CURSO DE ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL Y BIG DATA

Práctica: Instalación y Uso de MongoDB con Docker

En esta práctica vas a instalar y poner en marcha un contenedor de MongoDB utilizando Docker. Una vez que la base de datos esté en ejecución, accederás a ella mediante MongoDB Compass para crear una base de datos, definir colecciones e insertar documentos.

JUAN FRANCISCO GARCÍA HINOJOSA

ÍNDICE.

1. Introducción

En esta práctica vas a trabajar con **MongoDB**, una base de datos de tipo NoSQL muy utilizada en el ámbito de Big Data. A diferencia de las bases de datos SQL tradicionales, MongoDB almacena la información en documentos JSON, permitiendo estructuras flexibles y anidadas.

Antes de comenzar la instalación, es importante entender las diferencias fundamentales entre SQL y NoSQL. La siguiente tabla resume las equivalencias más comunes:

SQL	MongoDB
Base de Datos	Base de Datos
Tabla	Colección
Índice	Índice
Fila	Documento
Columna	Campo
Joins	Integrados en documentos

Estas diferencias te ayudarán a interpretar correctamente la estructura interna de MongoDB cuando trabajes con documentos, colecciones y campos.

Ejemplo de documento en MongoDB

El siguiente ejemplo muestra un documento real en formato JSON almacenado dentro de una colección de MongoDB:


```
{
 "_id": "1",
 "nombre": "Juan",
 "apellido": "Pérez",
 "edad": 30,
 "direccion": {
  "calle": "Calle Principal",
  "ciudad": "Ciudad de México",
  "pais": "México"
 },
 "intereses": ["deportes", "música", "viajes"],
 "trabajos": [
  {
   "empresa": "ABC Inc.",
   "puesto": "Desarrollador",
   "años": 2
  },
  {
   "empresa": "XYZ Corp.",
   "puesto": "Analista",
   "años": 3
  }
 ]
}
```


En este documento podemos distinguir:

- **_id**: identificador único del documento.
- **nombre, apellido, edad**: campos simples.
- **dirección**: un subdocumento con varios subcampos.
- **intereses**: un array de valores.
- **trabajos**: un array de subdocumentos.

MongoDB permite almacenar estructuras complejas sin necesidad de relaciones entre tablas, lo cual es una ventaja en aplicaciones que manejan datos flexibles.

2. Instalación de MongoDB con Docker

Ya que hemos visto un poco como funciona Docker, en esta sección crearás un contenedor con MongoDB siguiendo las indicaciones oficiales. La documentación completa está disponible en:

`https://www.mongodb.com/docs/v7.0/tutorial/install-mongodb-community-with-docker/`

1. Descargar la imagen oficial

Utiliza el siguiente comando para descargar la imagen más reciente:

```
docker pull mongodb/mongodb-community-server:latest
```

Si necesitas una versión concreta, puedes especificarla:

```
docker pull mongodb/mongodb-community-server:5.0-ubuntu2004
```

2. Crear y ejecutar el contenedor

Crea un contenedor llamado `mongodb` mapeando el puerto 27017 del contenedor al de tu máquina:

```
docker run --name mongodb -p 27017:27017 -d mongodb/mongodb-community
-server:latest
```

Para comprobar que está corriendo:

```
docker container ls
```

Asegúrate de que el estado aparece como `Up`.

3. Cliente gráfico MongoDB Compass

Para trabajar de forma cómoda con la base de datos, se usará **MongoDB Compass**, el cliente visual oficial.

Descárgalo desde:

`https://www.mongodb.com/try/download/compass`

Una vez instalado, abre el programa y utiliza la cadena de conexión (seguramente salga por defecto):

`mongodb://localhost:27017`

Si tu contenedor está en ejecución y te has conseguido conectar correctamente, Compass mostrará:

- las bases de datos existentes,
- el número de colecciones,
- el tamaño de los documentos,
- estadísticas de almacenamiento.

4. Creación de bases de datos y documentos

Una vez conectado a MongoDB Compass:

1. Crear una base de datos

Selecciona:

1. **Create Database**
2. Escribe el nombre, por ejemplo: `empresa`
3. Define la primera colección, por ejemplo: `empleados`

2. Insertar un documento

Puedes insertar documentos JSON como el siguiente:

```
{
  "_id": "1",
  "nombre": "Juan",
  "apellido": "Perez",
  "edad": 30,
  "direccion": {
    "calle": "Calle Principal",
    "ciudad": "Ciudad de Mexico",
    "pais": "Mexico"
  },
  "intereses": ["deportes", "musica", "viajes"],
  "trabajos": [
    {"empresa": "ABC Inc.", "puesto": "Desarrollador", "tiempo": 2},
    {"empresa": "XYZ Corp.", "puesto": "Analista", "tiempo": 3}
  ]
}
```

5. Entrega de la práctica

Para completar esta práctica debes entregar únicamente el ejercicio final de modelado NoSQL. A partir del siguiente texto, identifica las colecciones que consideres necesarias y construye los documentos correspondientes en MongoDB. Después, crea la base de datos en tu contenedor, inserta los documentos y haz capturas del resultado en MongoDB Compass.

Texto para analizar

La empresa **TechNova** desarrolla proyectos tecnológicos para clientes de distintos sectores. Actualmente cuenta con varios empleados en plantilla. Uno de ellos es **Laura Gómez**, de 28 años, que trabaja en el departamento de *Desarrollo*. Su correo profesional es `laura.gomez@technova.com`. Laura ha participado en varios proyectos, incluyendo *Sistema de recomendación 2022*, *Chatbot corporativo 2023* y *Optimización logística 2024*. Vive en Madrid, en la calle *Alcalá 128*.

Otro empleado es **Carlos Ruiz**, de 35 años, que forma parte del departamento de *Marketing*. Su correo es `carlos.ruiz@technova.com`. Carlos ha trabajado en campañas como *Campaña internacional 2021* y *Lanzamiento de marca 2023*. Reside en Barcelona, en la avenida *Diagonal 250*.

TechNova también gestiona una lista de **clientes activos**. Uno de ellos es la empresa *DataPlus*, dedicada al análisis financiero. Otro cliente es *GreenHome*, especializada en energías renovables. Cada cliente cuenta con un contacto principal: en DataPlus es **Marta Salinas** (`marta@dataplus.com`) y en GreenHome es **Pedro Lara** (`pedro@greenhome.com`).

La empresa asigna proyectos a clientes. Por ejemplo, el *Sistema de recomendación 2022* se desarrolló para *DataPlus*, mientras que *Chatbot corporativo 2023* y *Lanzamiento de marca 2023* fueron encargados por *GreenHome*.

Tareas a realizar

1. A partir del texto, decide qué colecciones vas a crear.
2. Diseña los documentos correspondientes, utilizando subdocumentos o arrays cuando sea necesario (por ejemplo, para los proyectos).
3. Crea en MongoDB una base de datos llamada `empresa`.
4. Captura el resultado en MongoDB Compass (vista de la colección y de los documentos).

Entrega

Debes entregar un archivo comprimido que incluya:

- Un archivo de texto o PDF con:
  - Las colecciones que has decidido crear.
  - El diseño de los documentos en formato JSON.
- Capturas de pantalla de MongoDB Compass mostrando:
  - La base de datos creada.
  - Las colecciones.
  - Los documentos insertados.

6. Introducción a mongosh

En esta guía vamos a aprender a trabajar con **MongoDB** desde la consola utilizando la herramienta oficial `mongosh`. El objetivo es que entiendas cómo funciona la consola de mongo para así poder crear bases de datos, colecciones y documentos, y cómo se insertan datos de forma individual o masiva.

A lo largo de esta parte combinaremos **explicaciones teóricas** con **ejemplos prácticos**, para que entiendas lo que ocurre y puedas aplicarlo por tu cuenta.

7. Instalación de `mongosh` en Windows

Para trabajar con MongoDB desde la terminal necesitaremos la herramienta oficial **MongoDB Shell**, conocida como `mongosh`. Este programa nos permite ejecutar todos los comandos del servidor MongoDB directamente desde consola. A continuación veremos cómo instalarlo correctamente en Windows.

7.1. Descargar `mongosh`

Accedemos a la página oficial de descargas:

`https://www.mongodb.com/try/download/shell`

Descargamos el archivo ZIP correspondiente a la versión de Windows (por ejemplo, `mongosh-1.10.6-win32-x64.zip`).

Una vez descargado, descomprimimos su contenido en una ubicación permanente del equipo, por ejemplo:

```
C:\Program Files\mongosh\
```

7.2. Añadir `mongosh` a las variables de entorno

Para poder ejecutar `mongosh.exe` desde cualquier terminal (CMD, PowerShell o Windows Terminal), debemos añadir su ruta al **Path del sistema**.

Pasos:

1. Abrir el menú Inicio y escribir: Editar las variables de entorno del sistema.
2. Pulsar en el botón: **Variables de entorno**.
3. En **Variables del sistema**, seleccionar la variable `Path`.

4. Hacer clic en **Editar**.

5. Añadir una nueva entrada con la ruta donde se encuentra `mongosh.exe`.

Por ejemplo:

```
C:\Program Files\mongosh\
```

Guardamos los cambios y cerramos las ventanas.

7.3. Probar la instalación

Abrimos una terminal nueva y escribimos:

```
mongosh.exe
```

Si todo está instalado correctamente, veremos un mensaje similar:

```
Using MongoDB:        7.0.x
Using Mongosh:        1.10.6
test>
```

Esto indica que `mongosh` está listo para usar.

7.4. Conectarse al servidor MongoDB en Docker

Si tenemos un contenedor MongoDB funcionando en el puerto `27017`, podemos conectarnos desde `mongosh` simplemente con:

```
mongosh "mongodb://localhost:27017"
```

Una vez conectados, ya podemos ejecutar comandos sobre el servidor.

8. Conceptos básicos esenciales de MongoDB

Antes de comenzar a trabajar con datos, es importante comprender la estructura básica de MongoDB y cómo organiza la información. Estos conceptos serán fundamentales para interpretar correctamente los comandos que ejecutaremos más adelante con `mongosh`.

8.1. Bases de datos, colecciones y documentos

Aunque ya lo he explicado anteriormente, esto es a modo de resumen.
MongoDB utiliza un modelo orientado a documentos. A diferencia de SQL, donde trabajamos con tablas y filas, en MongoDB usamos:

- **Bases de datos (databases)**: contienen colecciones.
- **Colecciones (collections)**: equivalentes a tablas, pero sin un esquema fijo.
- **Documentos (documents)**: equivalentes a filas, pero en formato JSON.

Un ejemplo de documento en MongoDB sería:

```
{
  "_id": "emp1",
  "nombre": "Laura",
  "edad": 28,
  "departamento": "Desarrollo",
  "direccion": {
    "ciudad": "Madrid",
    "calle": "Alcala 128"
  },
  "intereses": ["musica", "lectura"]
}
```

Como se observa, MongoDB permite:

- Arrays
- Subdocumentos
- Campos opcionales (cada documento puede tener distintas claves)

8.2. El campo obligatorio _id

Todos los documentos deben tener un campo **_id**, que actúa como identificador único dentro de la colección.

Si no lo indicamos, MongoDB genera uno automáticamente del tipo **ObjectId()**, por ejemplo:

```
ObjectId("65f0a1cb3d91f4c859c1ed21")
```

Sin embargo, también podemos definirlo manualmente:

```
_id: "emp1"
```

Ambas formas son válidas.

8.3. Conectar con el servidor desde `mongosh`

Una vez instalado `mongosh`, desde terminal o CMD ejecutamos `mongosh.exe`. Si la conexión es correcta, veremos un prompt del estilo:

```
test>
```

Este será el punto de partida para todas las operaciones que realizaremos a continuación.

8.4. El objeto `db`

El objeto `db` representa la base de datos actual sobre la que estamos trabajando. Cada vez que ejecutamos un comando del tipo:

```
db.coleccion.metodo()
```

estamos utilizando un método dentro de esa base de datos, por ejemplo:

- `db.createCollection()`
- `db.empleados.insertOne()`
- `db.departamentos.find()`

A partir de aquí comenzaremos a movernos por bases de datos, crear colecciones e insertar documentos.

9. Comandos básicos en `mongosh`

Una vez conectados a MongoDB con `mongosh`, podemos comenzar a trabajar con bases de datos y colecciones. En esta sección veremos los comandos esenciales para movernos dentro del entorno y preparar el terreno antes de insertar datos.

9.1. Cambiar o crear una base de datos

El comando `use` permite cambiar de base de datos. Si la base de datos no existe, MongoDB la creará automáticamente cuando insertes el primer documento.

```
use empresa
```

MongoDB responderá:

```
switched to db empresa
```

9.2. Mostrar todas las bases de datos

```
show dbs
```

Importante: la base de datos recién creada **no aparecerá** hasta que contenga al menos un documento.

9.3. Mostrar las colecciones de la base de datos actual

```
show collections
```

Si la base de datos está vacía, este comando no mostrará nada.

9.4. Crear colecciones de forma explícita

Aunque MongoDB crea colecciones automáticamente al insertar datos, también podemos crearlas manualmente:

```
db.createCollection("empleados")
db.createCollection("departamentos")
```

Podemos comprobar que se han creado:

```
show collections
```

9.5. Crear otra colección: `ciudades`

Continuando con el ejemplo, añadimos una colección más:

```
db.createCollection("ciudades")
```

Ahora deberíamos ver:

```
empleados
departamentos
ciudades
```

10. Operaciones CREATE: Insertar documentos

En MongoDB insertamos documentos dentro de colecciones usando métodos como `insertOne()` e `insertMany()`. A diferencia de SQL, no existe un esquema rígido: cada documento puede tener estructura distinta siempre que sea válido en formato JSON.

10.1. Insertar un documento con `insertOne()`

El método `insertOne()` sirve para insertar un único documento dentro de una colección.
Su sintaxis general es:

```
db.coleccion.insertOne({
  campo1: valor1,
  campo2: valor2,
  campo3: valor3
})
```

Los campos se escriben como en JSON, separados por comas y encerrados entre llaves `{}`.

Ejemplo práctico

```
db.empleados.insertOne({
  _id: "emp1",
  nombre: "Laura",
  apellido: "Gomez",
  edad: 28
})
```

Después podemos comprobarlo con:

```
db.empleados.find()
```

Ejercicio rápido

Inserta un empleado con los datos:

- _id: emp2
- nombre: el tuyo
- departamento: el que quieras
- aficiones: un array de aficiones

10.2. Insertar varios documentos con `insertMany()`

Cuando queremos insertar varios documentos a la vez utilizamos:

```
db.coleccion.insertMany([
  { campo1: v1, campo2: v2 },
  { campo1: v3, campo2: v4 }
])
```

Cada documento va dentro de un array y separado por comas.

Ejemplo: insertar varios departamentos

```
db.departamentos.insertMany([
  { _id: "dep2", nombre: "Recursos Humanos" },
  { _id: "dep3", nombre: "Finanzas" },
  { _id: "dep4", nombre: "Sistemas" }
])
```

Ejercicio rápido

Inserta 3 ciudades nuevas en la colección `ciudades`.

10.3. Insertar documentos con arrays y subdocumentos

MongoDB permite almacenar datos complejos gracias a que acepta estructuras JSON completas.

Ejemplo con array

```
db.productos.insertOne({
  cod_producto: 1,
  nombre: "Martillo X2",
  precio: 20,
  fabricantes: ["fab1", "fab2", "fab3"]
})
```

Ejemplo con subdocumento

```
db.fabricantes.insertOne({
  _id: 1,
  nombre: "fab1",
  direccion: {
    ciudad: "Buenos Aires",
    pais: "Argentina",
    calle: "Calle Pez 27",
    cod_postal: 29000
  }
})
```

Ejercicio rápido

Inserta un documento con:

- ciudad: “Madrid”
- coordenadas: lat: 40.4167, lng: -3.7033
- barrios: Centro", "Salamanca", Chamberí"

11. Práctica básica: ejercicios desde terminal

En esta práctica pondrás en práctica los comandos vistos en clase. Debes realizar cada tarea utilizando únicamente los conocimientos explicados anteriormente.

1. Crear la base de datos

Crea una base de datos llamada **tienda**. Comprueba si aparece en el listado de bases de datos al finalizar el ejercicio.

2. Crear colecciones

Dentro de la base de datos **tienda**, crea las siguientes colecciones:

- `clientes`
- `productos`
- `pedidos`

3. Insertar documentos simples

Inserta al menos dos clientes en la colección `clientes`. Cada cliente debe tener como mínimo:

- un identificador propio (_id)
- un nombre
- un email

4. Insertar un documento con array

En la colección `productos`, inserta un producto que contenga:

- un identificador
- un nombre
- un precio
- un array llamado `categorias` con dos o más valores

5. Insertar un documento con subdocumento

Inserta un cliente que incluya una dirección con la siguiente estructura:

- ciudad
- calle
- código postal

6. Insertar múltiples documentos a la vez

Inserta al menos dos productos adicionales en la colección `productos`, utilizando una sola instrucción.

7. Crear un pedido

Registra un pedido en la colección `pedidos` que contenga:

- un código de pedido (_id)
- el identificador de un cliente existente
- un array con los identificadores de uno o más productos
- una fecha

8. Comprobación final

Muestra por pantalla el contenido de todas las colecciones creadas:

- `clientes`
- `productos`
- `pedidos`

Verifica que todos los datos se han insertado correctamente.

12. Inserción masiva: colección `libros`

En esta sección vamos a cargar una colección completa de libros utilizando un script JSON que se proporciona como archivo adjunto.

El archivo contiene una instrucción `insertMany()` totalmente preparada para insertar todos los libros en una sola operación.

1. Archivo adjunto

Asegúrate de disponer del archivo:
Ábrelo para ver su contenido. No es necesario modificar nada: el archivo ya incluye la llamada completa a:

```
db.libros.insertMany({ ... })
```

2. Tarea: insertar los datos

Realiza los siguientes pasos:

1. Cambia a la base de datos que quieras utilizar para este ejercicio (por ejemplo, `libreria`).
2. Abre el archivo `libros.json`.
3. Copia todo su contenido.
4. Pégalo en `mongosh` y ejecútalo.
5. Comprueba que se ha creado la colección `libros`.
6. Verifica que se han insertado los 9 documentos.

3. Preparación para las consultas

Una vez cargados los libros, esta colección se utilizará en la siguiente sección para practicar operaciones READ (consultas).

13. Operaciones READ en MongoDB

En esta sección aprenderás a realizar consultas sobre la colección `libros`. Seguiremos un orden progresivo, empezando por las consultas más básicas y avanzando hacia combinaciones de condiciones y operadores.

13.1. Búsquedas básicas: método `find()`

El método `find()` devuelve los documentos que coinciden con un criterio de búsqueda. Si no se indica ningún criterio, devuelve todos los documentos:

```
db.libros.find()
```

Para buscar solo por igualdad, basta con indicar el campo y el valor:

```
db.libros.find({ editorial: "Planeta" })
```

Esto muestra todos los libros cuya editorial sea exactamente `Planeta`.

13.2. Operadores de comparación

MongoDB permite hacer comparaciones avanzadas mediante operadores. La siguiente tabla resume los operadores más usados:

Operador	Descripción
$eq	Igualdad
$gt	Mayor que
$gte	Mayor o igual
$in	Buscar un elemento en un array
$lt	Menor que
$lte	Menor o igual
$ne	Todos los elementos que no sean igual
$nin	Ninguno de los valores en el array

Figura 1: Operadores de comparación en MongoDB

Los operadores se usan así:

```
{ campo: { operador: valor } }
```

13.3. Ejemplos con operadores de comparación

$gt — mayor que

```
db.libros.find({ precio: { $gt: 20 } })
```

$lte — menor o igual que

```
db.libros.find({ cantidad: { $lte: 12 } })
```

$in — dentro de una lista

```
db.libros.find({ editorial: { $in: ["Planeta", "Biblio"] } })
```

13.4. Recuperar un único documento: `findOne()`

Si queremos obtener solo un documento (el primero que coincida), usamos:

```
db.libros.findOne({ editorial: "Planeta" })
```

Diferencias:

- `find()` → devuelve varios documentos.
- `findOne()` → devuelve solo uno.

13.5. Otros operadores útiles

$ne — distinto de

```
db.libros.find({ precio: { $ne: 25 } })
```

$nin — ninguno de estos valores

```
db.libros.find({ editorial: { $nin: ["Planeta", "Biblio"] } })
```

13.6. Operador `$and`

Se usa para combinar condiciones que deben cumplirse a la vez.
Ejemplo:

```
db.libros.find({
  $and: [
    { editorial: "Biblio" },
    { precio: { $gt: 20 } }
  ]
})
```

MongoDB también permite escribirlo sin $and:

```
db.libros.find({ editorial: "Biblio", precio: { $gt: 20 } })
```

13.7. Operador `$or`

Devuelve los documentos que cumplan al menos una condición.

```
db.libros.find({
  $or: [
    { editorial: "Planeta" },
    { precio: { $lt: 20 } }
  ]
})
```

13.8. Combinar `$and` y `$or`

Ejemplo: Libros de `Biblio` con precio entre 20 y 35:

```
db.libros.find({
  editorial: "Biblio",
  $and: [
    { precio: { $gte: 20 } },
    { precio: { $lte: 35 } }
  ]
})
```

Ejemplo combinando OR dentro de AND:

```
db.libros.find({
  $and: [
    { precio: { $in: [20, 25] } },
    { cantidad: { $gt: 10 } }
  ]
})
```

13.9. Mostrar solo ciertos campos

Podemos indicar qué campos queremos mostrar:

```
db.libros.find(
  {},
  { titulo: 1, precio: 1, _id: 0 }
)
```

Otro ejemplo:

```
db.libros.find(
  {},
  { titulo: 1, editorial: 1, cantidad: 1 }
)
```

13.10. Ejercicios finales

1. Libros cuyo precio sea exactamente 25.
2. Libros con precio mayor que 20 y cantidad menor que 15.
3. Libros cuya editorial sea `Planeta` o `Biblio`.
4. El libro de `Siglo XXI` más barato (usar sort + limit).
5. Libros cuyo precio no sea ni 20, ni 25, ni 50.
6. Libros entre 20 y 30 euros.
7. Mostrar solo título, editorial y precio.
8. El primer libro con cantidad mayor que 10 (usar findOne()).

Importación de datos JSON en MongoDB (Windows)

Para poder importar archivos `JSON` o `JSONL/NDJSON` en MongoDB utilizando el comando `mongoimport`, es necesario instalar previamente las **MongoDB Database Tools**.

1. Descarga de MongoDB Database Tools

Las herramientas oficiales se pueden descargar desde la página de MongoDB:

`https://www.mongodb.com/try/download/database-tools`

En dicha página se debe seleccionar:

- Sistema operativo: **Windows**
- Formato: **ZIP**

Una vez descargado el archivo, se debe descomprimir en una carpeta del sistema, por ejemplo:

```
C:\mongodb-tools\
```

Dentro de esta carpeta se encontrará el ejecutable `mongoimport.exe`.

2. Configuración de la variable de entorno PATH

Para poder ejecutar el comando `mongoimport.exe` desde cualquier ventana de comandos, es necesario añadir la ruta de las herramientas a la variable de entorno `PATH` de Windows.

Los pasos son los siguientes: Pasos:

1. Abrir el menú Inicio y escribir: Editar las variables de entorno del sistema.
2. Pulsar en el botón: **Variables de entorno**.
3. En **Variables del sistema**, seleccionar la variable `Path`.
4. Hacer clic en **Editar**.

   ```
   C:\mongodb-tools\bin
   ```

5. Aceptar todos los cambios y cerrar las ventanas.

Importante: es necesario cerrar y volver a abrir la ventana de comandos para que los cambios tengan efecto.

3. Comprobación de la instalación

Para comprobar que las herramientas se han instalado correctamente, se puede ejecutar el siguiente comando en la consola de Windows:

```
mongoimport.exe --version
```

Si se muestra la versión instalada, la configuración es correcta y ya se puede proceder a la importación de archivos JSON en MongoDB.

Importación del fichero `movies.json` en MongoDB

Para importar el fichero `movies.json` (en formato `JSONL/NDJSON`) en MongoDB, se recomienda crear previamente una base de datos específica y una colección asociada. En este caso, se utilizará la base de datos `cine` y la colección `peliculas`.

Comando de importación

Desde una consola de Windows, situada en la carpeta donde se encuentra el fichero `movies.json`, se debe ejecutar el siguiente comando:

```
mongoimport.exe --db cine --collection peliculas --file movies.json (ruta donde
se encuentre situado el fichero)
```

Dado que el fichero está en formato `NDJSON` (un documento JSON por línea), **no se debe utilizar la opción** `-jsonArray`.

Comprobación de la importación

Una vez finalizado el proceso, se puede comprobar que los datos se han importado correctamente accediendo a la consola de MongoDB y ejecutando los siguientes comandos:

```
use cine
db.peliculas.countDocuments()
db.peliculas.findOne()
```

El número de documentos debería de ser unos `23539`.

Ejercicio: Consultas sobre la colección `peliculas`

Una vez importado el fichero `movies.json` en la base de datos `cine` y la colección `peliculas`, se pide realizar las siguientes consultas utilizando el método `find()` de MongoDB.

1. Buscar las películas que sean de género **Drama** o **Romance**, que tengan una duración inferior a **100 minutos** y cuya valoración **IMDb** sea mayor o igual que **7**.
2. Buscar las películas dirigidas por **Steven Spielberg** o **Martin Scorsese**, pero que no estén producidas en **Estados Unidos**, y mostrar **únicamente el título de la película**.
3. Obtener las películas producidas en **Estados Unidos** o **España**, cuyo idioma sea **inglés** o **español**, y que se hayan estrenado entre los años **1990 y 2010** (ambos inclusive).
4. Obtener las películas en las que actúe **Tom Hanks** o **Leonardo DiCaprio**, cuyo idioma sea **español**, y mostrar **únicamente el título de la película**.
5. Mostrar las películas que tengan una valoración **IMDb mayor o igual que 8** o una puntuación del **crítico en Rotten Tomatoes mayor o igual que 90**, y que además tengan **al menos 10 000 votos en IMDb**.
6. Buscar las películas dirigidas por **Alfred Hitchcock** que no estén clasificadas como **G** o **PG**, y mostrar **únicamente el año de estreno**.
7. Listar las películas que **no sean de género Comedy**, que tengan una duración comprendida entre **80 y 120 minutos** y cuya clasificación por edades sea **PG-13 o R**.
8. Mostrar las películas europeas, producidas en **España**, **Francia** o **Italia**, cuyo idioma sea **español** o **francés**, y mostrar **únicamente el título y el país de producción**.
9. Seleccionar las películas que hayan **ganado tres o más premios** y que tengan **como máximo una nominación**.
10. Obtener las películas de tipo **movie** dirigidas por **Christopher Nolan** o **Denis Villeneuve**, y mostrar **únicamente el título y el director**.
11. Buscar las películas cuya **valoración IMDb sea superior a la valoración de los espectadores en Rotten Tomatoes** y que además tengan **más de 1 000 votos en IMDb**. *Pista: en esta consulta se comparan dos campos del mismo documento. Investiga cómo funciona el operador `$expr` y utilízalo para resolverla.*
12. Obtener las películas en las que la **puntuación del crítico en Rotten Tomatoes** sea mayor o igual que la **puntuación de los espectadores**, y cuya **valoración IMDb sea mayor o igual que 7**. *Pista: en esta consulta se comparan dos campos del mismo documento. Investiga cómo funciona el operador `$expr` y utilízalo para resolverla.*

13. Mostrar las películas producidas en **Estados Unidos** o **Reino Unido** cuyo idioma **no sea inglés**, y mostrar **únicamente el título y el idioma**.
14. Mostrar las películas que tengan **al menos un comentario** o **más de 5 000 votos en IMDb**, y cuya valoración **IMDb esté comprendida entre 6,5 y 8,5**.
15. Buscar las películas dirigidas por **King Vidor** o **Alfred Hitchcock** que tengan una valoración **IMDb mayor o igual que 7,5** y que además sean **anteriores a 1970** o tengan una duración **superior a 140 minutos**.
16. Listar las películas que tengan **50 000 votos o más en IMDb**, pero cuya valoración **IMDb esté en un rango medio**, entre **6 y 7,2**.

Consultas con muchos resultados

Todas las consultas de este ejercicio deben realizarse utilizando el método `find()` de MongoDB. En algunos casos, una consulta puede devolver muchos documentos y la salida por pantalla puede resultar demasiado larga o difícil de leer. Cuando esto ocurra, no es necesario mostrar todos los resultados.

En su lugar, se puede utilizar la función `count()` únicamente para saber cuántos documentos cumplen la condición, sin mostrar el contenido completo.

Ejemplo

La siguiente consulta obtiene las películas con una valoración IMDb mayor o igual que 7:

```
db.peliculas.find({
  "imdb.rating": { $gte: 7 }
})
```

Si aparecen demasiados resultados por pantalla, se puede mostrar solo el número total de películas de la siguiente forma:

```
db.peliculas.find({
  "imdb.rating": { $gte: 7 }
}).count()
```

De esta manera, se sigue utilizando `find()` para hacer la consulta, pero se evita saturar la consola cuando hay muchos resultados.

14. De `find()` a `aggregate()`

Hasta ahora todas las consultas que hemos realizado en MongoDB se han basado en el método `find()`. Este método nos permite buscar documentos dentro de una colección, aplicando condiciones y mostrando únicamente los campos que nos interesen.

Cuando utilizamos `find()`, el funcionamiento es siempre el mismo:

- MongoDB filtra documentos según una condición.
- Devuelve documentos existentes en la colección.
- El resultado mantiene la estructura original del documento.

Este tipo de consultas es suficiente cuando queremos ver datos concretos, pero se queda corto cuando necesitamos analizar información, por ejemplo:

- calcular totales,
- obtener medias,
- agrupar resultados,
- o clasificar documentos según ciertos criterios.

Para este tipo de operaciones MongoDB proporciona el método `aggregate()`.

14.1. Qué es `aggregate()`

El método `aggregate()` permite trabajar con los documentos mediante un conjunto de etapas encadenadas, lo que se conoce como un **pipeline**.

En lugar de ejecutar una única consulta, los documentos:

- entran en una primera etapa,
- se filtran o transforman,
- pasan a la siguiente etapa,
- y así sucesivamente hasta obtener un resultado final.

La diferencia fundamental con respecto a `find()` es que el resultado de `aggregate()`:

- no tiene por qué ser un documento original de la colección,
- puede ser un resumen,
- un recuento,
- o información calculada que no existe almacenada directamente.

14.2. Esquema visual del funcionamiento

El siguiente esquema resume de forma gráfica la diferencia entre ambos métodos.

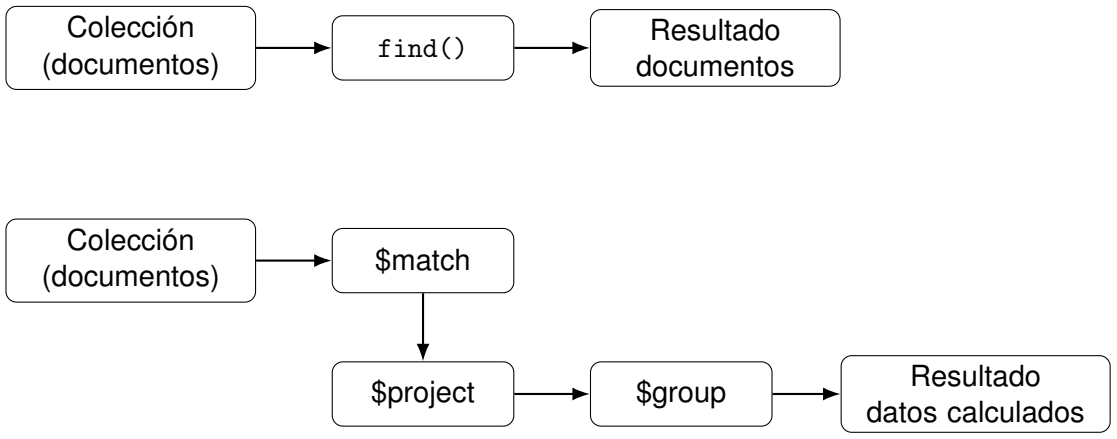

Figura 2: Con `find()` se devuelven documentos. Con `aggregate()` los documentos se transforman paso a paso.

A partir de ahora utilizaremos `aggregate()` para realizar consultas más avanzadas. Para entenderlo mejor, comenzaremos con ejemplos usando la colección `libros`.

14.3. Primeros ejemplos con la colección `libros`

La colección `libros` es adecuada para empezar a trabajar con `aggregate()` porque contiene pocos campos y los resultados son fáciles de interpretar.

Filtrar documentos con `$match`

La etapa `$match` se utiliza para filtrar documentos dentro del pipeline. Conceptualmente realiza la misma función que el filtro de `find()`, pero permite seguir trabajando con los documentos en etapas posteriores.

Si queremos quedarnos únicamente con los libros cuyo precio sea mayor o igual que 25, la consulta sería:

```
db.libros.aggregate([
  { $match: { precio: { $gte: 25 } } }
])
```

El resultado sigue siendo una lista de libros completos, pero ahora esos documentos forman parte de un pipeline y pueden seguir transformándose.

Seleccionar campos con `$project`

Una vez filtrados los documentos, es habitual querer mostrar solo algunos campos. La etapa `$project` se encarga de definir qué información se devuelve.

Por ejemplo, si solo nos interesa ver el título y el precio de cada libro:

```
db.libros.aggregate([
  { $project: { _id: 0, titulo: 1, precio: 1 } }
])
```

En este caso no se eliminan libros, sino que se modifica la forma en la que se presentan los datos, ocultando los campos que no nos interesan.

Crear campos calculados

Una de las ventajas de `aggregate()` es la posibilidad de crear campos nuevos a partir de los existentes.

Si queremos conocer el valor total del stock de cada libro, podemos multiplicar el precio por la cantidad disponible y añadir ese resultado como un campo nuevo:

```
db.libros.aggregate([
  {
    $project: {
      titulo: 1,
      precio: 1,
      cantidad: 1,
      valorStock: { $multiply: ["$precio", "$cantidad"] }
    }
  }
])
```

Los documentos que se devuelven siguen representando libros, pero ahora incluyen información calculada que no está almacenada directamente en la base de datos.

Agrupar documentos con `$group`

Hasta ahora hemos trabajado con libros de forma individual. La etapa `$group` permite agrupar varios documentos y obtener información resumida.

Si agrupamos los libros por editorial y contamos cuántos hay en cada una, obtenemos una visión general de la colección:

```
db.libros.aggregate([
  { $group: { _id: "$editorial", totalLibros: { $sum: 1 } } }
])
```

A partir de este punto, el resultado ya no son libros concretos, sino un resumen por editorial.

También podemos aprovechar la agrupación para calcular valores como el precio medio:

```
db.libros.aggregate([
  {
    $group: {
      _id: "$editorial",
      precioMedio: { $avg: "$precio" }
    }
  }
])
```

Ordenar el resultado

Una vez obtenidos valores agregados, suele ser útil ordenarlos. Para ello se utiliza la etapa `$sort`.

Por ejemplo, podemos ordenar las editoriales según su precio medio de mayor a menor:

```
db.libros.aggregate([
  {
    $group: {
      _id: "$editorial",
      precioMedio: { $avg: "$precio" }
    }
  },
  { $sort: { precioMedio: -1 } }
])
```

Contar documentos

En ocasiones no necesitamos ver los documentos, sino únicamente saber cuántos cumplen una condición. En estos casos podemos utilizar la etapa `$count`.

Por ejemplo, para saber cuántos libros tienen un precio superior a 20 euros:

```
db.libros.aggregate([
  { $match: { precio: { $gt: 20 } } },
  { $count: "total" }
])
```

El resultado es un único documento con el número total de libros que cumplen la condición indicada.

14.4. Trabajar con condiciones en `aggregate()`

Además de filtrar o agrupar, con `aggregate()` también podemos crear campos que dependan de una condición. Esto viene muy bien para etiquetar o clasificar documentos.

Condición simple con `$cond`

El operador `$cond` funciona de forma parecida a un `if/else`: si se cumple una condición, se asigna un valor; si no, se asigna otro.

Por ejemplo, podemos añadir un campo que indique si un libro es caro o barato en función de su precio (en este caso, consideramos caro a partir de 30 euros):

```
db.libros.aggregate([
  {
    $project: {
      titulo: 1,
      precio: 1,
      tipoPrecio: {
        $cond: {
          if: { $gte: ["$precio", 30] },
          then: "caro",
          else: "barato"
        }
      }
    }
  }
])
```

El resultado sigue siendo una lista de libros, pero ahora cada documento incluye un campo nuevo (`tipoPrecio`) con una etiqueta.

Varias condiciones con `$switch`

Cuando necesitamos más de dos casos, utilizamos `$switch`. La idea es comprobar varias condiciones en orden y quedarnos con la primera que se cumpla.

Por ejemplo, podemos clasificar los libros en:

- `barato` si el precio es menor que 20,
- `precio medio` si está entre 20 y 30,
- `caro` si es 30 o más.

```
db.libros.aggregate([
  {
    $project: {
      titulo: 1,
      precio: 1,
      categoriaPrecio: {
        $switch: {
          branches: [
            { case: { $lt: ["$precio", 20] }, then: "barato" },
            { case: { $and: [
                { $gte: ["$precio", 20] },
                { $lt: ["$precio", 30] }
              ]}, then: "precio medio" }
          ],
          default: "caro"
        }
      }
    }
  }
])
```

De nuevo, el resultado son libros, pero ahora con una clasificación más completa.

Usar una clasificación para resumir datos

Una vez creado un campo de clasificación, podemos agrupar por ese campo para obtener un resumen.

Por ejemplo, contar cuántos libros hay en cada categoría de precio:

```
db.libros.aggregate([
  {
    $project: {
      categoriaPrecio: {
        $switch: {
          branches: [
            { case: { $lt: ["$precio", 20] }, then: "barato" },
            { case: { $and: [
                { $gte: ["$precio", 20] },
                { $lt: ["$precio", 30] }
              ]}, then: "precio medio" }
          ],
          default: "caro"
        }
      }
    }
  },
  {
    $group: {
      _id: "$categoriaPrecio",
      totalLibros: { $sum: 1 }
    }
  }
])
```

Aquí el resultado ya no son libros, sino un resumen con el número de libros en cada categoría.

15. Ejercicios con `aggregate()`

Ejercicios con la colección `libros`

En los primeros ejercicios se repasan las etapas básicas de `aggregate()`. En los últimos se utilizan condiciones de una forma distinta a los ejemplos, para obligarnos a pensar cómo aplicar la lógica dentro del pipeline.

1. Mostrar los libros cuyo precio sea mayor o igual que 25.
2. Mostrar únicamente el título y el precio de todos los libros.
3. Mostrar el título, el precio, la cantidad y el campo calculado `valorStock`, cuyo valor se obtiene de la multiplicación entre precio y cantidad.
4. Calcular cuántos libros hay por editorial.
5. Calcular el precio medio de los libros por editorial y ordenar el resultado de menor a mayor.
6. Para cada libro, crear un campo llamado `stockAlto` que valga `true` si la **cantidad de ese libro** es mayor o igual que 20, y `false` en caso contrario. *(No hay que agrupar, solo añadir el campo).*
7. Para cada libro, crear un campo llamado `estadoLibro` que indique:
   - `agotado` si la cantidad es 0,
   - `pocas unidades` si la cantidad es menor que 5,
   - `disponible` en cualquier otro caso.
8. Para cada libro, clasificarlo como `rentable` o `no rentable` en función de si el valor del stock (`precio` multiplicado por `cantidad`) es mayor o igual que 500.
9. Contar cuántos libros hay de cada tipo de `estadoLibro`. *(*crea primero el campo con una condición y después agrupa.)

Ejercicios con la colección `peliculas`

Aplica las mismas ideas vistas con libros a la colección `peliculas`.

1. Contar cuántas películas hay a partir del año 2000.
2. Mostrar el título, el año y el número de géneros de cada película. *(Recuerda: `genres` es un array.)*
3. Mostrar los 10 años con más películas.
4. Clasificar las películas según su duración y contar cuántas hay de cada tipo, considerando:
   - **corta**: duración menor que 90 minutos,
   - **media**: duración entre 90 y 119 minutos,
   - **larga**: duración mayor o igual que 120 minutos.
5. Mostrar el título, el año y un campo llamado `epoca` que indique:
   - `clasica` si el año es anterior a 1980,
   - `moderna` si está entre 1980 y 1999,
   - `actual` si es del año 2000 o posterior.
6. Mostrar los **10 géneros** con más películas (como `genres` es un array, tendrás que usar `$unwind` antes de agrupar.)
7. Mostrar los **10 directores** con más películas. (`directors` también es un array, así que vuelve a aparecer `$unwind`.)
8. Clasificar las películas según su popularidad por número de votos en IMDb y mostrar el **título**, los **votos** y la etiqueta asignada. Utiliza estas reglas:
   - **poco votada**: menos de 1 000 votos,
   - **popular**: entre 1 000 y 50 000 votos,
   - **muy popular**: más de 50 000 votos.
9. Contar cuántas películas hay de cada tipo de popularidad (`poco votada`, `popular`, `muy popular`). (crea primero la etiqueta y después agrupa por ella.)

16. Relaciones entre documentos en MongoDB

En el modelo relacional (SQL) las relaciones forman parte del **núcleo** del diseño: tablas conectadas por claves primarias y foráneas, y consultas que combinan datos con `JOIN`.

En MongoDB (NoSQL documental) las relaciones **siguen existiendo**, pero el motor no obliga a representarlas con claves foráneas ni a combinarlas automáticamente. En su lugar, el diseñador decide cómo reflejar la relación en el modelo de documentos.

16.1. Entidad y relación: la idea antes de tocar código

Una **entidad** es una representación de un concepto del mundo real (por ejemplo, *Empleado* o *Departamento*). Antes de hablar de relaciones, conviene ver que existen entidades independientes:

Figura 3: Entidades (Empleado y Departamento) como conceptos independientes

Una **relación** describe cómo se conectan esas entidades dentro del sistema:

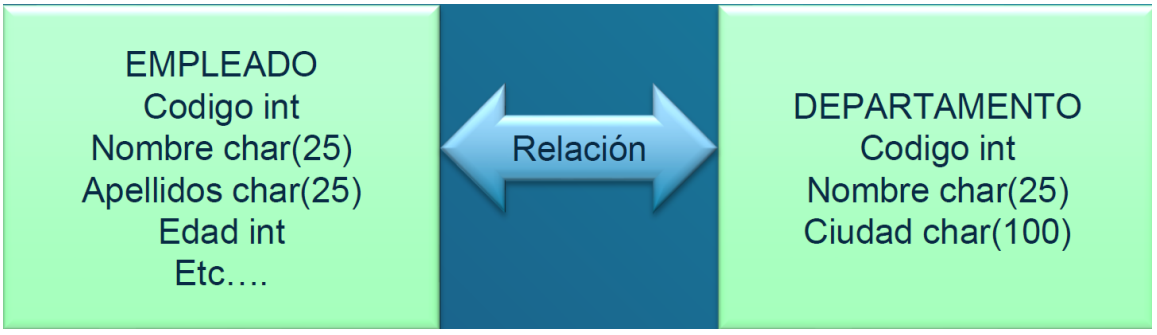

Figura 4: Una relación conecta entidades (idea general)

16.2. Tipos de relaciones

Al igual que en SQL, al estudiar la realidad que queremos representar con datos, vemos que unas cosas dependen de otras y se relacionan entre sí.

- **Uno a Uno (1:1)**.
- **Uno a Muchos (1:N)**.
- **Muchos a Muchos (N:M)**.

La diferencia clave no es el tipo de relación (que es el mismo), sino **cómo se implementa** en MongoDB:

- **Embebida**: la información relacionada vive dentro del mismo documento.
- **Referenciada**: la información vive en colecciones distintas y se conecta mediante campos (IDs / códigos).

16.3. Regla práctica: ¿embebo o referencio?

MongoDB suele funcionar mejor cuando el documento se diseña pensando en **cómo se consulta**.

- **Embebido** si se consulta casi siempre “junto”, el tamaño es razonable y el crecimiento está controlado.
- **Referenciado** si el lado “muchos” puede crecer mucho, si hay reutilización o si se actualiza por separado con frecuencia.

17. Relación Uno a Uno (1:1): Departamento y Responsable

17.1. Qué significa (concepto)

Una relación 1:1 aparece cuando un elemento de una entidad se asocia con **un único** elemento de otra entidad. Ejemplo: un departamento tiene un responsable y un responsable dirige un departamento.

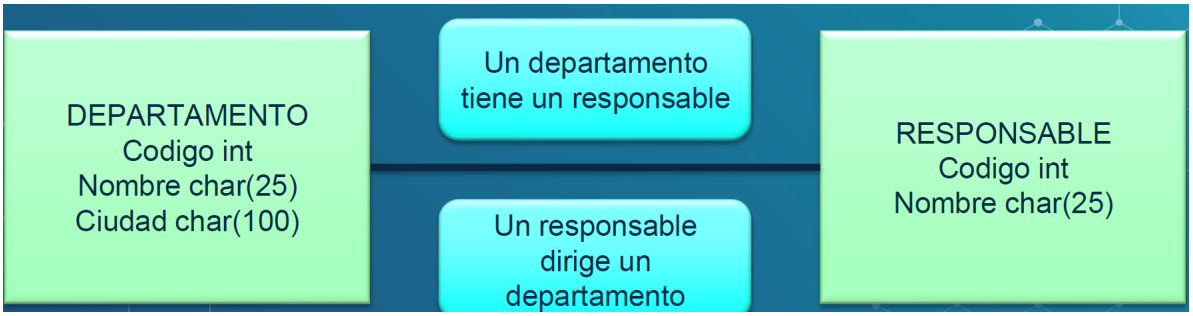

Figura 5: Relación 1:1: Departamento y Responsable

17.2. Semejanza con SQL

En SQL, una 1:1 suele resolverse con una clave foránea y una restricción de unicidad para asegurar “uno a uno”.

17.3. MongoDB: modelo embebido (lo más natural en 1:1)

Si el responsable se consulta siempre junto con el departamento (muy común en interfaces y listados), lo normal es embebido: el responsable es un subdocumento.

```
db.departamentos.insertMany([
{
  _id: 1,
  nombre: "Informática",
  responsable: {
    nombre: "Pedro",
    apellidos: "Ramirez Perez",
    edad: 35
  },
  descripcion: "ejemplo de departamento"
},
{
  _id: 2,
  nombre: "Recursos Humanos",
  responsable: {
    nombre: "Raul",
    apellidos: "García López",
```

```
    edad: 42
  },
  descripcion: "ejemplo de departamento"
}
])
```

```
db.departamentos.find(
  {},
  { _id: 0, nombre: 1, "responsable.nombre": 1, "responsable.apellidos": 1 }
)
```

17.4. MongoDB: modelo referenciado (cuando tiene sentido)

Aunque en 1:1 el embebido suele ser la opción más sencilla, a veces interesa separar:

- si el responsable es una entidad reutilizable (por ejemplo, también aparece como empleado en otra colección),
- si tiene mucha información adicional,
- o si debe gestionarse por separado con permisos distintos.

En ese caso, se guarda un campo en el departamento (por ejemplo, `cod_responsable`) y el responsable vive en otra colección. La "unión" se hace en la aplicación o con una agregación `$lookup` cuando sea necesario como ya veremos.

18. Relación Uno a Muchos (1:N): Departamento y Empleados

18.1. Qué significa (concepto)

Una relación 1:N aparece cuando un elemento de la entidad "uno" se asocia con **muchos** elementos de la entidad del lado "muchos". Ejemplo clásico: un departamento tiene varios empleados; un empleado pertenece a un departamento.

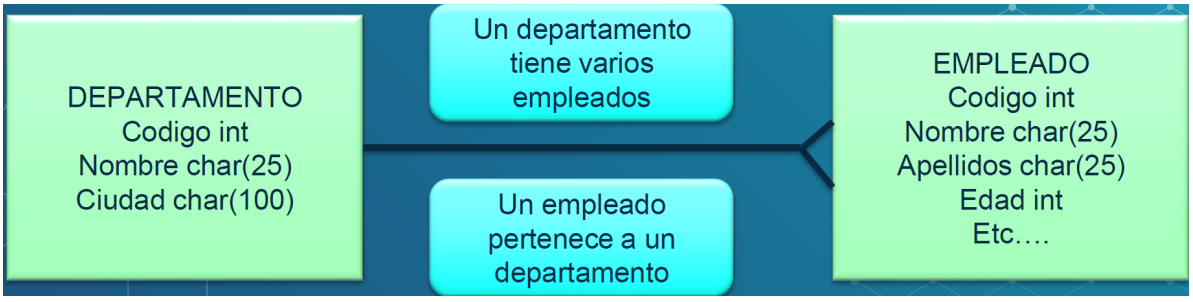

Figura 6: Relación 1:N: Departamento y Empleados

18.2. Semejanza con SQL

En SQL, se implementa con una clave foránea en el lado "muchos":

$$\texttt{empleados.cod\_departamento} \rightarrow \texttt{departamentos.id}$$

y las consultas conjuntas se resuelven con `JOIN`.

18.3. MongoDB: dos formas de modelarlo

18.3.1. Modelo embebido

En el enfoque embebido, los empleados vivirían como un array dentro del departamento. Esto hace que leer un departamento traiga "todo lo que necesito" en una sola lectura.

```
db.departamentos.insertOne({
  _id: 10,
  nombre: "Marketing",
  ciudad: "Sevilla",
  empleados: [
    { cod_empleado: 501, nombre: "Ana", apellidos: "López", edad: 29 },
    { cod_empleado: 502, nombre: "Luis", apellidos: "Pérez", edad: 41 }
  ]
})
```

Consulta

```
db.departamentos.find(
  { _id: 10 },
  { _id: 0, nombre: 1, "empleados.nombre": 1, "empleados.apellidos": 1 }
)
```

18.3.2. Modelo referenciado (cuando el lado “muchos” crece o se gestiona aparte)

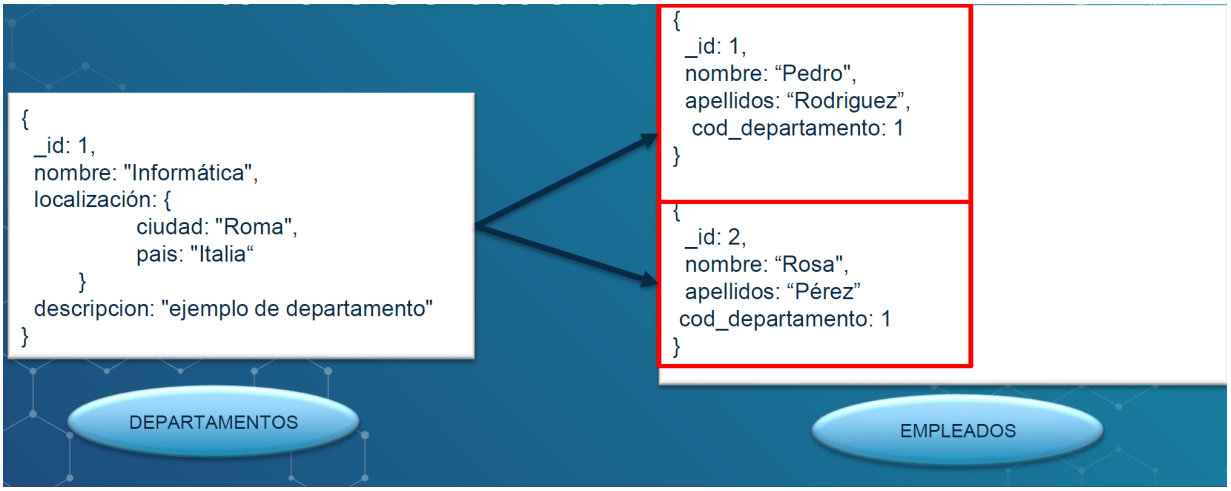

Figura 7: Ejemplo visual de modelo referenciado: colecciones separadas y campo de referencia

En este enfoque, los departamentos viven en una colección y los empleados en otra. Los empleados guardan el `cod_departamento` como referencia.

Documentos (departamentos y empleados en colecciones distintas)

```
db.departamentos.insertMany([
  { _id: 1, nombre: "Informática", ciudad: "Roma", descripcion: "ejemplo de dep" },
  { _id: 2, nombre: "Recursos Humanos", ciudad: "Roma", descripcion: "ejemplo de dep" }
])
```

```
db.empleados.insertMany([
  { _id: 1, nombre: "Pedro", apellidos: "Rodriguez", cod_departamento: 1 },
  { _id: 2, nombre: "Rosa", apellidos: "Pérez", cod_departamento: 1 }
])
```

18.4. Reconstruyendo la relación en consulta: `$lookup` (JOIN en MongoDB)

Como la relación no está "unida" físicamente, si queremos ver departamentos con sus empleados en un mismo resultado, usamos una agregación con `$lookup`. A nivel conceptual, se parece a un `JOIN`, pero ocurre dentro del pipeline.

Lookup + Project (departamentos con empleados)

```
db.departamentos.aggregate( [
  {
    $lookup: {
      from: "empleados",
      localField: "_id",
      foreignField: "cod_departamento",
      as: "empleados"
    }
  },
  {
    $project: {
      _id: 0,
      nombre: 1,
      "empleados.nombre": 1,
      "empleados.apellidos": 1
    }
  }
] )
```

19. Relación Muchos a Muchos (N:M): Profesores y Cursos

19.1. Qué significa (concepto)

Una relación N:M aparece cuando varios elementos de una entidad se relacionan con varios elementos de otra. Ejemplo: un profesor imparte varios cursos y un curso tiene varios profesores.

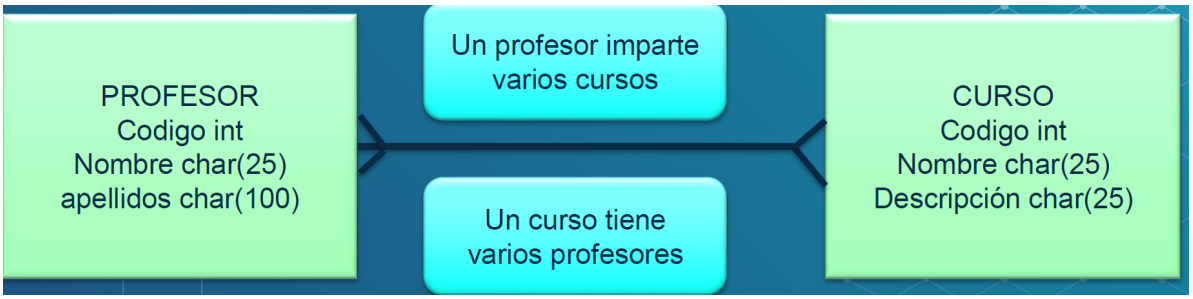

Figura 8: Relación N:M: Profesor y Curso

19.2. Semejanza con SQL: tabla intermedia

En SQL, el modelo típico es crear una tabla intermedia (tabla puente) para representar N:M:

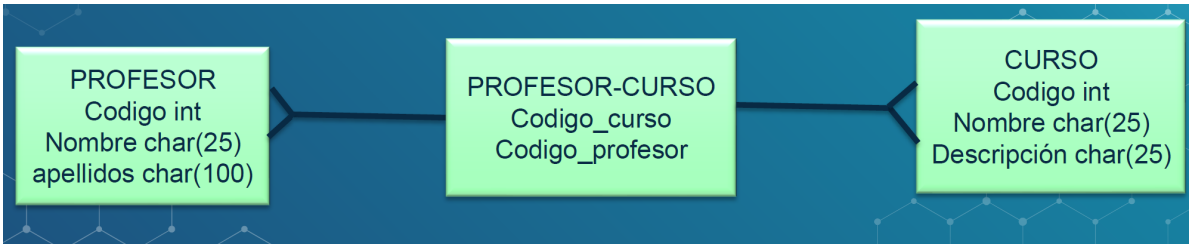

Figura 9: N:M en SQL usando entidad intermedia (tabla puente)

19.3. MongoDB: representaciones habituales

En MongoDB hay varias estrategias. Una de las más comunes es:

- guardar en **Cursos** un array con los IDs de los profesores, y
- recuperar el detalle con `$lookup`.

19.3.1. Documentos reales (profesores y cursos)

Colección `profesores`

```
db.profesores.insertMany([
  { _id: 101, nombre: "Profesor1", apellidos: "Apellidos1", edad: 45, salario: 1000 },
  { _id: 102, nombre: "Profesor2", apellidos: "Apellidos2", edad: 47, salario: 1200 },
  { _id: 103, nombre: "Profesor3", apellidos: "Apellidos3", edad: 49, salario: 1500 }
])
```

Colección `cursos` con array de IDs

```
db.cursos.insertMany([
  { _id: 1, nombre: "Curso1", aula: "aula1", profesores: [101, 102] },
  { _id: 2, nombre: "Curso2", aula: "aula2", profesores: [101, 103] },
  { _id: 3, nombre: "Curso3", aula: "aula3", profesores: [102, 101, 101] }
])
```

19.3.2. Lookup en N:M (curso → profesores)

MongoDB permite que `localField` sea un array; en ese caso, el lookup busca coincidencias de `foreignField` contra cualquiera de los valores del array.

```
db.cursos.aggregate([
  {
    $lookup: {
      from: "profesores",
      localField: "profesores",
      foreignField: "_id",
      as: "Profesores"
    }
  }
])
```

20. Embebido vs Referenciado: ejemplo clásico con Localidades

Para fijar el concepto, un ejemplo muy habitual es almacenar datos de localización. Si cada departamento tiene una localización “propia” y se consulta junto con el departamento, es natural embeberla.

20.1. Embebido (localización como subdocumento)

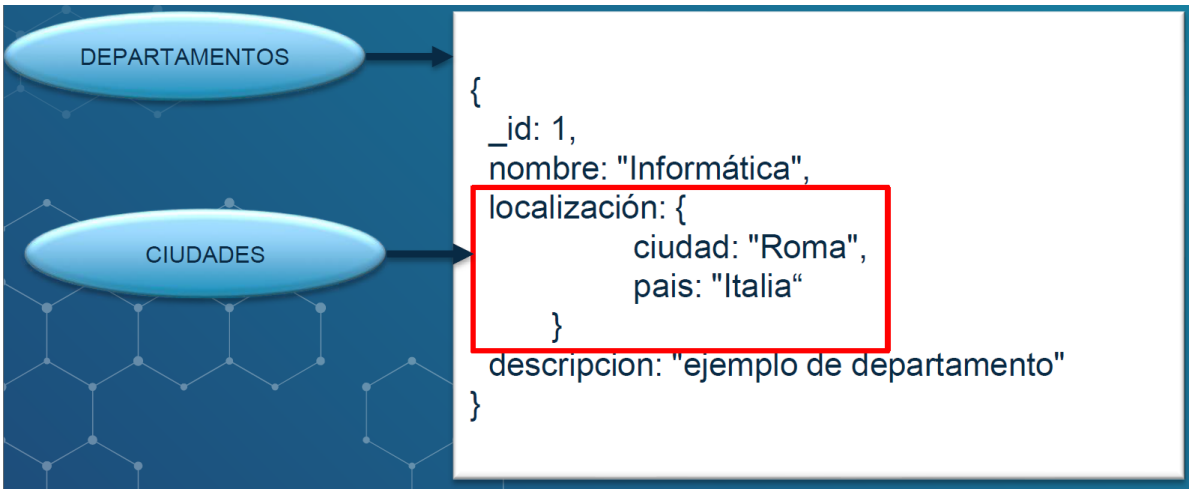

Figura 10: Localización embebida dentro de Departamento

```
db.departamentos.insertOne({
  _id: 20,
  nombre: "Diseño",
  localizacion: { ciudad: "Roma", pais: "Italia" },
  descripcion: "ejemplo de departamento"
})
```

20.2. Referenciado (localidades en otra colección)

Si las localidades se gestionan aparte, o hay muchas localidades asociadas a un departamento, o necesitas consultarlas independientemente, puedes separarlas en otra colección.

```
db.localidades.insertMany([
{
  _id: "BA",
  ciudad: "Buenos Aires",
  pais: "Argentina",
  poblacion: "16 millones",
  turismo: ["edificios","tango","gastronomia","museos","parques"],
  direccion: "Avenida ejemplo 1",
```

```
  cod_departamento: 1
},
{
  _id: "SA",
  ciudad: "Santiago",
  pais: "Chile",
  poblacion: " millones",
  turismo: ["iglesias","vino","gastronomia","museos"],
  direccion: "Calle Pez 23",
  cod_departamento: 2
}])
```

20.3. Lookup de departamentos con localidades

```
db.departamentos.aggregate( [
  {
    $lookup: {
      from: "localidades",
      localField: "_id",
      foreignField: "cod_departamento",
      as: "localidades"
    }
  },
  { $project: { _id: 0, nombre: 1, "localidades.ciudad": 1 } }
] )
```

21. Vistas: cuando una relación se consulta muy a menudo

A veces una consulta con `$lookup` se convierte en “la consulta de siempre” (informes, listados, paneles). MongoDB permite crear una **vista** que guarda esa agregación como un objeto consultable, como si fuera una colección.

21.1. Ejemplo de vista (consulta reutilizable)

```
db.createView(
  "datos_viajes",
  "conductores",
  [
    {
      $lookup: {
        from: "viajes",
        localField: "dni",
        foreignField: "cod_conductor",
        as: "viajes"
      }
    },
    { $project: { _id: 0, nombre: 1, "viajes.kilometros": 1 } }
  ]
)
```

Una vez creada la vista, se consulta con `find` como si fuese una colección:

```
db.datos_viajes.find()
```

22. Ejercicio: diseño del modelo de datos en MongoDB

Vamos a trabajar el diseño del modelo de datos de una aplicación interna para una empresa. La idea no es aplicar una solución única, sino **pensar cómo modelar los datos en MongoDB según el uso real que se hará de ellos**.

La empresa quiere una aplicación para gestionar su organización y la formación interna. En ella se manejará información sobre las distintas áreas de la empresa, las personas que trabajan en ellas, los cursos de formación y el personal que imparte dichos cursos.

De cada área interesa guardar información básica como el nombre y la ubicación, además de algunos datos de gestión. De las personas se almacenarán datos personales y laborales. También se quiere gestionar la información de los cursos y de quienes los imparten.

*La aplicación está pensada principalmente para **consultar información**. En la mayoría de los casos no se muestran datos sueltos, sino **información relacionada en una misma pantalla**. Por ejemplo, es habitual consultar una área de la empresa y ver las personas que trabajan en ella, o consultar una persona y ver a qué área pertenece.*

En el caso de la formación ocurre algo parecido. Es muy frecuente listar cursos mostrando quién los imparte, así como consultar a un formador y ver en qué cursos participa.
*Estas consultas **se repiten constantemente** en el uso diario de la aplicación y conviene tenerlas en cuenta a la hora de diseñar el modelo.*

Con esta información, la tarea consiste en **pensar qué elementos deben existir en la base de datos**, **cómo se relacionan entre sí** y **cómo conviene estructurar los documentos y las colecciones en MongoDB**. Es importante fijarse en **qué datos se consultan juntos**, **cuáles pueden crecer con el tiempo** y **qué información tiene sentido mantener unida**.

Además de proponer el modelo, se pueden crear **ejemplos de documentos JSON** que representen la solución planteada. En los casos en los que una consulta combinada se utilice de forma muy habitual, puede tener sentido usar **agregaciones** y, si se considera útil, **definir alguna vista** para reutilizar esas consultas.

No hay una única solución correcta. Lo importante es que **el modelo tenga sentido**, y que las decisiones tomadas **se puedan explicar con lógica**.

22.1. Entrega

La entrega consistirá en uno o varios archivos generados desde Visual Studio Code utilizando la extensión de MongoDB que ya hemos visto en clase.

La idea es que el archivo se pueda abrir y ejecutar sin modificaciones, de forma que sea posible crear las colecciones, insertar los documentos y probar las consultas directamente