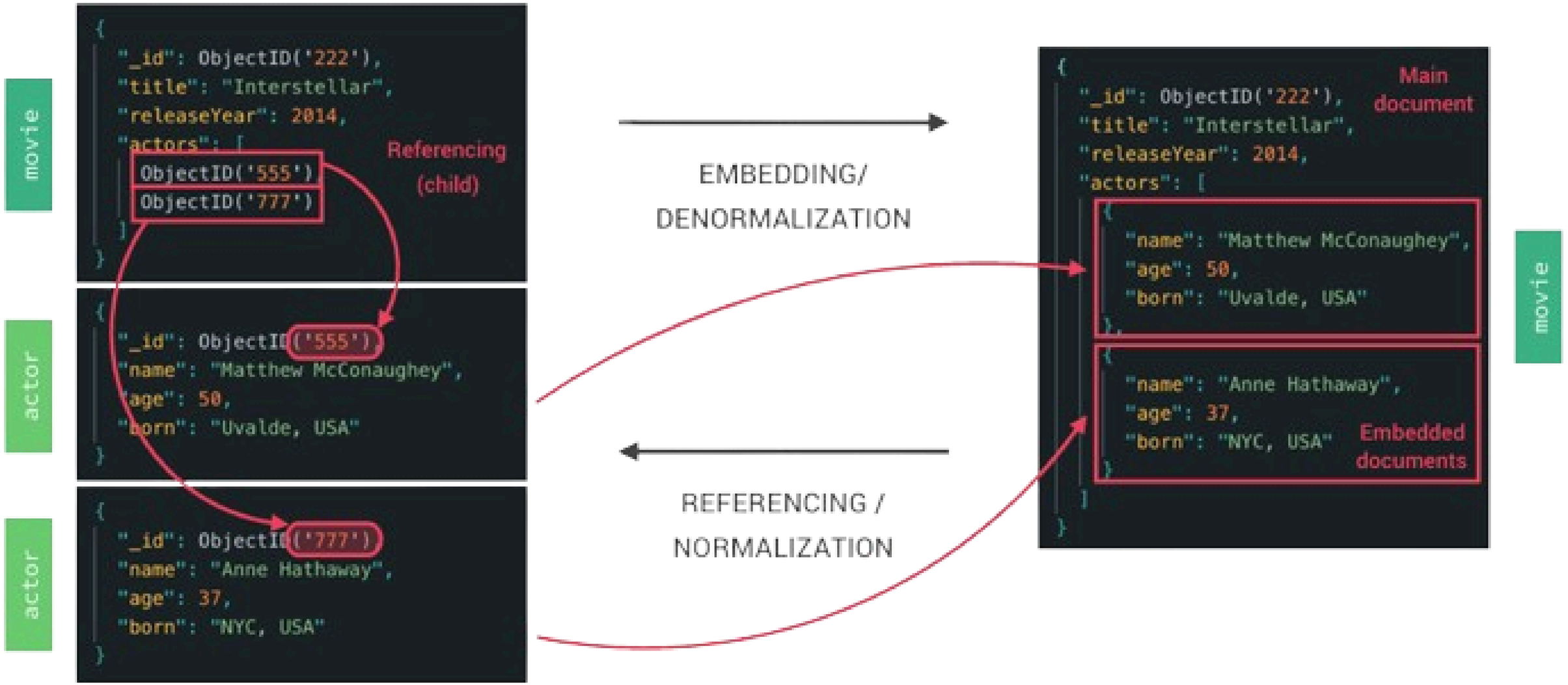


MANY:MANY



*(One movie can have **many** actors, but one actor can also play in **many** movies)*

2-Referencing VS Embedding



2-Referencing VS Embedding

😍Performance: it’s easier to query each document on its own.

😢 We need 2 queries to get data from referenced document .

😍 Performance: we can get all the information in one query.

😢 Impossible to query the embedded document on its own.

3-Embedding or referencing other documents.

when to embedd and when to reference?

to answer this question , we need to combine 3 criteria to take decision

1-RELATIONSHIP TYPE

2-DATA ACCESS PATTERNS

3-DATA CLOSENESS

EMBEDDING

REFERENCING

1 **RELATIONSHIP TYPE**

(How two datasets are related to each other)

- 1:FEW
- 1:MANY

`Movies + Images (100)` ?

- 1:MANY
- 1:TON
- MANY:MANY

2 **DATA ACCESS PATTERNS**

(How often data is read and written. Read/write ratio)

- Data is mostly **read**
- Data does **not** change quickly
- (**High** read/write ratio)

`Movies + Images`

- Data is **updated** a lot
- (**Low** read/write ratio)

`Movies + Reviews`

3 **DATA CLOSENESS**

(How "much" the data is related, how we want to query)

- Datasets **really** belong together

`User + Email Addresses`

- We frequently need to query both datasets **on their own**

`Movies + Images`

4-Types of referencing.

CHILD REFERENCING

- **Use Case Example**: An application storing a small number of log entries.

- **Structure:**

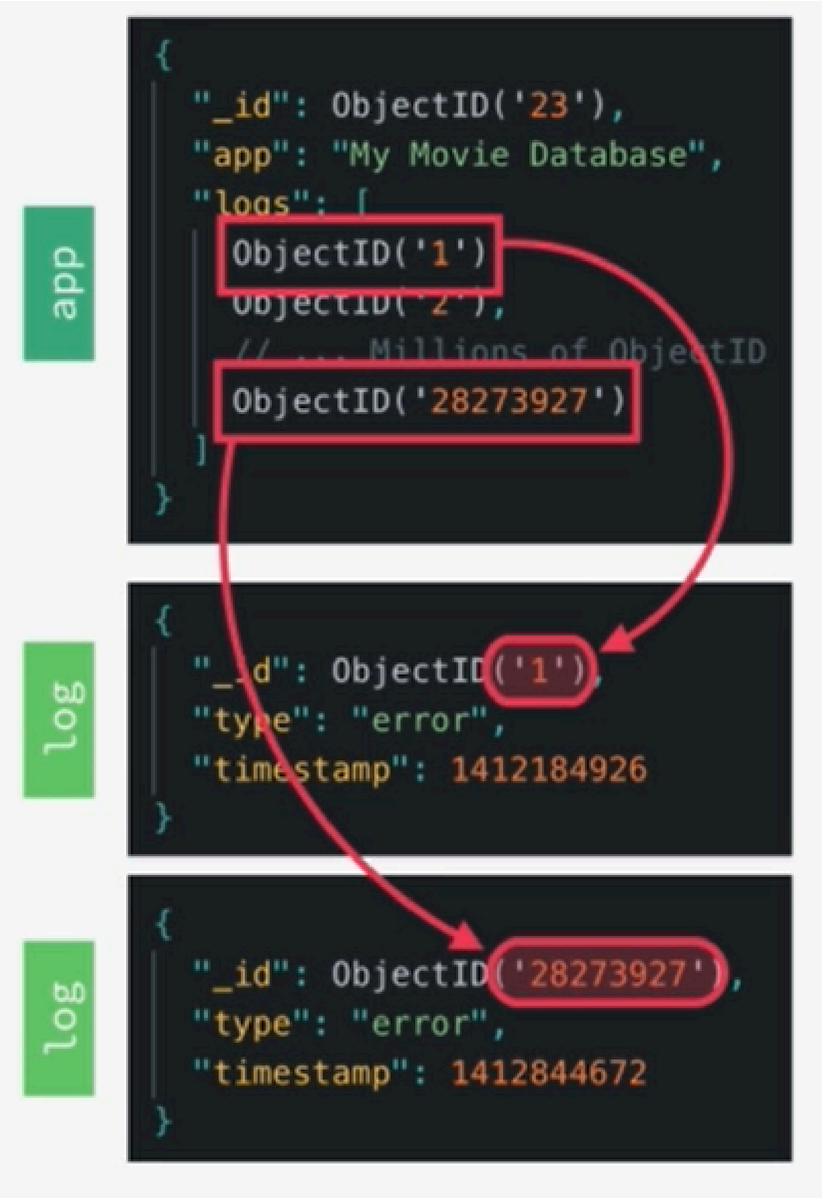
The app document contains an embedded array (logs) of ObjectIDs referencing log documents.

Each log document exists separately and can be accessed using these references.

✅ Best Practice:

Suitable when there are a limited number of related items (e.g., a few logs per app).

Uses embedding within an array for efficiency.



PARENT REFERENCING

- **Use Case Example**: An application logging multiple error messages.**(1:M)**

- **Structure:**

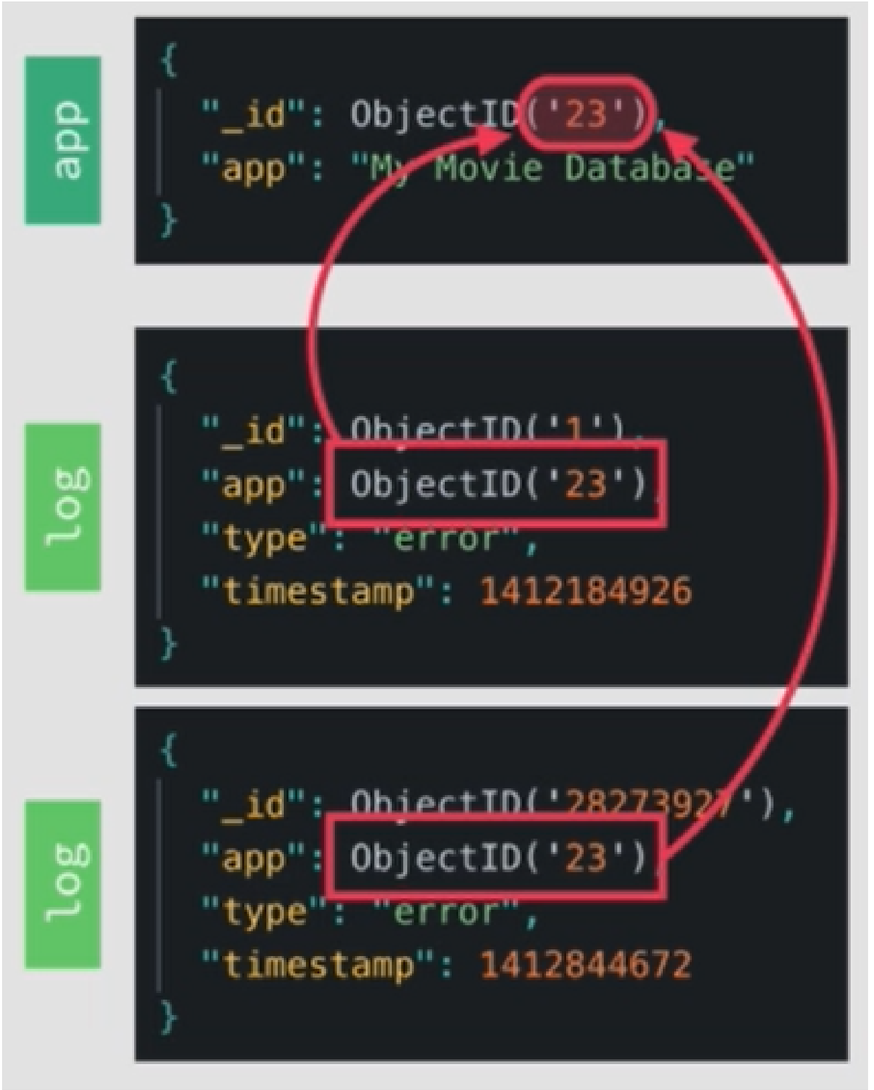
Each log document has a reference (ObjectID) back to the app document.

Instead of embedding logs in the app document, each log stores the app_id.

✅ Best Practice:

Works well when many documents are linked to one main document.

Uses referencing, which is more scalable than embedding.



PARENT REFERENCING

✌️

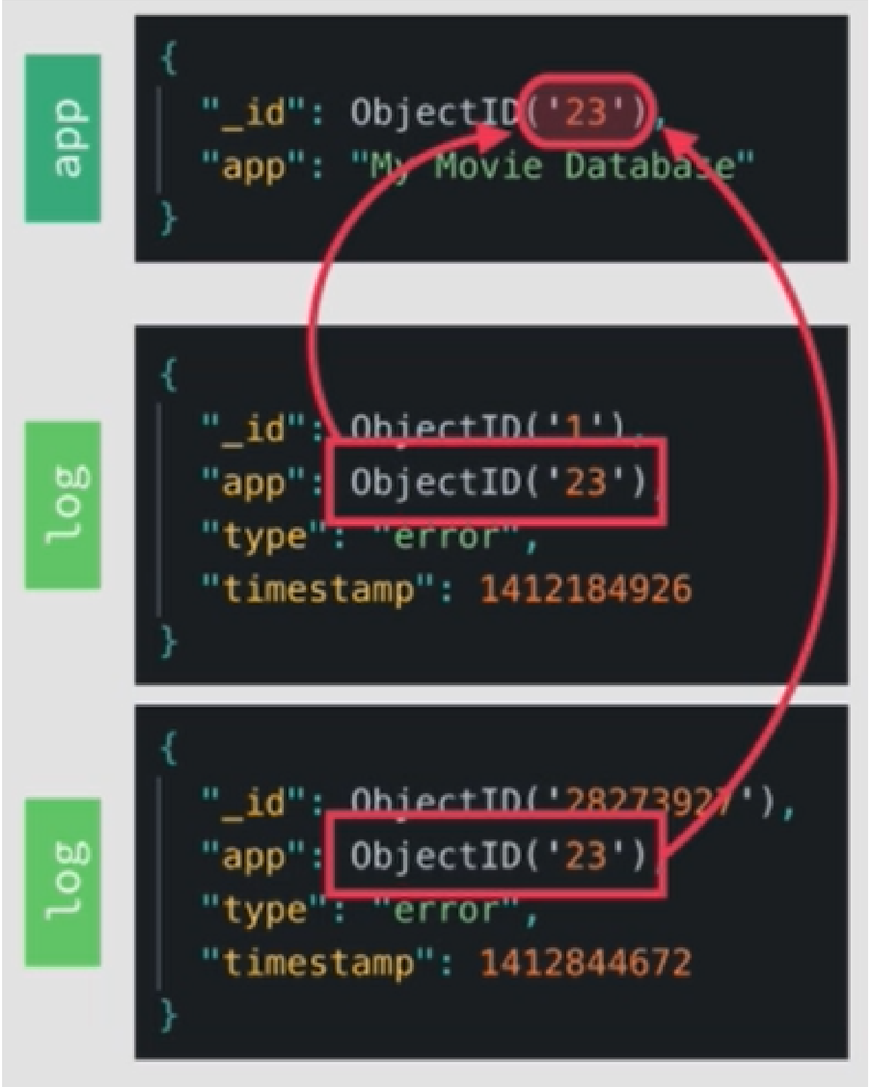
-Use Case Example: A system that generates massive logs (millions)**(1:T).**

-Structure:

- Each log document references the app document using an ObjectID, just like 1:Many.
- The key difference is scale—here, logs could reach millions.

✅ Best Practice:

- Avoid embedding in such cases due to performance and document size limits.
- Use indexes for efficient querying.



TWO-WAY REFERENCING

-Use Case Example: Movies and actors relationship.

-Structure:

The movie document contains an array of actor ObjectIDs.

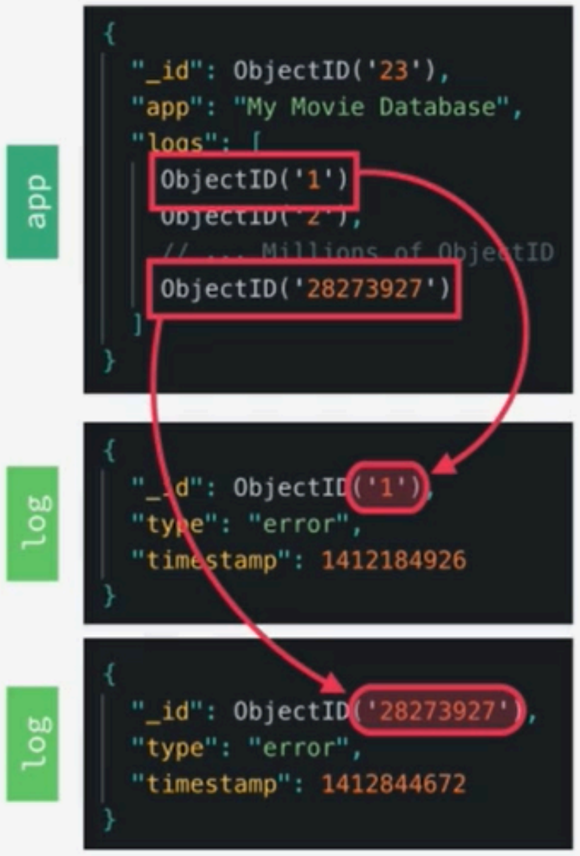
The actor document contains an array of movie ObjectIDs.

✅ Best Practice:

Ideal when both entities have multiple relationships (e.g., actors in multiple movies).

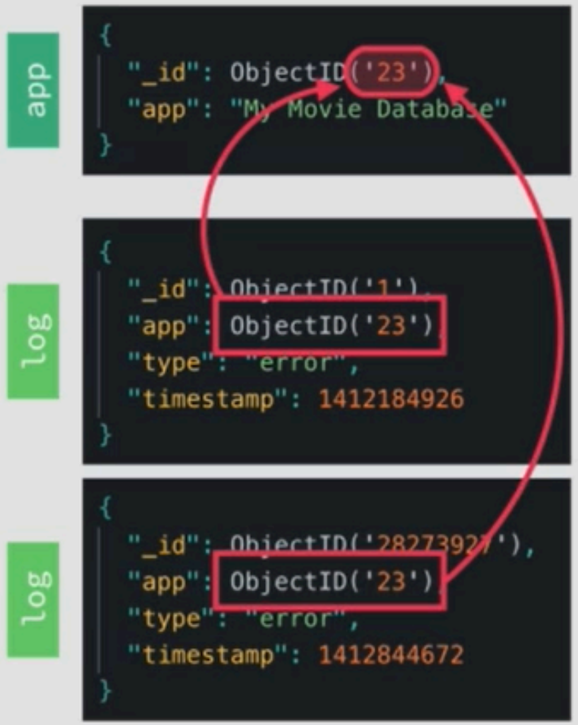
Uses referencing for flexibility.

CHILD REFERENCING



👉 1:FEW

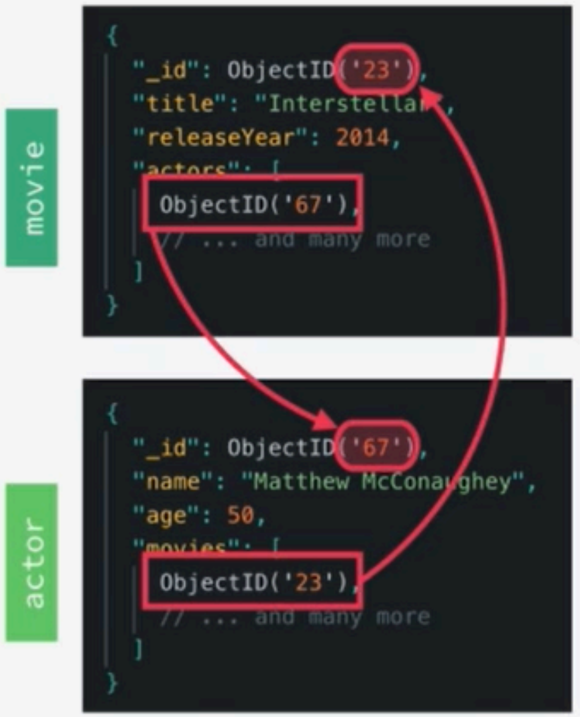
PARENT REFERENCING



👉 1:MANY

👉 1:TON

TWO-WAY REFERENCING



👉 MANY:MANY

SUMMARY

1- The **most important** principle is: Structure your data to **match the ways that your application queries and updates data.**

2- In other words: Identify the questions that arise from your **application's use cases** first, and then model your data so that the **questions can get answered** in the most efficient way.

3- In general, **always favor embedding**, unless there is a good reason not to embed. Especially on 1:FEW and 1:MANY relationships.

4-A 1:TON or a MANY:MANY relationship is usually a good reason to **reference** instead of embedding.

5- Also, favor **referencing** when data is updated a lot and if you need to frequently access a dataset on its own.

6- Use **embedding** when data is mostly read but rarely updated, and when two datasets belong intrinsically together.

7- Don’t allow arrays to grow indefinitely. Therefore, if you need to normalize, use **child referencing** for 1:MANY relationships, and **parent referencing** for 1:TON relationships.

8- Use **two-way referencing** for MANY:MANY relationships.

Introduction to Indexes

What is an Index?

- An index is a data structure that improves query performance.
- It helps MongoDB find documents faster instead of scanning the entire collection.
- Analogy: Like an index in a book, it allows quick lookups.

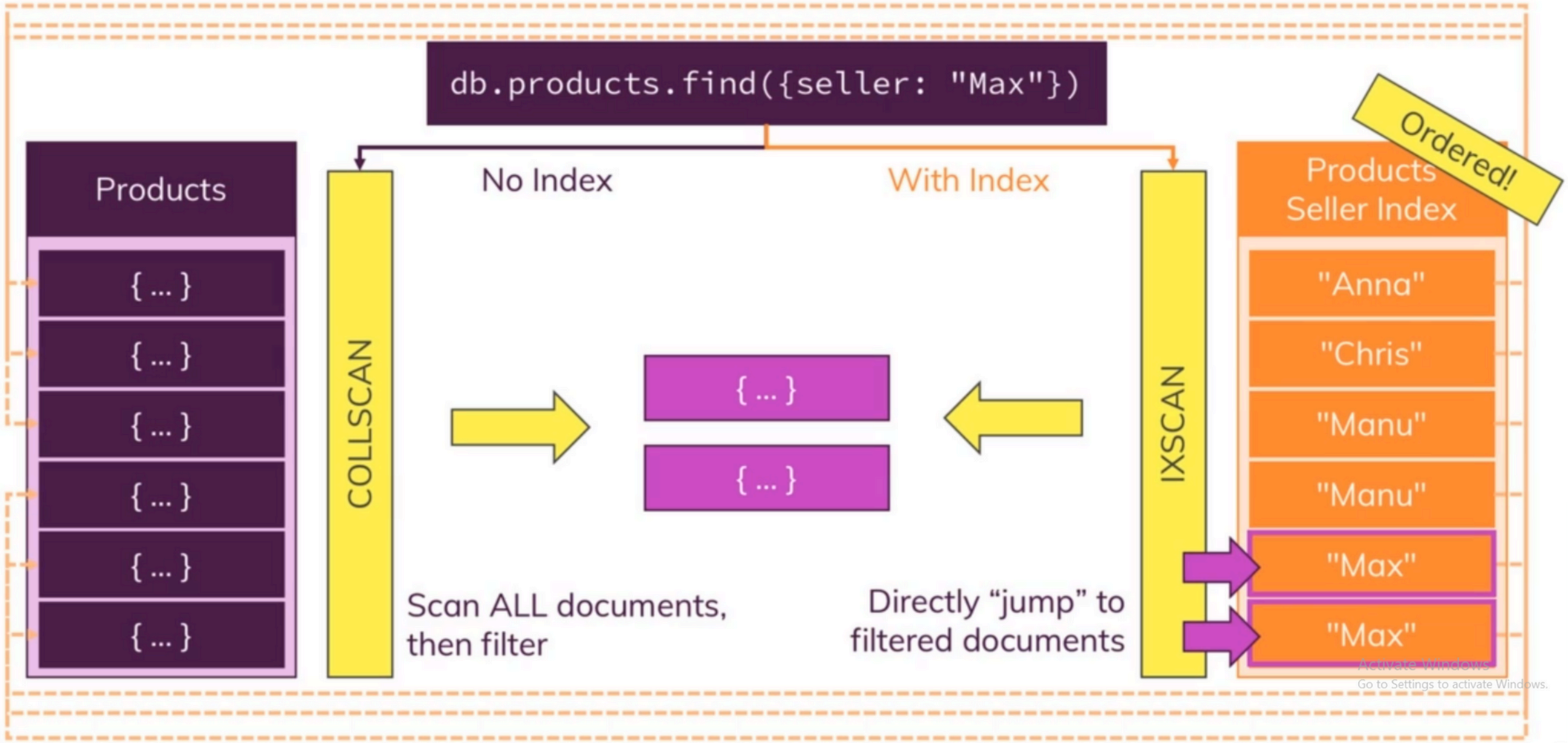
Benefits of Indexing

✅ Faster Queries – Reduces full collection scans.

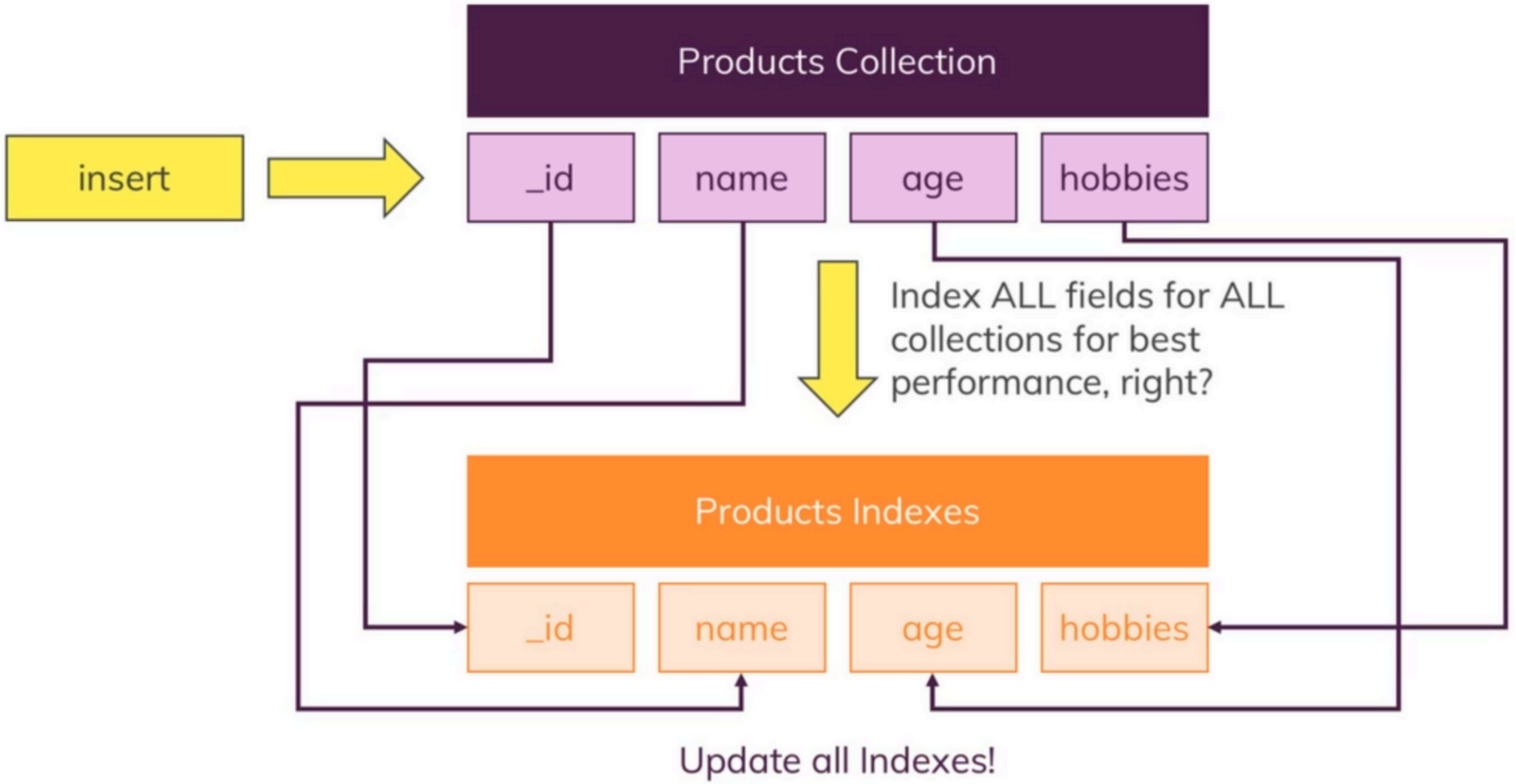
✅ Efficient Sorting – Improves sort() operations.

✅ Optimized Query Execution – Reduces query time.

How Indexes actually works



Don’t use too many indexes



IXSCAN vs COLLSCAN

What are IXSCAN and COLLSCAN?

- IXSCAN (Index Scan): MongoDB uses an index to find relevant documents efficiently.
- COLLSCAN (Collection Scan): MongoDB scans the entire collection when no suitable index exists.

Checking Query Execution Plan:

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

- If an index is used, the result should show "IXSCAN".
- If no index is used, the result will show "COLLSCAN", meaning a full scan was performed.

Creating a Basic Index

Default _id Index

- MongoDB automatically creates an index on _id.

Creating an Index on a Field

```
db.users.createIndex({ name: 1 })  // Ascending order
```

Querying with Index:

```
db.users.find({ name: "Ali" })
```

Compound Index

Indexing Multiple Fields

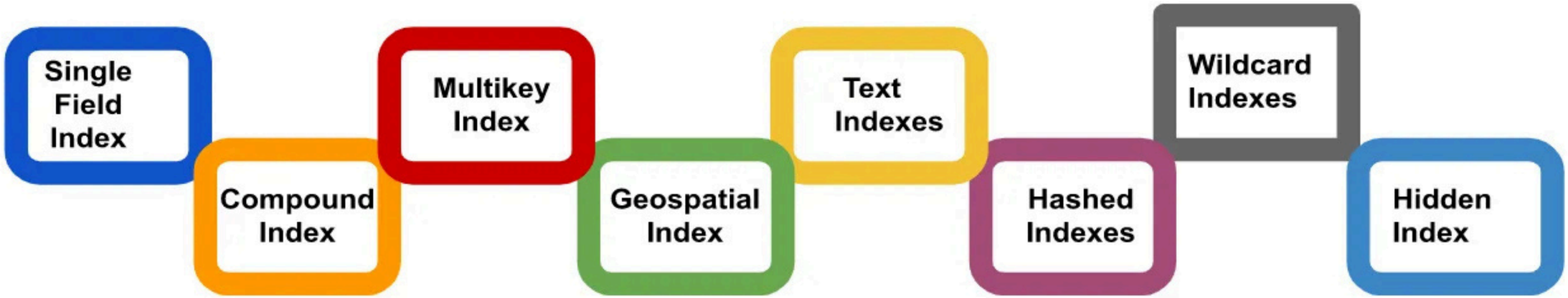
- Useful when filtering by multiple fields.

```
db.orders.createIndex({ customerId: 1, createdAt: -1 })
```

- Optimizes queries like:

```
db.orders.find({ customerId: "12345" }).sort({ createdAt: -1 })
```

Types of Indexes



Checking and Dropping Indexes

View Indexes on a Collection

```
db.users.getIndexes()
```

Check Query Performance

```
db.users.explain("executionStats").find({ name: "Ali" })
```

Remove an Index

```
db.users.dropIndex("name_1")
```

How MongoDB Stores and Uses Indexes

- Indexes are stored as B-trees, allowing efficient searching.
- Each index entry contains:
  - The indexed field(s).
  - A pointer to the actual document in the collection.
- Queries use indexes first before accessing the full document.

When NOT to Use Indexes

- Overusing indexes can slow down writes because every insert/update must update indexes.
- Avoid indexes on small datasets.
- Remove unused indexes.
- Be mindful when handling large-scale insert operations.

Summary

- Indexes improve query performance
- Use compound indexes for multi-field queries.
- Check with explain() to verify index usage.
- Avoid unnecessary indexes to optimize write performance.

Questions ?

03

Aggregation & GeoSpatial Data

Aggregation

MongoDB Aggregation Pipeline

- The aggregation pipeline is a framework for data processing in MongoDB.
- It consists of multiple stages that transform and process documents.
- Stages are executed sequentially.

Aggregation Pipeline Stages Overview

- Aggregation stages help filter, transform, and summarize data. Key stages:
- **$match** → Filters documents (like find()).
- **$project** → Includes/excludes fields, creates computed fields.
- **$group** → Groups documents, performs aggregations.
- **$count** → Counts the number of documents.
- **$sort** → Orders documents.
- **$skip** → Skips a specific number of documents.
- **$limit** → Limits the number of output documents.
- **$addFields / $set** → Adds new fields to documents.
- **$unset** → Removes fields from documents.
- **$out** -> stores the output into a new collection
- **$merge** -> merge the output into the current collection

$match and $project

Find students who are 20 years old and only return their names.

```
db.students.aggregate([
  { $match: { age: 20 } },
  { $project: { first_name: 1, last_name: 1, _id: 0 } }
])
```

✓ $match filters the documents.

✓ $project includes only the first and last name.

$group and $count

Count how many students are in each department.

```
db.students.aggregate([
  { $group: { _id: "$department", count: { $sum: 1 } } }
])
```

✓ $group groups by department.

✓ $sum: 1 counts the students in each group.

$limit, $sort, and $skip

Find the 3rd highest tuition fee.

```
db.students.aggregate([
  { $sort: { tuition: -1 } },
  { $skip: 2 },
  { $limit: 1 }
])
```

✓ $sort arranges documents by tuition in descending order.

✓ $skip: 2 skips the top 2 results.

✓ $limit: 1 returns only the 3rd highest tuition.

$sum, $avg, $min, $max, $first, $last

Find the total and average GPA per department.

```
db.students.aggregate([
  { $group: {
      _id: "$department",
      totalGPA: { $sum: "$gpa" },
      avgGPA: { $avg: "$gpa" },
      minGPA: { $min: "$gpa" },
      maxGPA: { $max: "$gpa" },
      firstGPA: { $first: "$gpa" },
      lastGPA: { $last: "$gpa" }
    }
  }
])
```

✔ $sum, $avg, $min, $max, $first, $last perform calculations within groups.

$concat

Create full names for students.

```
db.students.aggregate([
    {$project : { full_name: {$concat:["$first_name"," ", "$last_name"]} } },
    {$merge : { into:"students" , whenMatched:"merge" , whenNotMatched:"insert" }}
])
```

- ✓ $concat joins strings together.
- ✓ $merge merge into the given collection.

$arrayElemAt, $concatArrays, $reverseArray

Working with arrays in aggregation.

```
db.students.aggregate([
  { $project: {
      firstCourse: { $arrayElemAt: ["$courses", 0] },
      allCourses: { $concatArrays: ["$courses", ["Extra Course"]] },
      reversedCourses: { $reverseArray: "$courses" }
    }
  }
])
```

- ✓ $arrayElemAt → Gets the element at index you give from an array.
- ✓ $concatArrays → Merges arrays.
- ✓ $reverseArray → Reverses an array.

$unwind

Breaks down an array into multiple documents.

```
db.students.aggregate([
  { $unwind: "$courses" }
])
```

✔ $unwind creates a separate document for each array element.

$lookup (Join Collections)

Join students with their courses collection.

```
db.students.aggregate([
  { $lookup: {
      from: "courses",
      localField: "course_id",
      foreignField: "_id",
      as: "course_details"
    }
  }
])
```

✔ $lookup performs a SQL-like join between collections.

$out (Writing to a Collection)

Save processed students into a new collection.

```
db.students.aggregate([
  { $project: { fullName: { $concat: ["$first_name", " ", "$last_name"] }, age: 1, department: 1 } },
  { $out: "processedStudents" }
])
```

✓ $out writes the result to a new collection.

✗ $out must be the last stage.

Aggregation vs. find()

Aggregation can be used instead of find(), but with more flexibility.

```
db.students.find({ age: 20 }) // Equivalent to using $match in aggregation.
```

✓ find() is useful for simple queries.

✓ Aggregation is more powerful for complex operations.

Order of Stages Matters

- **$match** and **$limit** should be as early as possible to improve performance.
- Stages like **$project**, **$addFields**, and **$set** do not modify the original data.
- **$out** and **$merge** do modify data.

Performance Considerations

🚀 Best Practices for Aggregation Performance:

✅ Use $limit early to reduce memory usage.

✅ Use $project to include only necessary fields.

✅ Aggregation results are not subject to the 16MB document limit.

✅ Each stage can use up to 100MB of RAM (beyond that, indexing helps).

✅ Transformation stages can prevent index usage in later stages.

Summary

🎯 Aggregation is powerful for:

✅ Data transformation & filtering.

✅ Computing sums, averages, and counts.

✅ Working with arrays and performing joins.

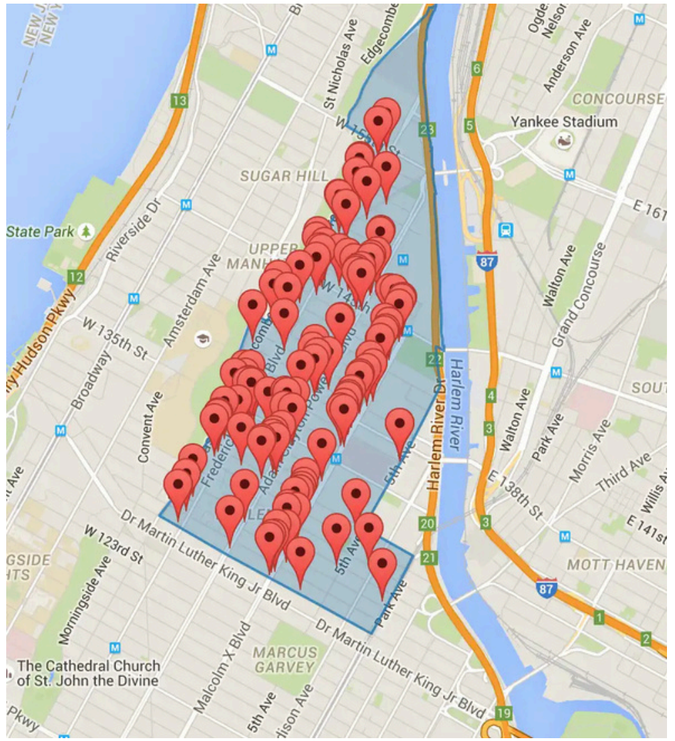
✅ Writing processed data to new collections.

✅ Optimizing queries for performance.

GeoSpatial Data

Geospatial Data

GeoSpatial Data refers to any data that includes geographic information related to objects on Earth's surface. This data is usually represented using geographical coordinates such as latitude and longitude, or through other coordinate systems.



GeoSpatial Data in MongoDB

1- Points(📍)

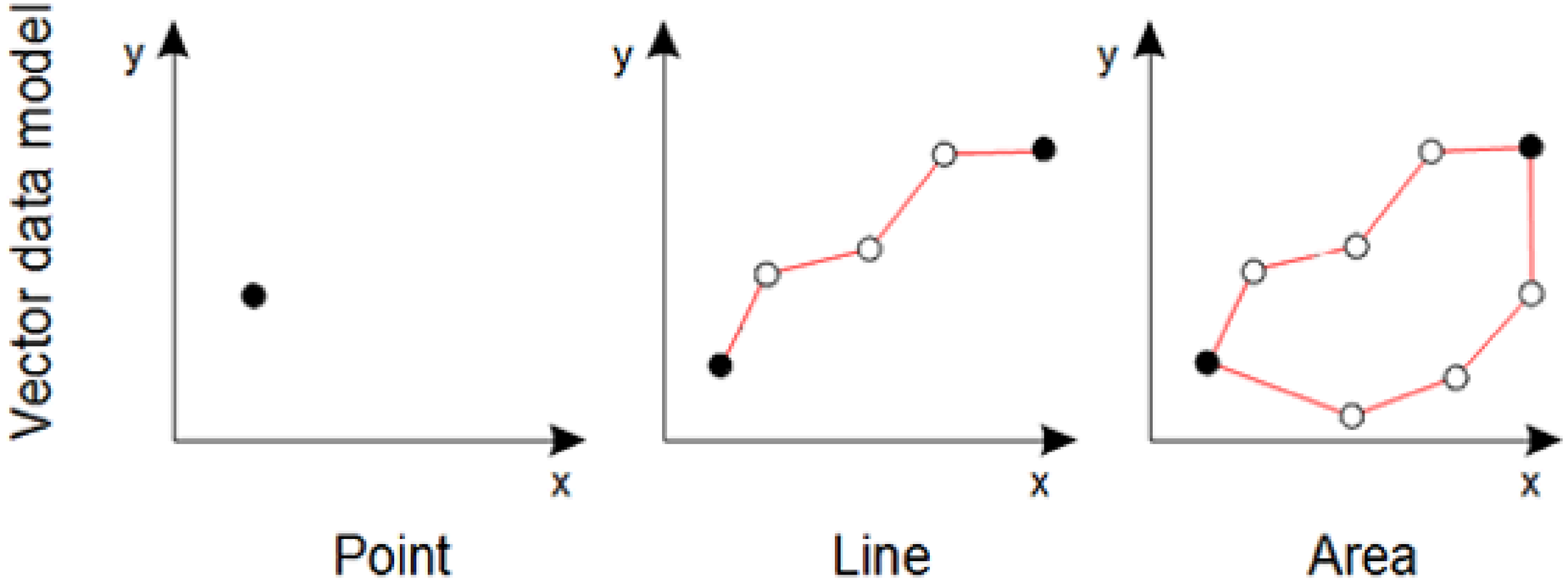
Represent specific locations in space with a single coordinate pair (latitude & longitude).

2- Lines(🛤️)

- Represent linear features that connect two or more points.
- Have length but no width.

3- Polygons(🏞️)

- Represent enclosed areas formed by multiple connected points.
- Have both length and area.


Vector data model
y
x
Point
y
x
Line
y
x
Area

1. Store Some Points and Find the Nearest Point

we have steps to do that :

Step 1: Insert Some Locations (Points)

We'll store some locations (e.g., coffee shops).

```
db.places.insertMany([
  { name: "Cafe A", location: { type: "Point", coordinates: [30.05, 31.22] } },
  { name: "Cafe B", location: { type: "Point", coordinates: [30.07, 31.21] } },
  { name: "Cafe C", location: { type: "Point", coordinates: [30.06, 31.25] } }
]);
```

Note : In Google Maps, the latitude (lat) comes first, followed by the longitude (long).
[latitude,longitude]

[longitude, latitude]

1. Store Some Points and Find the Nearest Point

Step 2: Create a 2dsphere Index

To enable geospatial queries, we must create an index on the location field:

```
db.places.createIndex({ location: "2dsphere" });
```

1. Store Some Points and Find the Nearest Point

Step 3: Find the Nearest Place to a Given Point

Let’s find the nearest café to the user’s location: [30.065, 31.23].

```
db.places.findOne({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [30.065, 31.23] // User's location
      }
    }
  }
});
```

How it Works:

$near → Finds the nearest point to the given coordinates.

$geometry → Defines the user's location as a point.

2. Store an Area (Polygon) and Check if a Point is Inside

we have steps to do that :

Step 1: Insert a Polygon (Area)

Let’s store an area (e.g., a park boundary).

```
db.areas.insertOne({
  name: "Central Park",
  location: {
    type: "Polygon",
    coordinates: [[
      [30.04, 31.23],
      [30.05, 31.24],
      [30.06, 31.22],
      [30.04, 31.23]  // Polygon must be closed (first and last point should be the same)
    ]]
  }
});
```

2. Store an Area (Polygon) and Check if a Point is Inside

Step 2: Create a 2dsphere Index

```
db.areas.createIndex({ location: "2dsphere" });
```

2. Store an Area (Polygon) and Check if a Point is Inside

Step 3: Check if a Point is Inside the Polygon

Let’s check if a user at [30.045, 31.235] is inside the park.

```
db.areas.find({
  location: {
    $geoIntersects: {
      $geometry: {
        type: "Point",
        coordinates: [31.357964, 31.046059]
      }
    }
  }
});
```

How it Works:

$geoWithin → Checks if the given point is inside a polygon.

3. Store Some Points and Get Locations Within a Given Circular Distance

we have steps to do that :

Step 1: Insert Some Locations (Points)

We’ll store some restaurants.

```
db.restaurants.insertMany([
  { name: "Restaurant X", location: { type: "Point", coordinates: [30.05, 31.22] } },
  { name: "Restaurant Y", location: { type: "Point", coordinates: [30.07, 31.21] } },
  { name: "Restaurant Z", location: { type: "Point", coordinates: [30.06, 31.25] } }
]);
```

3. Store Some Points and Get Locations Within a Given Circular Distance

Step 2: Create a 2dsphere Index

```
db.restaurants.createIndex({ location: "2dsphere" });
```

3. Store Some Points and Get Locations Within a Given Circular Distance

Step 3: Get Locations Within a Given Radius

Find restaurants within 5 km of the user's location: [30.065, 31.23].

```
db.restaurants.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [30.065, 31.23] // User's location
      },
      $maxDistance: 5000 // Distance in meters (5 km)
    }
  }
});
```

How it Works:

$near → Finds points closest to the given coordinates.

$maxDistance: 5000 → Limits results to 5 km (5000 meters).

Summary

-Find the Nearest Place → $near

-Check if a Point is Inside an Area → $geoIntersects

-Find Locations Within a Radius → $near with $maxDistance

-Let me know if you need any modifications or more explanations!

docs
https://www.mongodb.com/docs/manual/geospatial-queries/

Questions ?

Contact Info

Shady Radwan

Rwan Adel