MongoDB - Part 2

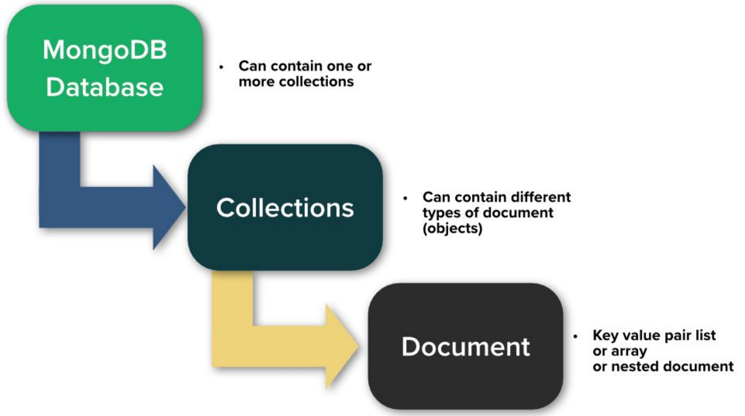
Goal: Continue building the mental model

What is MongoDB

MongoDB is a NoSQL database that stores data as documents instead of rows.

A **document is JSON-like** (BSON) and can contain nested objects and arrays.

Mapping from MySQL: row is like a document, table is like a collection, and schema is like a database.

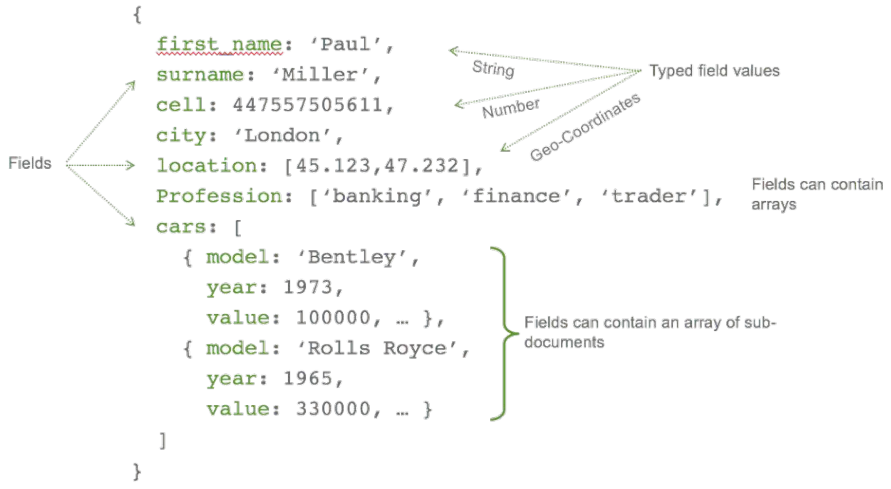

MongoDB Database
Can contain one or more collections
Collections
Can contain different types of document (objects)
Document
Key value pair list or array or nested document

Documents can hold related data together

You can store a user, their address, and their skills inside one document.

When you fetch that document, you often get everything needed in one read.

In MySQL, the same shape usually becomes multiple tables and joins.



```
{
  first_name: ‘Paul’,
  surname: ‘Miller’,
  cell: 447557505611,
  city: ‘London’,
  location: [45.123,47.232],
  Profession: [‘banking’, ‘finance’, ‘trader’],
  cars: [
    { model: ‘Bentley’,
      year: 1973,
      value: 100000, … },
    { model: ‘Rolls Royce’,
      year: 1965,
      value: 330000, … }
  ]
}
```

Fields
String
Number
Geo-Coordinates
Typed field values
Fields can contain arrays
Fields can contain an array of sub-documents

_id and ObjectId

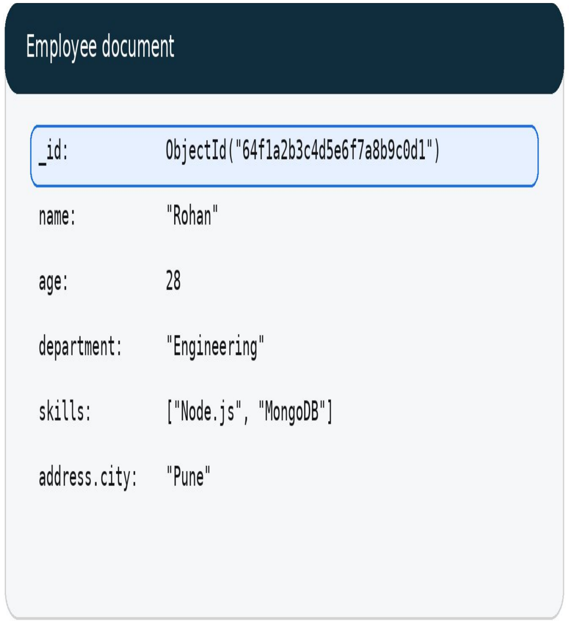
The _id field

Every document has an _id field, which uniquely identifies that document.

If you do not provide _id, MongoDB creates one for you as an ObjectId.

Think of _id as the database’s identity for a document, not your business identity.

```
db.employees.find({ _id:
ObjectId("64f1a2...") })
```



Employee document

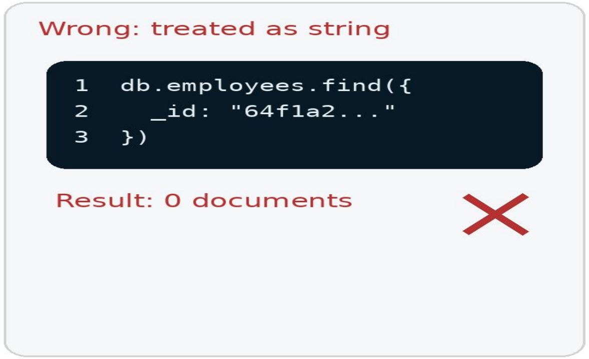
_id:	ObjectId("64f1a2b3c4d5e6f7a8b9c0d1")
name:	"Rohan"
age:	28
department:	"Engineering"
skills:	["Node.js", "MongoDB"]
address.city:	"Pune"

ObjectId is a type

If you copy an _id value as plain text, it may not match because the type is wrong.

When a field is stored as ObjectId, your query should use ObjectId(...) as well.

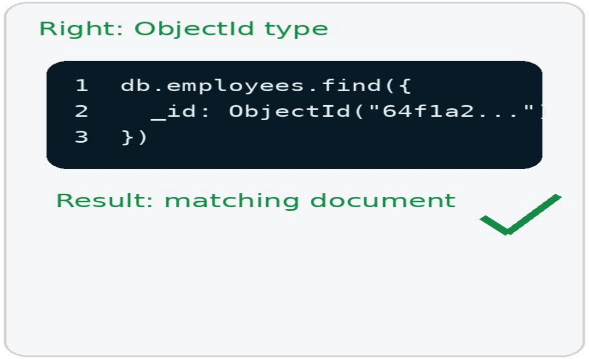
This is similar to comparing a number as text versus as an actual number.



Wrong: treated as string

```
db.employees.find({
  _id: "64f1a2..."
})
```

Result: 0 documents



Right: ObjectId type

```
db.employees.find({
  _id: ObjectId("64f1a2..."
})
```

Result: matching document

How’s ObjectId created?

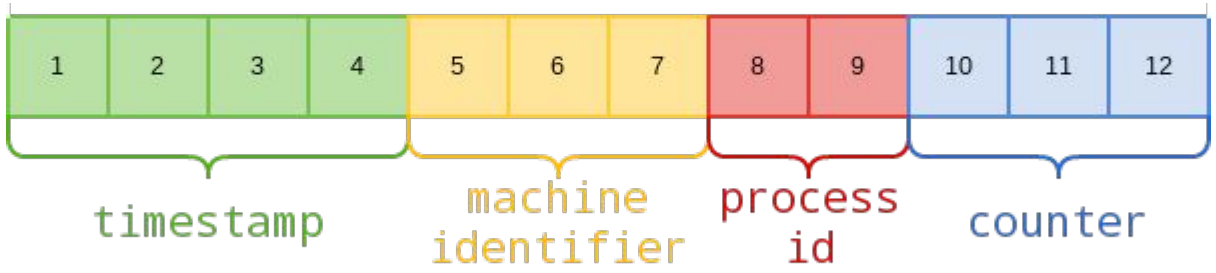
MongoDB generates an ObjectId automatically when you insert a document without an _id.

ObjectId is 12 bytes long and is designed to be unique without asking the server first.

Parts of ObjectId:

- Timestamp: when the id was created (seconds)
- Machine identifier: identifies the machine
- Process id: identifies the process on that machine
- Counter: increments to avoid collisions within the same second

Takeaway: ObjectId is not random. It is a structured id that helps uniqueness and can be sorted roughly by creation time



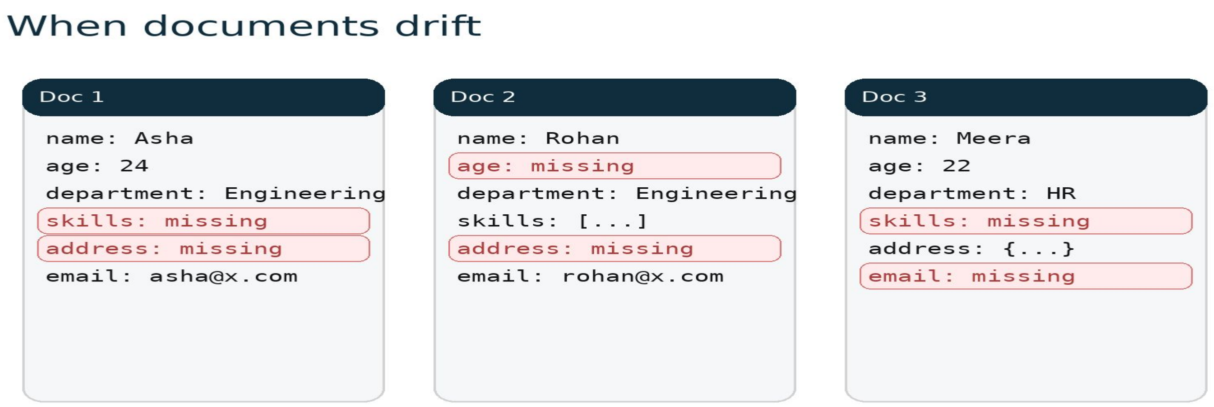
Consistency and duplicates

Schema flexibility is powerful

MongoDB does not force every document in a collection to have the same fields.

That flexibility helps you evolve your data model as your product changes.

The trade-off is that your team must actively prevent messy, drifting documents.



Visual cue: the same collection can slowly become inconsistent unless you enforce a shape.

Keeping documents consistent

Decide the fields you want to be present for every document in the collection.

Enforce the shape in two places: your **application code** (we’ll cover this later) and the **database itself**.

This makes your queries predictable and reduces bugs when the data grows.

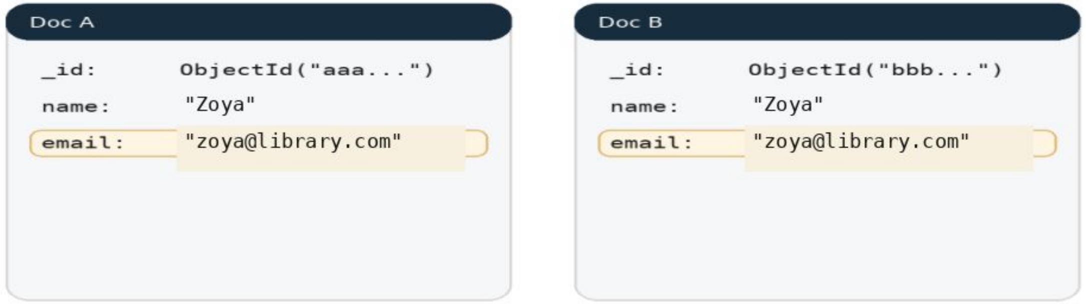
Example rule: every employee must have name, department, and age.

Duplicates happen when business identity is missing

MongoDB will allow two documents that look similar, because _id is different.

If your app cares about a field like email or unique ID for a book (eg: ISBN), treat it as a business key.

Once you pick a business key, you can prevent duplicates using a unique index.



Doc A

```
_id:      ObjectId("aaa...")
name:     "Zoya"
email:    "zoya@library.com"
```

Doc B

```
_id:      ObjectId("bbb...")
name:     "Zoya"
email:    "zoya@library.com"
```

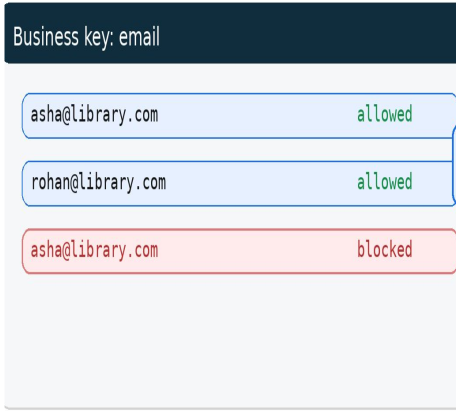
Prevent duplicates with a unique index

A unique index enforces that a chosen field cannot repeat across documents.

This turns a business rule into a database rule, so it cannot be bypassed.

If a duplicate insert happens, MongoDB rejects it instead of silently storing it.

```
db.employees.createIndex(
  { email: 1 },
  { unique: true }
)
```



Indexing foundations - Lecture 41

Query essentials

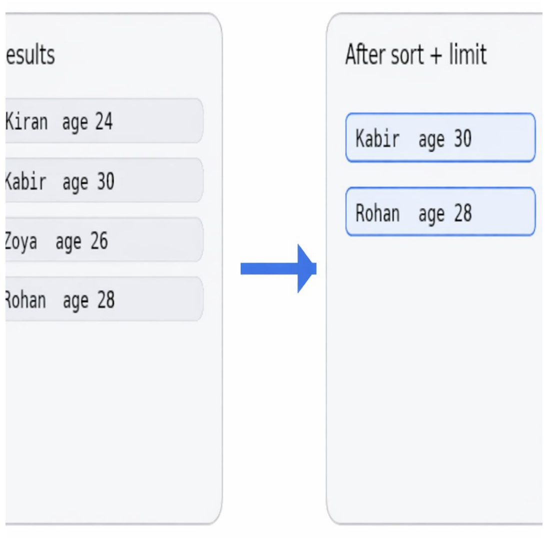
A simple query is filter + optional sort + optional

A filter selects documents that match your condition, similar to WHERE in MySQL.

Sorting orders the result set, and limit reduces how many documents you return.

This pattern keeps queries readable and avoids fetching more data than needed.

```
db.employees.find({ department:
"Engineering" })
  .sort({ age: -1 })
  .limit(2)
```



Use $in for an IN-style filter

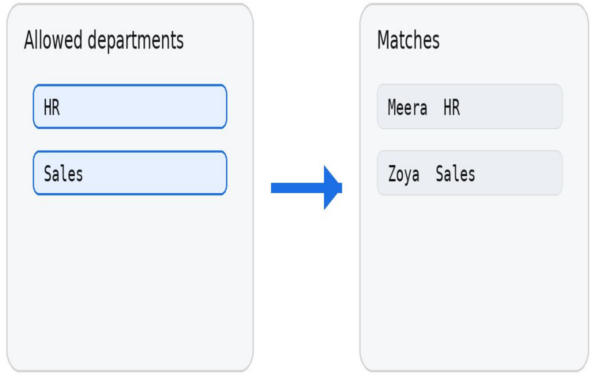
$in matches documents where a field equals any value from a list.

This is similar to IN (...) in MySQL and keeps multi-value filters clean.

```
db.employees.find({
  department: { $in: ["HR", "Sales"] }
})
```

Tip: combine $in with sort and limit when you build dashboards.

$in matches one of many values



Visual cue: $in keeps queries readable when you want several allowed values.

Updates beyond the basics

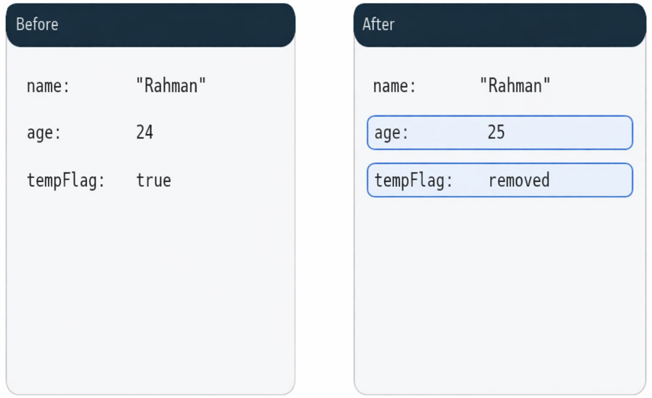
Update only what changes with $set and $unset

$set updates specific fields without replacing the whole document.

$unset removes a field when it is no longer needed.

Always include a clear filter, so you update the right document.

```
db.employees.updateOne(
  { name: "Asha" },
  {
        $set: { age: 25 },
        $unset: { tempFlag: "" }
  }
)
```



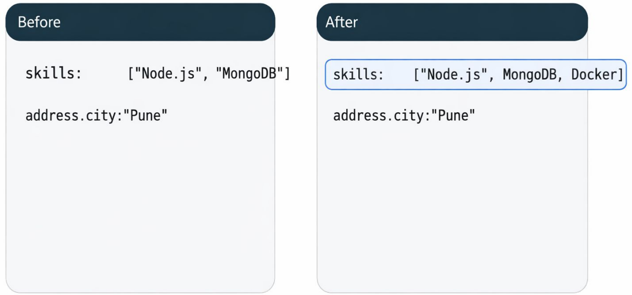
Arrays and nested fields

Use $addToSet to add to an array without creating duplicates.

Use dot notation to target a nested field like address.city.

This keeps your document natural while still being query-friendly.

```
db.employees.updateOne(
  { name: "Rohan" },
  {
    $addToSet: { skills: "Docker" },
    $set: { "address.city": "Pune" }
  }
)
```



Before

```
skills:      ["Node.js", "MongoDB"]

address.city:"Pune"
```

After

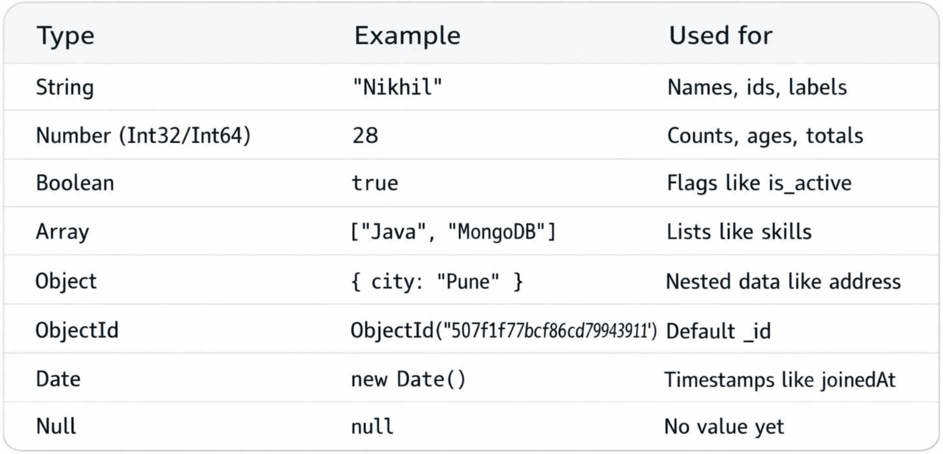
```
skills:    ["Node.js", MongoDB, Docker]

address.city:"Pune"
```

Data types reference

BSON data types you will see often

MongoDB stores documents as BSON, which supports rich types beyond plain JSON.

Knowing the type helps you query correctly, especially for ObjectId, Date, and Arrays.



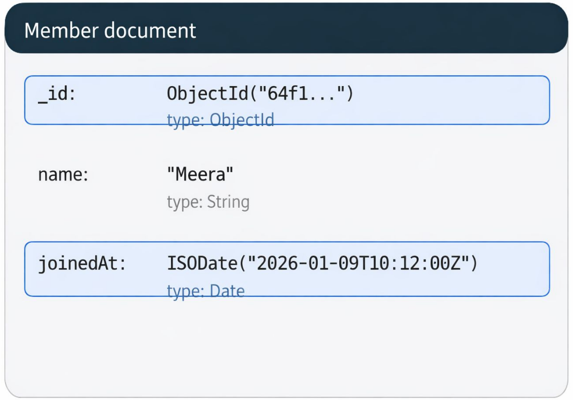
Type	Example	Used for
String	"Nikhil"	Names, ids, labels
Number (Int32/Int64)	28	Counts, ages, totals
Boolean	true	Flags like is_active
Array	["Java", "MongoDB"]	Lists like skills
Object	{ city: "Pune" }	Nested data like address
ObjectId	ObjectId("507f1f77bcf86cd79943911')	Default _id
Date	new Date()	Timestamps like joinedAt
Null	null	No value yet

Two practical types: ObjectId and Date

ObjectId is the default type for _id, so queries should use ObjectId(...) when needed.

Date should be stored as a Date type so sorting and range queries work reliably.

```
db.employees.find({
     joinedAt: { $gte: new Date("2026-01-01") }
})
.sort({ joinedAt: -1 })
```



Member document

_id:	ObjectId("64f1...") type: ObjectId
name:	"Meera" type: String
joinedAt:	ISODate("2026-01-09T10:12:00Z") type: Date