Introducción a MongoDB

Luis Valencia Cabrera (lvalencia@us.es)

Research Group on Natural Computing
Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

26-11-2025, Bases de Datos

Índice

Índice

MongoDB

- SGBD (de código abierto y multiplataforma) NoSQL ***orientado a documentos***, de esquema libre.
- Cada **registro** se denomina ***documento***.
- Los documentos se pueden agrupar en ***colecciones*** (como las tablas en una BD relacional, pero almacenando documentos muy diferentes, **sin esquema fijo**).
- Se pueden crear índices para atributos, manteniendo estructura interna eficiente para el acceso a los mismos.
- Destaca por su **velocidad** y su **rico pero sencillo sistema de consulta** de los contenidos de la base de datos.
- Buen equilibrio entre rendimiento y funcionalidad

Formato de los datos

- Los distintos **documentos** se almacenan en formato **BSON**, o ***Binary JSON***
- Versión modificada y enriquecida de JSON que permite búsquedas rápidas de datos.
- Guarda de forma explícita longitudes de campos, índices de arrays, y otra información útil para el escaneo de datos $\rightarrow$ ocupa más espacio que JSON.
- Conviene aprovechar el barato almacenamiento para ***incrementar en velocidad*** de localización de información dentro de un documento.

JSON

El formato JSON está muy ampliamente extendido para intercambio de información a través de web, con documentos que constan de pares clave-valor de distintos tipos.


```
{
  "name": "Alderaan",
  "rotation_period": 24,
  "diameter": 12500,
  "climate": "temperate",
  "terrain": "grasslands, mountains",
  "population": 2000000000,
  "residents": [
    "https://swapi.dev/api/people/5/",
    "https://swapi.dev/api/people/68/"
  ],
  "films": [
    "https://swapi.dev/api/films/6/",
    "https://swapi.dev/api/films/1/"
  ],
  "created": "2014-12-10T11:35:48.479000Z",
  "edited": "2014-12-20T20:58:18.420000Z",
  "url": "https://swapi.dev/api/planets/2/"
}
```


Tipos de elementos JSON:

- Strings (cadenas de caracteres)
- Números (enteros o reales)
- Booleanos (`true` o `false`)
- Valor nulo (`null`)
- Arrays (listas ordenadas de elementos JSON)
- Objetos JSON (compuestos por sus propios conjuntos de pares clave-valor)

Tipos de valores en JSON

Lo anterior es un JSON `Object`. Pero no todo valor JSON es un objeto, veamos los posible tipos de valores[1]:

- JSON `Object`, caracterizado por sus llaves externas y las entradas para cada dato, separadas por comas. Estas entradas tienen una serie de claves cuyos valores pueden ser a su vez de cualquier tipo JSON
- JSON `Array`, conteniendo una serie de valores JSON del mismo tipo
- Cadenas (`String`)
- Valores numéricos (`Number`)
- Valores lógicos (`Bool`) → `true` o `false`
- Valor nulo (`Null`) → `null`

Puede encontrar todo tipo de ejemplos en Internet, tanto de archivos json[2] como de llamadas a API[3][4] que devuelven los datos en este formato.

[1] https://www.json.org/
[2] http://json-schema.org/learn/miscellaneous-examples.html
[3] https://jsonapi.org/examples/
[4] https://github.com/endpoints/endpoints-example

Descarga e instalación

- MongoDB es multiplataforma, luego la instalación dependerá de las características del S.O. destino. Optaremos por la versión *MongoDB community edition* 8.2.2 (*o la más reciente*), de 64 bits.[5]
- Podemos descargarla en este enlace. La instalación es distinta para cada sistema operativo:
  - Windows: seguir el asistente, quitar opción de instalar como servicio, añadir la carpeta `bin` del directorio donde se haya instalado al PATH del sistema operativo (SO) y crear la carpeta `C:/data/db`.
  - Mac Os X: Instalar MongoDB en Mac, Installing MongoDB and related GUIs
  - Linux: Instalación en Ubuntu 22.04, Compass en Linux

[5] Existen también versiones de pago (*Enterprise*) y en la nube (*Atlas*)

Clientes para MongoDB

- Además del servidor, debemos disponer de clientes para interactuar con él. Habremos instalado el cliente `MongoDBCompass`, basado en una interfaz gráfica.
- Nota: en versiones anteriores se instalaba por defecto el cliente `mongo`, para lanzar desde consola e interactuar mediante texto, pero ya no se instala en la versión actual. En su lugar, en versiones recientes de MongoDB se recomienda emplear `mongosh` (nuevo shell), que suele venir ya instalado e integrado en MongoDBCompass. En caso contrario, lo descargamos del enlace, lo descomprimimos en la carpeta que deseemos, y añadimos la ruta de su carpeta `bin` al PATH del SO.
- Recomendamos también instalar `NoSQL Booster`, más amigable que la consola, con editor de texto, gratuito y bastante interesante.
- Opcionalmente se puede emplear `Studio3T Free`.

MongoDB Command Line Database Tools

- Hay programas asociados a MongoDB para tareas como ***importar datos***, *crear copias de seguridad*, etc. que hasta ahora venían con MongoDB y ahora hay que instalar por separado. Son las `MongoDB Command Line Database Tools`.
- Podemos descargarlas en este enlace.
- Las instrucciones para su instalación se pueden consultar aquí.

Primeros pasos con servidor y cliente

- El servidor de base de datos de MongoDB (contiene el SGDB) se lanza como un servicio, o bien mediante el programa ejecutable `mongod`.
- Abrimos un terminal (e.g., símbolo de sistema en Windows), y tecleamos `mongod`, lanzando así el servidor (asumiendo que hemos añadido al PATH la carpeta bin –opción recomendada–, en caso contrario necesitamos navegar hasta la carpeta `bin` donde hayamos instalado `MongoDB`,).
- El proceso para lanzar el cliente es muy similar. En los clientes visuales tendremos ventanas para conectarnos al servidor. En caso de usar mongosh, abrimos otro terminal, sin cerrar la consola del servidor, y lanzamos el comando `mongosh`.
- En ese momento estamos listos para lanzar órdenes contra la base de datos.

Arrancando con la *shell* 8mongo, mongosh o similar

- Para ver las bases de datos disponibles en nuestro servidor mongod, tecleamos `show dbs`.

Arrancando con la *shell* 8mongo, mongosh o similar

- Para ver las bases de datos disponibles en nuestro servidor mongod, tecleamos `show dbs`.
- El comando `db` devuelve la bd actual.

Arrancando con la *shell* 8mongo, mongosh o similar

- Para ver las bases de datos disponibles en nuestro servidor mongod, tecleamos `show dbs`.
- El comando `db` devuelve la bd actual.
- Podemos seleccionar una escribiendo: `use mibd`. Si no existe, se crea automáticamente al usarla **e insertar algo**, sin necesitar declaración explícita de creación.

Arrancando con la *shell* 8mongo, mongosh o similar

- Para ver las bases de datos disponibles en nuestro servidor mongod, tecleamos `show dbs`.
- El comando `db` devuelve la bd actual.
- Podemos seleccionar una escribiendo: `use mibd`. Si no existe, se crea automáticamente al usarla **e insertar algo**, sin necesitar declaración explícita de creación.
- Se elimina una BD mediante `db.dropDatabase()`, y una colección individual mediante `db.micoleccion.drop()`.

Arrancando con la *shell* 8mongo, mongosh o similar

- Para ver las bases de datos disponibles en nuestro servidor mongod, tecleamos `show dbs`.
- El comando `db` devuelve la bd actual.
- Podemos seleccionar una escribiendo: `use mibd`. Si no existe, se crea automáticamente al usarla **e insertar algo**, sin necesitar declaración explícita de creación.
- Se elimina una BD mediante `db.dropDatabase()`, y una colección individual mediante `db.micoleccion.drop()`.
- Si necesitamos acudir a la ayuda del sistema para consultar alguna funcionalidad podemos hacerlo mediante `help`, o a la ayuda de una determinada función con `db.micoleccion.nombrefuncion.help()`.

Consultar la referencia del lenguaje para más detalle.

Índice

Colecciones

- Una colección en MongoDB sería el *equivalente* a una tabla en modelo relacional. Podemos ver las existentes mediante `show collections`.

Colecciones

- Una colección en MongoDB sería el *equivalente* a una tabla en modelo relacional. Podemos ver las existentes mediante `show collections`.
- Para crear una colección, ***no*** es necesario ***definir*** explícitamente el nombre de la colección y su esquema; se genera y actualiza conforme vayamos insertando **documentos**.

Colecciones

- Una colección en MongoDB sería el *equivalente* a una tabla en modelo relacional. Podemos ver las existentes mediante `show collections`.
- Para crear una colección, ***no*** es necesario ***definir*** explícitamente el nombre de la colección y su esquema; se genera y actualiza conforme vayamos insertando **documentos**.
- Podemos añadir mediante `db.cosas.insertOne({v: 27})` (para insertar varios, `db.cosas.insertMany([{v: 12}, {v: 33})]`).

Colecciones

- Una colección en MongoDB sería el *equivalente* a una tabla en modelo relacional. Podemos ver las existentes mediante `show collections`.
- Para crear una colección, ***no*** es necesario ***definir*** explícitamente el nombre de la colección y su esquema; se genera y actualiza conforme vayamos insertando **documentos**.
- Podemos añadir mediante `db.cosas.insertOne({v: 27})` (para insertar varios, `db.cosas.insertMany([{v: 12}, {v: 33})]`).
- También podríamos hacer asignar datos a una variable (mediante Javascript), y posteriormente insertarla. Por ejemplo:

```
j = {name: "mongo"}
k = {x: 3, n: 5}
db.cosas.insertMany([j,k])
```

Consultas

- Para consultar documentos, hacemos `db.cosas.find()`, devolviendo los documentos de la colección `cosas`.

Consultas

- Para consultar documentos, hacemos `db.cosas.find()`, devolviendo los documentos de la colección `cosas`.
- Lo anterior devolverá los 5 registros introducidos:

```
[
  { _id: ObjectId("63887712df2f29a880994e9d"), v: 27 },
  { _id: ObjectId("6388780adf2f29a880994e9e"), v: 12 },
  { _id: ObjectId("6388780adf2f29a880994e9f"), v: 33 },
  { _id: ObjectId("63887865df2f29a880994ea0"), name: 'mongo' },
  { _id: ObjectId("63887865df2f29a880994ea1"), x: 3, n: 5 }
]
```

Consultas

- Para consultar documentos, hacemos `db.cosas.find()`, devolviendo los documentos de la colección `cosas`.
- Lo anterior devolverá los 5 registros introducidos:

```
[
  { _id: ObjectId("63887712df2f29a880994e9d"), v: 27 },
  { _id: ObjectId("6388780adf2f29a880994e9e"), v: 12 },
  { _id: ObjectId("6388780adf2f29a880994e9f"), v: 33 },
  { _id: ObjectId("63887865df2f29a880994ea0"), name: 'mongo' },
  { _id: ObjectId("63887865df2f29a880994ea1"), x: 3, n: 5 }
]
```

- También podríamos consultar documentos por algún criterio, como por ejemplo haciendo: `db.cosas.find({name: "mongo"})`, que devolverá el documento cuyo name sea igual a “mongo”.

Consultas

- Para consultar documentos, hacemos `db.cosas.find()`, devolviendo los documentos de la colección cosas.
- Lo anterior devolverá los 5 registros introducidos:

```
[
  { _id: ObjectId("63887712df2f29a880994e9d"), v: 27 },
  { _id: ObjectId("6388780adf2f29a880994e9e"), v: 12 },
  { _id: ObjectId("6388780adf2f29a880994e9f"), v: 33 },
  { _id: ObjectId("63887865df2f29a880994ea0"), name: 'mongo' },
  { _id: ObjectId("63887865df2f29a880994ea1"), x: 3, n: 5 }
]
```

- También podríamos consultar documentos por algún criterio, como por ejemplo haciendo: `db.cosas.find({name: "mongo"})`, que devolverá el documento cuyo name sea igual a “mongo”.
- Para devolver un único documento: `db.cosas.findOne()`. Y para limitar el número: `db.cosas.find().limit(3)`.

Bucles e iteradores

- Podemos escribir bucles de tipo `for` o `while` desde el cliente de MongoDB. Por ejemplo, para añadir una serie de documentos en la colección cosas:

```
for (let i = 1; i <= 20; i++) db.cosas.insertOne({x: 4, j: i})
```

Bucles e iteradores

- Podemos escribir bucles de tipo `for` o `while` desde el cliente de MongoDB. Por ejemplo, para añadir una serie de documentos en la colección cosas:

```
for (let i = 1; i <= 20; i++) db.cosas.insertOne({x: 4, j: i})
```

- Ahora podemos consultar sus elementos: `db.cosas.find()`

  Esto mostrará un número máximo de documentos, por defecto 20; una vez mostrados los primeros podremos escribir `it` para ver más documentos.

Bucles e iteradores

- Podemos escribir bucles de tipo `for` o `while` desde el cliente de MongoDB. Por ejemplo, para añadir una serie de documentos en la colección cosas:

```
for (let i = 1; i <= 20; i++) db.cosas.insertOne({x: 4, j: i})
```

- Ahora podemos consultar sus elementos: `db.cosas.find()`

  Esto mostrará un número máximo de documentos, por defecto 20; una vez mostrados los primeros podremos escribir `it` para ver más documentos.

- Podemos declarar una variable con el cursor y recorrer el iterador:

```
let c = db.cosas.find()
while (c.hasNext()) printjson(c.next())
```

Bucles e iteradores

- Podemos escribir bucles de tipo `for` o `while` desde el cliente de MongoDB. Por ejemplo, para añadir una serie de documentos en la colección cosas:

```
for (let i = 1; i <= 20; i++) db.cosas.insertOne({x: 4, j: i})
```

- Ahora podemos consultar sus elementos: `db.cosas.find()`

  Esto mostrará un número máximo de documentos, por defecto 20; una vez mostrados los primeros podremos escribir `it` para ver más documentos.

- Podemos declarar una variable con el cursor y recorrer el iterador:

```
let c = db.cosas.find()
while (c.hasNext()) printjson(c.next())
```

- También podemos obtener un array a partir del cursor y luego acceder a una posición del mismo:

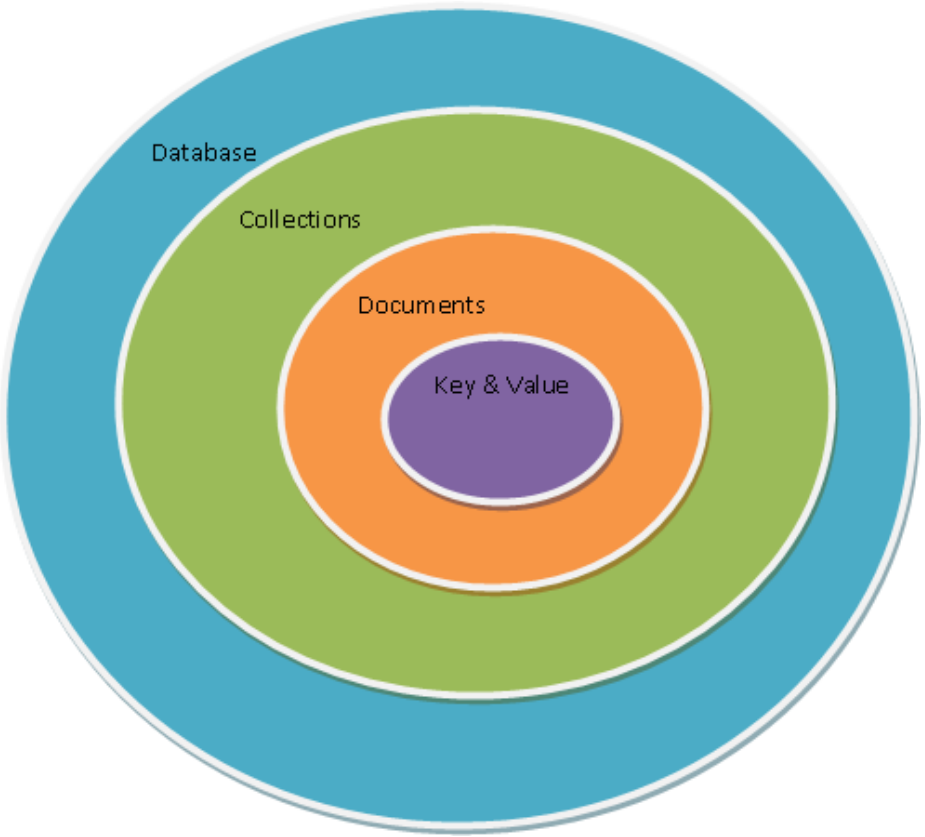
```
let c = db.cosas.find()
const v = c.toArray()
printjson(v[4])
```

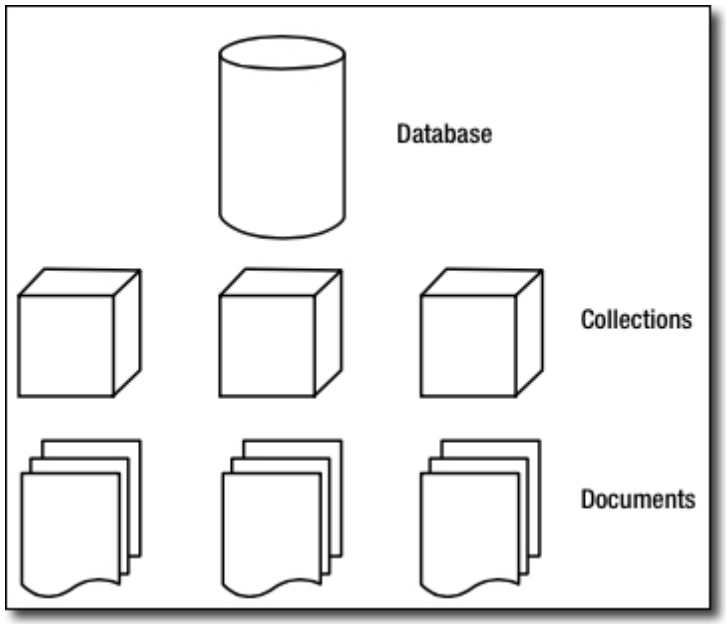
Próximos pasos

- Esta presentación proporciona únicamente una introducción a NoSQL y MongoDB.
- Encontraremos mucho más material en este curso completo. Revisaremos algunos aspectos de este curso (una versión anterior del mismo) en las próximas diapositivas.
- Prestaremos especial atención a distintos aspectos:
  - Uso básico del shell, importación/exportación de datos
  - Inserción de datos mediante `insert`, actualización mediante `update`.
  - Consulta de documentos mediante `find`.

NOTA (importante): *el contenido de las próximas diapositivas no es original, se toma del curso referenciado arriba; se han incluido aquí para que el alumnado tenga claro el material que entra en la asignatura*.

Estructuras en MongoDB: Vista general





Importación de datos

Podemos importar datos de archivos JSON, CSV o TSV.
Centrándonos en la opción de JSON, podemos distinguir dos casos:

- Archivo con un objeto JSON por línea:

```
mongoimport -d ejercicios -c ciudades --file mongo_cities1000.json
```

Importación de datos

Podemos importar datos de archivos JSON, CSV o TSV.
Centrándonos en la opción de JSON, podemos distinguir dos casos:

- Archivo con un objeto JSON por línea:

```
mongoimport -d ejercicios -c ciudades --file mongo_cities1000.json
```

- Archivo con un array de objetos JSON:

```
mongoimport -d ejercicios -c planetas --file planets.json --jsonArray
```

Importación de datos

Podemos importar datos de archivos JSON, CSV o TSV.
Centrándonos en la opción de JSON, podemos distinguir dos casos:

- Archivo con un objeto JSON por línea:

  ```
  mongoimport -d ejercicios -c ciudades --file mongo_cities1000.json
  ```

- Archivo con un array de objetos JSON:

  ```
  mongoimport -d ejercicios -c planetas --file planets.json --jsonArray
  ```

Si queremos que la importación elimine en su caso la colección preexistente, debemos añadir la partícula `--drop`.

Exportación de datos

Podemos exportar los datos a archivos JSON, de cualquiera de los tipos anteriores:

- Archivo con un objeto JSON por línea:

```
mongoexport -d ejercicios -c ciudades -o ciudades.json
```

Exportación de datos

Podemos exportar los datos a archivos JSON, de cualquiera de los tipos anteriores:

- Archivo con un objeto JSON por línea:

```
mongoexport -d ejercicios -c ciudades -o ciudades.json
```

- Archivo con un array de objetos JSON:

```
mongoexport -d ejercicios -c planetas -o planetas.json --jsonArray
```

Copia de seguridad (backup)

Podemos hacer una copia de seguridad de la base de datos completa, y restaurarla posteriormente en el mismo u otro equipo:

- Backup: crea copia de seguridad en subcarpeta `dump`

```
mongodump -d ejercicios
```

Copia de seguridad (backup)

Podemos hacer una copia de seguridad de la base de datos completa, y restaurarla posteriormente en el mismo u otro equipo:

- Backup: crea copia de seguridad en subcarpeta `dump`

  ```
  mongodump -d ejercicios
  ```

- Restore: restaura la base de datos con la información de la subcarpeta `dump`

  ```
  mongorestore --drop
  ```

Parámetros en consultas

Como patrón general, tenemos:
`db.micol.find(filtro,proyección)`, de modo que:

Parámetros en consultas

Como patrón general, tenemos:
`db.micol.find(filtro,proyección)`, de modo que:

- El `filtro` o `selección` debe ser un objeto JSON en el que se indiquen los criterios sobre los valores que deben tener los campos incluidos en el mismo.

Parámetros en consultas

Como patrón general, tenemos:
`db.micol.find(filtro,proyección)`, de modo que:

- El `filtro` o `selección` debe ser un objeto JSON en el que se indiquen los criterios sobre los valores que deben tener los campos incluidos en el mismo.
- La `proyección`, en caso de proporcionarse, debe ser otro objeto JSON en el que se indique qué campos se desea incluir (poniéndoles el valor `true` o 1) o excluir (asignándoles el valor `false` o 0).

Parámetros en consultas

Como patrón general, tenemos:
`db.micol.find(filtro,proyección)`, de modo que:

- El `filtro` o `selección` debe ser un objeto JSON en el que se indiquen los criterios sobre los valores que deben tener los campos incluidos en el mismo.
- La `proyección`, en caso de proporcionarse, debe ser otro objeto JSON en el que se indique qué campos se desea incluir (poniéndoles el valor `true` o 1) o excluir (asignándoles el valor `false` o 0).

Por ejemplo, podemos tener
`db.pobs.find({"prov": "Sevilla"},{"nomb": true, "_id": false})`.

Opciones de filtro

Formato:`{"clave1": criterio1, ..., "claveN": criterioN}`:

Opciones de filtro

Formato:`{"clave1": criterio1, ..., "claveN": criterioN}`:

- Las claves se corresponden con campos de la colección a filtrar.

Opciones de filtro

Formato:`{"clave1": criterio1, ..., "claveN": criterioN}`:

- Las claves se corresponden con campos de la colección a filtrar.
- Cada criterio puede ser:
  - Un valor concreto, que compararíamos por igualdad con el valor del campo en cada documento de la colección. Por ejemplo, `"Sevilla"` como criterio para la clave `"prov"`, quedando el par como `"prov": "Sevilla"`.

Opciones de filtro

Formato:`{"clave1": criterio1, ..., "claveN": criterioN}`:

- Las claves se corresponden con campos de la colección a filtrar.
- Cada criterio puede ser:
  - Un valor concreto, que compararíamos por igualdad con el valor del campo en cada documento de la colección. Por ejemplo, `"Sevilla"` como criterio para la clave `"prov"`, quedando el par como `"prov": "Sevilla"`.
  - Una condición, como por ejemplo `{"$gt": 5000}` para la clave `"pobl"` sería: `"pobl": {"$gt": 5000}`.
- El objeto de filtro con ambas condiciones quedaría como:
  `{"prov": "Sevilla", "pobl": {"$gt": 5000}}`

Opciones de filtro

Formato:`{"clave1": criterio1, ..., "claveN": criterioN}`:

- Las claves se corresponden con campos de la colección a filtrar.
- Cada criterio puede ser:
  - Un valor concreto, que compararíamos por igualdad con el valor del campo en cada documento de la colección. Por ejemplo, `"Sevilla"` como criterio para la clave `"prov"`, quedando el par como `"prov": "Sevilla"`.
  - Una condición, como por ejemplo `{"$gt": 5000}` para la clave `"pobl"` sería: `"pobl": {"$gt": 5000}`.
- El objeto de filtro con ambas condiciones quedaría como: `{"prov": "Sevilla", "pobl": {"$gt": 5000}}`

De forma similar a `"$gt"` (mayor que), tenemos `"$lt"` (menor que), y sus correspondientes `"$gte"` y `"$lte"` (que incluyen la igualdad).

Otros operadores

Al mismo nivel de los anteriores, disponemos de operadores como:

- $ne (no igualdad): `db.grades.find({type:{"$ne":"quiz"}})`

[6] Similar a RLIKE. Consulte esta chuleta sobre expresiones regulares

Otros operadores

Al mismo nivel de los anteriores, disponemos de operadores como:

- $ne (no igualdad): `db.grades.find({type:{"$ne":"quiz"}})`
- $exists (existencia): `db.grades.find({"score":{"$exists":true}})`

[6] Similar a RLIKE. Consulte esta chuleta sobre expresiones regulares

Otros operadores

Al mismo nivel de los anteriores, disponemos de operadores como:

- $ne (no igualdad): `db.grades.find({type:{"$ne":"quiz"}})`
- $exists (existencia):
  `db.grades.find({"score":{"$exists":true}})`
- $not (negación):
  `db.grades.find({score:{"$not": {"$mod": [5,0]}}})`

[6] Similar a RLIKE. Consulte esta chuleta sobre expresiones regulares

Otros operadores

Al mismo nivel de los anteriores, disponemos de operadores como:

- $ne (no igualdad): `db.grades.find({type:{"$ne":"quiz"}})`
- $exists (existencia):
  `db.grades.find({"score":{"$exists":true}})`
- $not (negación):
  `db.grades.find({score:{"$not": {"$mod": [5,0]}}})`
- $regex (expresión regular[6]):
  `db.people.find({nombre: {"$regex":/aitor/i}})`

[6] Similar a RLIKE. Consulte esta chuleta sobre expresiones regulares

Condiciones adicionales (I)

Campos anidados:

```
db.catalogo.find({"precio":{$gt:10000},
                  "reviews.calificacion":{$gte:5}})
```

Condiciones adicionales (I)

Campos anidados:

```
db.catalogo.find({"precio":{$gt:10000},
                  "reviews.calificacion":{$gte:5}})
```

Condiciones compuestas:
- Tipo OR:

```
db.grades.find({ $or:[ {"type":"exam"}, {"score":{$gte:65}} ]})
db.grades.find({ $or:[ {"score":{$lt:50}}, {"score":{$gt:90}} ]})
```

Condiciones adicionales (I)

Campos anidados:

```
db.catalogo.find({"precio":{$gt:10000},
                  "reviews.calificacion":{$gte:5}})
```

Condiciones compuestas:
- Tipo OR:

```
db.grades.find({ $or:[ {"type":"exam"}, {"score":{$gte:65}} ]})
db.grades.find({ $or:[ {"score":{$lt:50}}, {"score":{$gt:90}} ]})
```

- Tipo AND:

```
db.grades.find({ $and:[ {type:"exam"}, {score:{$gte:65}} ] })
```

Condiciones adicionales (I)

Campos anidados:

```
db.catalogo.find({"precio":{$gt:10000},
                  "reviews.calificacion":{$gte:5}})
```

Condiciones compuestas:
- Tipo OR:

```
db.grades.find({ $or:[ {"type":"exam"}, {"score":{$gte:65}} ]})
db.grades.find({ $or:[ {"score":{$lt:50}}, {"score":{$gt:90}} ]})
```

- Tipo AND:

```
db.grades.find({ $and:[ {type:"exam"}, {score:{$gte:65}} ] })
```

Nótese que esto último es equivalente a

```
db.grades.find({ type:"exam", score:{$gte:65} })
```

Condiciones adicionales (II)

- NOR ($nor, negación de OR):

```
db.grades.find({ score:{$gte:65},
                 $nor:[ {type:"quiz"}, {type:"homework"} ] })
```

Condiciones adicionales (II)

- NOR ($nor, negación de OR):

```
db.grades.find({ score:{$gte:65},
                 $nor:[ {type:"quiz"}, {type:"homework"} ] })
```

- IN ($in, coincidencia con algún valor del array dado):

```
db.grades.find({ type:{$in:["quiz","exam"]} })
```

Condiciones adicionales (II)

- NOR ($nor, negación de OR):

```
db.grades.find({ score:{$gte:65},
                 $nor:[ {type:"quiz"}, {type:"homework"} ] })
```

- IN ($in, coincidencia con algún valor del array dado):

```
db.grades.find({ type:{$in:["quiz","exam"]} })
```

- NIN ($nin, negación de IN).

```
db.grades.find({score:{$gte:65}, type:{$nin:["quiz","homework"]} })
```

Consultas sobre arrays

- ALL ($all, devuelve los documentos que contengan todos los valores del array dado en el array indicado como clave):

```
db.people.find({ amistades: {$all: ["Juan", "David"]},
                 hobbies: {$in: ["footing", "baloncesto"]} })
```

Consultas sobre arrays

- ALL ($all, devuelve los documentos que contengan todos los valores del array dado en el array indicado como clave):

```
db.people.find({ amistades: {$all: ["Juan", "David"]},
                 hobbies: {$in: ["footing", "baloncesto"]} })
```

- $elemMatch, devuelve documentos que contengan en el array indicado un objeto que cumpla todas las condiciones impuestas:

```
db.scores.find({
  results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } })
```

Consultas sobre arrays

- ALL ($all, devuelve los documentos que contengan todos los valores del array dado en el array indicado como clave):

```
db.people.find({ amistades: {$all: ["Juan", "David"]},
                 hobbies: {$in: ["footing", "baloncesto"]} })
```

- $elemMatch, devuelve documentos que contengan en el array indicado un objeto que cumpla todas las condiciones impuestas:

```
db.scores.find({
  results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } })
```

- $size, filtra por tamaño del array:

```
db.people.find( {hobbies : {$size : 3}} )
```

Consultas sobre arrays

- ALL ($all, devuelve los documentos que contengan todos los valores del array dado en el array indicado como clave):

```
db.people.find({ amistades: {$all: ["Juan", "David"]},
                 hobbies: {$in: ["footing", "baloncesto"]} })
```

- $elemMatch, devuelve documentos que contengan en el array indicado un objeto que cumpla todas las condiciones impuestas:

```
db.scores.find({
  results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } })
```

- $size, filtra por tamaño del array:

```
db.people.find( {hobbies : {$size : 3}} )
```

- $slice, usado en la proyección, permite restringir el número de elementos del array que mostraremos

```
db.people.find( {hijos: {$gt:1}}, {hobbies: {$slice:2}} )
```

Distinct

Devuelve los distintos valores del campo indicado, aplicando el filtro establecido. Como patrón general, tenemos: `db.micol.distinct(campo,filtro)`.

Distinct

Devuelve los distintos valores del campo indicado, aplicando el filtro establecido. Como patrón general, tenemos: `db.micol.distinct(campo,filtro)`.

```
> db.grades.distinct('type')
---> [ "exam", "quiz", "homework" ]

> db.grades.distinct('type', { score: { $gt: 99.9 } } )
---> [ "exam" ]
```

CountDocuments (o count sobre un cursor)

Cuenta el número de documentos que cumplen el filtro indicado.
Vemos que admite distintos tipos de uso:

CountDocuments (o count sobre un cursor)

Cuenta el número de documentos que cumplen el filtro indicado.
Vemos que admite distintos tipos de uso:

```
db.grades.countDocuments({type:"exam"})

db.grades.find({type:"exam"}).count()

db.grades.countDocuments({type:"essay", score:{$gt:90}})
```

Update

Permite actualizar uno o muchos documentos. Como patrón general, tenemos: `db.micol.update(filtro,actualización,opciones)`, siendo el último parámetro opcional (no lo emplearemos así nosotros).

```
db.people.update({nombre:"Steve Jobs"},
                 {$set: {nombre:"Domingo Gallardo", salario: 1000000}})
db.people.update({salario: {$ge: 80000}},
                 {$set: {sueldo:"Alto"}},
                 {multi: true})
```

Update

Permite actualizar uno o muchos documentos. Como patrón general, tenemos: `db.micol.update(filtro,actualización,opciones)`, siendo el último parámetro opcional (no lo emplearemos así nosotros).

```
db.people.update({nombre:"Steve Jobs"},
                 {$set: {nombre:"Domingo Gallardo", salario: 1000000}})
db.people.update({salario: {$ge: 80000}},
                 {$set: {sueldo:"Alto"}},
                 {multi: true})
```

Como observamos, el segundo ejemplo tiene una opción para que se haga actualización múltiple (**ojo**: no usaremos ese parámetro). Hoy día la función `update` está *deprecated* (obsoleta), por lo que se recomienda usar en su lugar las funciones `updateOne` y `updateMany`. Lo mismo ocurre con la función `insert` y las correspondientes funciones `insertOne` e `insertMany`.

Operadores de actualización

Tenemos muchos tipos de actualizaciones a realizar[7]. Algunos habituales son:

- $set, modifica solo lo indicado, no sustituyendo todo el documento:

```
db.people.update({nombre:"Aitor Medrano"},{ $set:{salario: 1000000} })
```

[7] Nótese que en los ejemplos vereis `update`, pero lo reemplazamos con `updateOne` o `updateMany` según el caso

[8] Más información en este enlace

Operadores de actualización

Tenemos muchos tipos de actualizaciones a realizar[7]. Algunos habituales son:

- $set, modifica solo lo indicado, no sustituyendo todo el documento:

```
db.people.update({nombre:"Aitor Medrano"},{ $set:{salario: 1000000} })
```

- $inc, incrementa el valor de la variable en la cantidad indicada:

```
db.people.update({nombre:"Aitor Medrano"},{ $inc:{salario: 1000} })
```

[7] Nótese que en los ejemplos vereis `update`, pero lo reemplazamos con `updateOne` o `updateMany` según el caso

[8] Más información en este enlace

Operadores de actualización

Tenemos muchos tipos de actualizaciones a realizar[7]. Algunos habituales son:

- $set, modifica solo lo indicado, no sustituyendo todo el documento:

```
db.people.update({nombre:"Aitor Medrano"},{ $set:{salario: 1000000} })
```

- $inc, incrementa el valor de la variable en la cantidad indicada:

```
db.people.update({nombre:"Aitor Medrano"},{ $inc:{salario: 1000} })
```

- $unset, elimina el campo indicado:

```
db.people.update({nombre:"Aitor Medrano"},{ $unset:{twitter: ''} })
```

[7] Nótese que en los ejemplos vereis `update`, pero lo reemplazamos con `updateOne` o `updateMany` según el caso

[8] Más información en este enlace

Operadores de actualización

Tenemos muchos tipos de actualizaciones a realizar[7]. Algunos habituales son:

- $set, modifica solo lo indicado, no sustituyendo todo el documento:

```
db.people.update({nombre:"Aitor Medrano"},{ $set:{salario: 1000000} })
```

- $inc, incrementa el valor de la variable en la cantidad indicada:

```
db.people.update({nombre:"Aitor Medrano"},{ $inc:{salario: 1000} })
```

- $unset, elimina el campo indicado:

```
db.people.update({nombre:"Aitor Medrano"},{ $unset:{twitter: ''} })
```

Otros operadores: $mul, $min, $max, $currentDate, etc.[8]

[7] Nótese que en los ejemplos vereis `update`, pero lo reemplazamos con `updateOne` o `updateMany` según el caso

[8] Más información en este enlace

Otras actualizaciones

- Renombrado de campos (por cada par de elementos pasados al `rename`, el primer campo es el nombre antiguo y el segundo es el nuevo):

```
db.people.update( { _id: 1 },
  { $rename: { 'nickname': 'alias', 'cell': 'movil' } } )
```

Otras actualizaciones

- Renombrado de campos (por cada par de elementos pasados al `rename`, el primer campo es el nombre antiguo y el segundo es el nuevo):

```
db.people.update( { _id: 1 },
  { $rename: { 'nickname': 'alias', 'cell': 'movil' } } )
```

- Método `findAndModify`, encuentra, modifica y devuelve el objeto (lo indicamos con el campo `new` a `true`):

Otras actualizaciones

- Renombrado de campos (por cada par de elementos pasados al `rename`, el primer campo es el nombre antiguo y el segundo es el nuevo):

```
db.people.update( { _id: 1 },
  { $rename: { 'nickname': 'alias', 'cell': 'movil' } } )
```

- Método `findAndModify`, encuentra, modifica y devuelve el objeto (lo indicamos con el campo `new` a `true`):

```
db.grades.findAndModify({
  query: { student_id: 0, type: "exam"},
  update: { $inc: { score: 1 } },
  new: true
})
```

Otras actualizaciones

- Renombrado de campos (por cada par de elementos pasados al `rename`, el primer campo es el nombre antiguo y el segundo es el nuevo):

```
db.people.update( { _id: 1 },
  { $rename: { 'nickname': 'alias', 'cell': 'movil' } } )
```

- Método `findAndModify`, encuentra, modifica y devuelve el objeto (lo indicamos con el campo `new` a `true`):

```
db.grades.findAndModify({
  query: { student_id: 0, type: "exam"},
  update: { $inc: { score: 1 } },
  new: true
})
```

Nota: usaremos el `rename` durante el curso, pero no el `findAndModify`.

Actualización sobre arrays

Añadir elementos

Push: añadir a lista

- `$push` simple:

```
db.people.update({nombre: "Luis"}, {$push: {aficiones: "Cine"}})
```

Actualización sobre arrays

Añadir elementos

Push: añadir a lista

- $push simple:

```
db.people.update({nombre: "Luis"}, {$push: {aficiones: "Cine"}})
```

- $push múltiple (nótese el empleo de $each):

```
db.people.update(
  {nombre: "Luis2"},
  {$push: {aficiones: {$each : ["Música", "Juegos de mesa"]}}})
```

Actualización sobre arrays

Añadir elementos

Push: añadir a lista

- $push simple:

```
db.people.update({nombre: "Luis"}, {$push: {aficiones: "Cine"}})
```

- $push múltiple (nótese el empleo de $each):

```
db.people.update(
  {nombre: "Luis2"},
  {$push: {aficiones: {$each : ["Música", "Juegos de mesa"]}}})
```

AddToSet: añadir a conjunto

```
db.enlaces.update({titulo:"google"}, {$addToSet: {tags:"buscador"} } )
db.enlaces.update(
  {titulo:"google"},
  {$addToSet: {tags: { $each:["drive", "traductor"]}}})
```

Actualización sobre arrays

Eliminar elementos

Pull - `$pull` elimina todas las ocurrencias en el array del elemento indicado, `$pullAll` elimina todas las ocurrencias de todos los elementos indicados:

```
db.enlaces.update({titulo:"google"}, {$pull: {tags:"traductor"}})
db.enlaces.update({titulo:"google"},
                  {$pullAll: {tags:["calendario","email"]}})
```

Actualización sobre arrays

Eliminar elementos

Pull - `$pull` elimina todas las ocurrencias en el array del elemento indicado, `$pullAll` elimina todas las ocurrencias de todos los elementos indicados:

```
db.enlaces.update({titulo:"google"}, {$pull: {tags:"traductor"}})
db.enlaces.update({titulo:"google"},
                  {$pullAll: {tags:["calendario","email"]}})
```

Pop: elimina elementos por el principio (-1) o por el final (1):

```
db.enlaces.update({titulo:"google"},{$pop: {tags:-1}})
```

Actualización sobre arrays

Operación posicional

Modifica el elemento de una posición del array (el referenciado por el operador $)

- Ejemplo: dada una colección que incluye el elemento `{ "_id" : 1, "grades" : [ 80, 85, 90 ] }`, y ante la necesidad de actualizar el 80 por un 82:

```
db.students.update({ _id: 1, grades: 80}, {$set: {"grades.$": 82}})
```

Actualización sobre arrays

Operación posicional

Modifica el elemento de una posición del array (el referenciado por el operador $)

- Ejemplo: dada una colección que incluye el elemento `{ "_id" : 1, "grades" : [ 80, 85, 90 ] }`, y ante la necesidad de actualizar el 80 por un 82:

```
db.students.update({ _id: 1, grades: 80}, {$set: {"grades.$": 82}})
```

- Ejemplo: dada la colección siguiente

```
{"_id" : 4, "grades" :
  [{grade: 80, mean: 75, std: 8},
   {grade: 85, mean: 90, std: 5},
   {grade: 90, mean: 85, std: 3}]}
```

y la necesidad de cambiar a 6 el campo `std` de las notas con `grade` a 85:

```
db.students.update({_id: 4, "grades.grade": 85 },
                   {$set: {"grades.$.std": 6}})
```

Delete

Permite eliminar un documento con `deleteOne`, o muchos con `deleteMany`. Como patrón general, tenemos: `db.micol.deleteOne(filtro)` (o Many, en su caso).

```
db.people.deleteOne({nombre:"Domingo Gallardo"})
db.people.deleteMany({age:{$lt: 18}})
```

Índice

Pipeline de Agregaciones. Idea general

- **Alternativa a** realizar una serie de **llamadas sucesivas a distintas funciones**
- Además, permite hacer **cosas que no podemos hacer** mediante las funciones **anteriores**
- Se estructuran bien las operaciones mediante la función `db.micol.aggregate`, que **recibe un array** de operaciones.
- Se incluyen operaciones de agrupación, filtrado, cálculo, proyección, etc.
- Información detallada en `https://aitor-medrano.github.io/iabd/sa/agregaciones.html#pipeline-de-agregacion`

Pipeline de Agregaciones. Formato general

Formato: `db.micol.aggregate([op1,...,opN])`
Estructura de cada operación:

- Cada operación es un **objeto JSON**: (`{$tipoOp: valor}`)
- El `valor` puede ser un valor simple como cadena, número, etc., o de nuevo un objeto con la configuración de la operación.

Pipeline de Agregaciones. Operación `$match`

Filtra documentos según condiciones, equivalente a WHERE en SQL.

```
db.usuarios.aggregate([
  { $match: { edad: { $gte: 18 } } }
])
```

Pipeline de Agregaciones. Operación $group

Agrupa documentos según un campo o conjunto de campos y permite aplicar acumuladores.

```
db.ventas.aggregate([
  { $group: { _id: "$categoria", total: { $sum: "$importe"
])
```

Pipeline de Agregaciones. Operación $project

Selecciona o crea campos específicos.

```
db.usuarios.aggregate([
  { $project: {
      nombreCompleto: { $concat: ["$nombre", " ", "$apellic
      edad: 1 }
  }
])
```

Pipeline de Agregaciones. Operación $sort

Ordena el resultado.

```
db.productos.aggregate([
  { $sort: { precio: -1 } }   // Descendente
])
```

Pipeline de Agregaciones. Operación `$limit`

Limita el número de resultados.

```
db.logs.aggregate([
  { $limit: 10 }
])
```

Pipeline de Agregaciones. Operación `$skip`

Omite los primeros N resultados. Útil para paginación.

```
db.logs.aggregate([
  { $skip: 20 }
])
```

Pipeline de Agregaciones. Operación $unwind

Desestructura un array en documentos individuales.

```
db.pedidos.aggregate([
  { $unwind: "$items" }
])
```

Pipeline de Agregaciones. Operación $lookup

Une colecciones (similar a JOIN). **OJO**: vamos a tratar de evitar al máximo este tipo de operaciones.

```
db.pedidos.aggregate([
  { $lookup: {
      from: "clientes",
      localField: "clienteId",
      foreignField: "_id",
      as: "cliente" }
  }
])
```

Pipeline de Agregaciones. Operación $addFields

Añade nuevos campos al documento.

```
db.usuarios.aggregate([
  { $addFields: { esAdulto: { $gte: ["$edad", 18] }} }
])
```

Pipeline de Agregaciones. Operación $set

Alias moderno de $addFields.

```
db.productos.aggregate([
  { $set: { stockBajo: { $lt: ["$stock", 10] } } }
])
```

Pipeline de Agregaciones. Operación $count

Cuenta los documentos resultantes del pipeline.

```
db.usuarios.aggregate([
  { $match: { activo: true } },
  { $count: "numUsuariosActivos" }
])
```

Índice

Principios de diseño en NoSQL

Acerca del diseño de BBDD no relacionales, cabe mencionar que:[9]:

- No hay una base tan sólida y sistemática como en BBDD relacionales.
- Hay una gran heterogeneidad en tipologías de BBDD no relacionales.
- Cada tipo de BD no relacional puede seguir distintas directrices.
- Debemos *desaprender* algunos conceptos que nos han llevado a separar relaciones y atomizar la información.
- Nos centraremos en el diseño de BBDD de tipo *documentos*, como MongoDB.

[9] Tomado de `https://aitor-medrano.github.io/iabd/sa/modelado.html`

Diseño de una BD de tipo Documentos como MongoDB

Sobre el diseño en MongoDB:

- Debemos olvidarnos de tener una gran cantidad de relaciones normalizadas enlazadas entre sí.
- No existen claves ajenas ni joins, aunque hay cierta operación similar (minoritaria).
- Debemos cambiar de enfoque.
- Si ciertos bloques de información se van a consultar frecuentemente en conjunto, deberían formar parte del mismo documento.
- Para ello, deberemos *embeber* unos documentos en otros.
- Cuando queramos separar entre colecciones, deberemos establecer referencias (manuales o de tipo DBRef)

Referencias

Referencias manuales

```
var idUsuario = ObjectId();

db.usuario.insert({
  _id: idUsuario,
  nombre: "123xyz"
});

db.contacto.insert({
  usuario_id: idUsuario,
  telefono:  "123 456 7890",
  email: "xyz@ejemplo.com"
});
```

Referencias

Referencias mediante DBRef

```
var idUsuario = ObjectId();

db.usuario.insert({
  _id: idUsuario,
  nombre: "123xyz"
});

db.contacto.insert({
  usuario_id: new DBRef("usuario", idUsuario),
  telefono:  "123-456-7890",
  email: "xyz@example.com"
});
```

Datos embebidos

- El verdadero cambio de enfoque surge al embeber los datos mediante subdocumentos y arrays dentro de los valores de un documento
- Esto permite obtener todos los datos con un único acceso a un documento, y sin *joins*.
- Se emplean generalmente para capturar:
  - Relaciones 1:1 o 1:N, con un N razonablemente pequeño ("uno a pocos").
  - Relaciones 1:N con un N grande ("uno a muchos") pero consultado con mucha frecuencia desde el contexto de la parte 1
  - Relaciones N:M, vistas frecuentemente desde una de las dos partes (*one-way embedding*) o desde las dos (*two-way embedding*)

Datos embebidos. Ejemplos

Relaciones 1:1

```
{
  nombre: "Aitor",
  edad: 38,
  direccion: {
    calle: "Mayor",
    ciudad: "Elx"
  }
}
```

- Ventaja: con una sola consulta tenemos todo, pero bien estructurado.
- ¿Cuándo evitar? Si se consulta muy rara vez la parte embebida y es muy grande, o si casi siempre modificamos una pequeña parte y queremos evitar trabajar con un documento mucho más pesado.

Datos embebidos. Ejemplos

Relaciones 1:N de 1 a pocos

```
{
  titulo: "La broma asesina",
  url: "http://es.wikipedia.org/wiki/Batman:_The_Killing_Joke",
  texto: "La dualidad de Batman y Joker",
  comentarios: [
    {
      autor: "Bruce Wayne",
      fecha: ISODate("2015-04-01T09:31:32Z"),
      comentario: "A mi me encantó"
    },
    {
      autor: "Bruno Díaz",
      fecha: ISODate("2015-04-03T10:07:28Z"),
      comentario: "El mejor"
    }
  ]
}
```

Datos embebidos. Ejemplos

Relaciones 1:N de 1 (Editorial) a muchos (Libro)

```
{
  _id: 1,
  nombre: "O'Reilly",
  pais: "EE.UU."
}
{
  _id: 1234,
  titulo: "MongoDB: The Definitive Guide",
  autor: [ "Kristina Chodorow", "Mike Dirolf" ],
  numPaginas: 216,
  editorial_id: 1,
}
{
  _id: 1235,
  titulo: "50 Tips and Tricks for MongoDB Developer",
  autor: "Kristina Chodorow",
  numPaginas: 68,
  editorial_id: 1,
}
```

- ¿Cuándo embeber? Cuando queremos consultar conjuntamente la información de ambas entidades, o bien cuando la información de interés para la parte 1 se ciñe a un momento concreto de la evolución de la parte N, que puede cambiar posteriormente en la misma y queremos preservar inalterada en la parte 1.
- ¿Cuándo referenciar? Cuando la N es demasiado grande (limitación de BSON), cuando los documentos incluyen demasiada información, o cuando se consultan poco en conjunto ambas entidades.

Relaciones N:M

- También pueden ser pocos a pocos o bien muchos a muchos.
- Podemos emplear tres enfoques:
  - Relacional, con una colección por cada entidad original y un documento en una nueva colección, que referencia a ambas entidades vinculadas.
  - Un documento embebido en otro (*one-way embedding*), todo en una única colección, siempre que no crezca demasiado el documento y no produzca inconsistencias notables por anomalías de modificación frecuentes. Si hay órdenes de magnitud muy diferentes, embeber la entidad con menos ocurrencias dentro de la más poblada.
  - Dos documentos (*two-way embedding*), cada uno con los ids del otro documento, o con algún dato más de la otra entidad.

Relaciones N:M. Enfoque relacional

```
{
  autor_id: 1,
  libro_id: 1
}
```

En general, debemos evitar este enfoque, poco usual en BBDD no relacionales.

Relaciones N:M. One-way embedding

```
{
  _id: 1,
  titulo: "La historia interminable",
  anyo: 1979,
  autores: [{nombre:"Michael Ende", pais:"Alemania"}]
}
{
  _id: 2,
  titulo: "Momo",
  anyo: 1973,
  autores: [{nombre:"Michael Ende", pais:"Alemania"}]
}
```

Buen enfoque si solemos ver la entidad desde la perspectiva de la parte 1.

Relaciones N:M. Two-way embedding

Libro referencia autor:

```
{
  _id: 1,
  titulo: "La historia interminable",
  anyo: 1979,
  autores: [1]
}
{
  _id: 2,
  titulo: "Momo",
  anyo: 1973,
  autores: [1]
}
```

Y autor referencia a libro:

```
{
  _id: 1,
  nombre: "Michael Ende",
  pais: "Alemania",
  libros: [1,2]
}
```

Otros casos especiales

- Jerarquías: podemos tener todos los documentos en una sola colección, con más o menos campos según el hijo del que se trate cada documento.
- Relaciones reflexivas: referenciamos al documento o documentos que corresponda en la propia colección.