



Bases de Datos No SQL

Tema 3. Bases de Datos Orientadas a Documentos. **MongoDB**



Universidad
Rey Juan Carlos

Bibliografía

- Dan Sullivan (2015). **NoSQL for Mere Mortals.** Addison-Wesley Professional
- Dan McCreary & Ann Kelly (2014). **Making Sense of NoSQL. A guide for managers and the rest of us.** Manning Publications
- Gaurav Vaish (2013). **Getting Started with NoSQL. Your guide to the world and technology of NoSQL.** PACKT Publishing
- Joe Celko (2014). **Joe Celko's Complete guide to NoSQL. What every SQL professional needs to know about nonrelational databases.** Morgan Kauffmann

Bibliografía

- MongoDB. **The MongoDB 4.2 Manual**
<https://docs.mongodb.org/manual/>
- Boug Bierer (2018). **MongodDB 4 Quick Start**. Ed.: Packt Publishing
- Cyrus Dasadia y Amol Nayak (2016). **MongoDB Cookbook**. 2ª Edición Ed.: Packt Publishing
- David Hows, Peter Membrey, Eelco Plugge, Tim Hawkins (2015). **The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB**, 3ª Edición. Ed.: Apress
- Alex Giamas (2017). **Mastering MongoDB 3.x**. Ed.: Packt Publishing

Índice

- **Introducción a BD orientadas a Documentos**
- Introducción a MongoDB
- Configuración del Entorno de Trabajo
- Operaciones CRUD con MongoDB
- Creación y Uso de Índices
- Aggregation Framework

Bases de Datos Orientadas a Documentos

- Las Bases de Datos Orientadas a Documentos o Documentales son el tipo de BD NoSQL más populares.
- Productos de BD orientadas a documentos: **MongoDB**, Couchbase y CouchDB.
- Utilizan una **aproximación Clave-Valor con diferencias importantes**:
 - Almacena los **valores** como **documentos** (entidades semiestructuradas de datos que se almacenan como strings o representaciones binarias de strings).
 - **Documentos** típicamente en un formato estándar como **JSON** (JavaScript Object Notation) o **XML** (Extensible Markup Language).
 - **Documentos** almacenan tanto **estructura** como **contenido**.
 - En un **único documento** se almacenan todos los **atributos de una entidad** (en vez de almacenar cada atributo de una entidad con una clave separada).
 - Ejemplo: documento JSON:

```
{
  nombre: "Luis",
  apellido: "García",
  cargo: "Gerente",
  despacho: "2-120",
  teléfono: "555-222-3456",
}
```

Bases de Datos Orientadas a Documentos

- Las BD orientadas a Documentos **no necesitan definir un esquema predefinido** de añadir datos:
 - Cuando añadimos un documento se crea la estructura de datos subyacente necesaria.
- La “falta de esquema” da a los desarrolladores **más flexibilidad** que con las BD relacionales.
- Ejemplo:

```
{
  nombre: "Luis",
  apellido: "García",
  cargo: "Gerente",
  despacho: "2-120",
  teléfono: "555-222-3456",
}
```

```
{
  nombre: "Roberto",
  apellido: "Rodríguez",
  cargo: "Asesor",
  despacho: "2-130",
  teléfono: "555-222-3478",
  f.alta: "1-Feb-2010",
  f.baja: "12-Ago-2014"
}
```

- No hay problema en que **f.alta** y **f.baja** no estén en el documento de “Luis”.
- Los desarrolladores pueden añadir atributos si lo necesitan, **pero**: sus programas son **responsables de gestionarlo** bien.

Bases de Datos Orientadas a Documentos

- Un documento **puede incluir a otro documento (documentos embebidos)**, listas de documentos/valores...
 - Esto **elimina la necesidad de unir/combinar (*join*)** documentos como hacemos con las tablas.

- Ejemplo utilizando JavaScript Object Notation (JSON):

```
{  
  "id_cliente":187693,  
  "nombre": "Ana Pérez",  
  "dirección" :  
    {  
      "calle" : "C/ Mayor, 12",  
      "ciudad" : "Móstoles",  
      "provincia" : "Madrid",  
      "CP" : "28933"  
    },  
  "primer_pedido" : "15/01/2013",  
  "ultimo_pedido" : "27/06/2014"  
}
```

Documento
Embebido

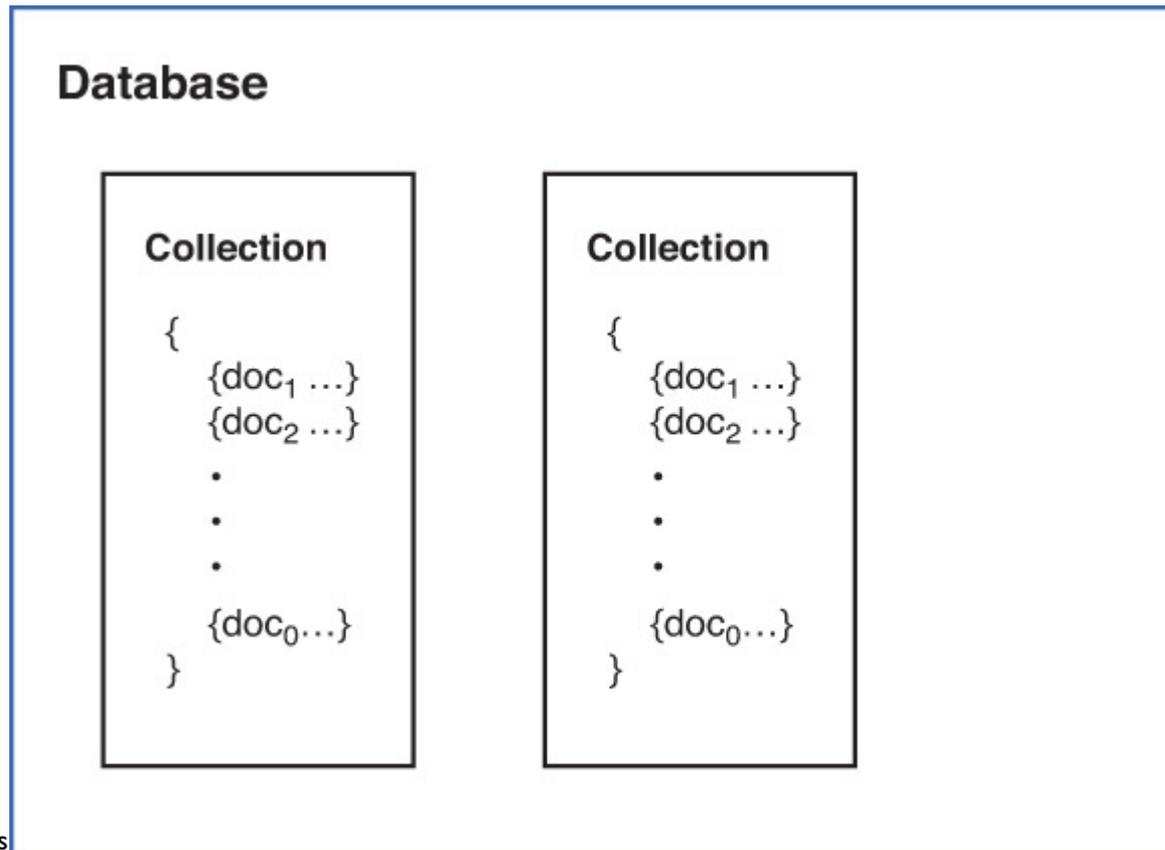
Bases de Datos Orientadas a Documentos.

En resumen:

- Combinan la **flexibilidad** de las Bases de Datos NoSQL con la posibilidad de **gestionar datos complejos**.
- Los documentos contienen tanto información de la **estructura** como **contenido (datos)**.
 - Un **documento es un conjunto de pares clave-valor**.
 - Las claves en un par nombre-valor indican un **atributo** y se representan como cadenas de caracteres.
 - Los valores en un par nombre-valor es el **dato asignado** al atributo y pueden ser tipos de datos básicos (números, cadenas o booleanos) o estructuras (arrays u objetos).
 - JSON y XML son dos formatos utilizados habitualmente para definir documentos.
- Existe gran **libertad en la estructura** de los documentos.
 - **No** precisan definir la **estructura** de los datos a priori.
 - Son **flexibles** en cuanto a los atributos utilizados.
- Permiten **consultar y filtrar colecciones** de documentos:
 - Permiten consultas con **múltiples atributos**.

Bases de Datos Orientadas a Documentos.

En una **BD orientada a Documentos** los documentos “similares” se agrupan en colecciones. Una BD puede tener un **conjunto de colecciones**, que están compuestas a su vez por un **conjunto de documentos**.



Bases de Datos Orientadas a Documentos. **Diseño.**

En el **diseño** de una Base de Datos orientada a Documentos hay que tener en cuenta los siguientes aspectos:

- qué colecciones de documentos
- qué esquema, cuánta desnormalización
- cómo modelar las relaciones
- qué y cuántos índices utilizar

¿Cuándo hay que usar **documentos embebidos** y cuándo **referencias** a otros documentos?

➤ Los **principios de diseño en Bases de Datos NoSQL** hay que aplicarlos con **flexibilidad**.

Siempre se deben considerar los beneficios y desventajas de un principio de diseño en una **situación particular**.

Bases de Datos Orientadas a Documentos. **Colecciones.**

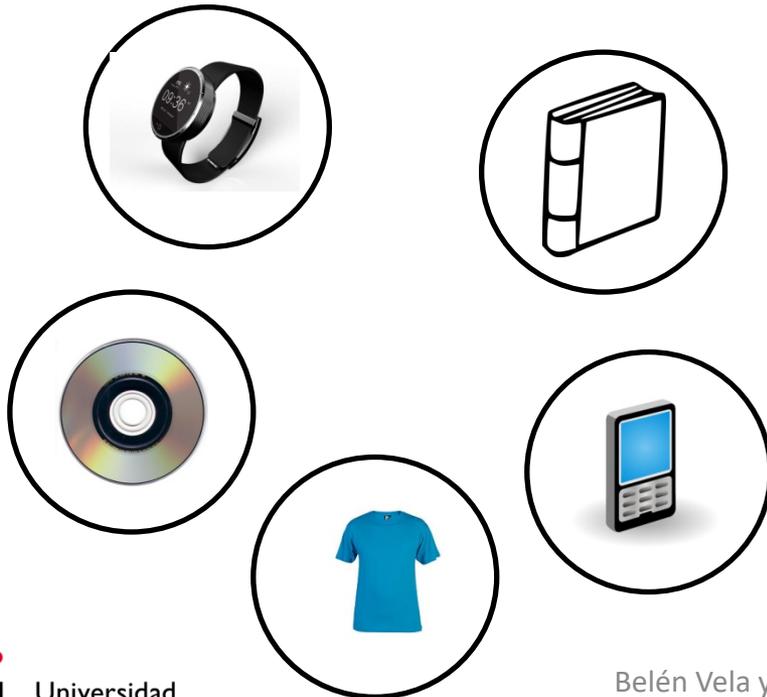
- Gestión de múltiples documentos en **colecciones**:
 - Los documentos **se agrupan en colecciones** de documentos “similares”.
 - Los documentos en una colección no tienen por qué tener estructuras idénticas, pero deberían **compartir alguna estructura común**.
 - Un aspecto clave es decidir **cómo organizar los documentos en colecciones** (dado que no es obligatorio que estén relacionados).



Bases de Datos Orientadas a Documentos. Colecciones.

- ¿Cómo diseñar la BD?

Entidades separadas para cada producto



Una entidad para todos los productos indicando el tipo de producto



Bases de Datos Orientadas a Documentos. Colecciones.

- ¿Cómo decidir cómo organizar los datos en **una o más colecciones** de documentos?
Podemos empezar con: **¿Cómo se utilizan los datos? ¿Cuáles son las consultas más frecuentes?**
 - ¿Cuál es el número medio de **productos** comprados por cliente?
 - ¿Cuál es el rango (mínimo, máximo) de **productos adquiridos** por cliente?
 - ¿Cuáles son los 20 **productos** más populares por cliente?
 - ¿Cuál es el valor medio de las ventas por **producto** (precio estándar por cliente-coste del producto)?
 - ¿Cuántos **productos de cada tipo** se vendieron en los últimos 30 días?
- Todas **las consultas** utilizan datos de todos los tipos de productos, y únicamente la última necesita subtotales por tipo de producto. Este es un buen indicador de que los productos deberían estar en una **colección de documentos única**.

Bases de Datos Orientadas a Documentos. Colección

- ¿Cómo diseñar la BD?

Otro motivo en favor de una **colección de documentos única** es que **probablemente haya nuevos tipos de productos**. Si crece hasta decenas o centenares de tipos, el **número de colecciones** será poco manejable.

Entidades Separadas



Entidad única para tipos de producto



En las **BD NoSQL** en vez de empezar por los datos, tratando de organizarlos, **podemos empezar por las consultas (aplicaciones)** para entender **cómo se usarán** los datos.

Bases de Datos Orientadas a Documentos. Esquema.

Schemaless:

- Cuando trabajamos con BD relacionales definimos un esquema (una especificación que describe la estructura de un objeto, en este caso una tabla).

```
CREATE TABLE customer (  
  customer_ID integer,  
  name varchar(100),  
  street varchar(100),  
  city varchar(100),  
  state varchar(2),  
  zip varchar(5),  
  first_purchase_date date,  
  last_purchase_date date);
```

⇒ En este esquema **todos** los clientes tienen los mismos datos.

- En una **BD Relacional** se **diseñan las tablas antes de que los desarrolladores** del software puedan interactuar con las tablas de la BD.
- En las **BD orientadas a Documentos** no se requiere este paso, ya que los **desarrolladores pueden crear colecciones y documentos insertándolas directamente en la BD** (sin especificación previa del esquema).

Bases de Datos Orientadas a Documentos. Esquema.

Schemaless:

- No existe la necesidad de elaborar un modelo que contenga todos los posibles campos.
 - **Schemaless**
- Sin embargo, existe una **organización implícita** en los documentos que se insertan. Esta organización está explícita en el código que manipula los documentos.
- Los **documentos** (a diferencia de las tuplas) son **polimórficos: no es obligatorio que tengan la misma estructura de datos**. Podemos añadir un dato diferente a un documento, sin que el resto de documentos tengan que tener dicho dato.
- Las Bases de Datos orientadas a Documentos tienen un **esquema polimórfico**, es decir, que pueden tener al mismo tiempo **“varios esquemas”**.

Bases de Datos Orientadas a Documentos. Documentos Embebidos.

Documentos embebidos (embedding):

- Esta decisión implica la *desnormalización* de los datos, almacenando dos documentos relacionados en un **único documento** (en MongoDB mismo `_id`).
- **¿Cuándo? En general:**
 - Relaciones del **tipo agregación** (“o contiene”) entre entidades.
 - Relaciones **Uno a Uno**.
 - Relaciones **1:N (Uno a Pocos)** donde la parte de muchos siempre aparecen o se consultan desde el contexto del documento padre o de nivel superior.
- ⇒ Las operaciones a este documento son **mucho menos costosas** para el servidor que las operaciones que involucran múltiples documentos.
- ⇒ **BENEFICIO:** Recuperación más rápida de información.
- ⇒ **PERO:** No podemos acceder al documento embebido si no es a través del documento padre.

Bases de Datos Orientadas a Documentos. **Modelado de Relaciones.**

- **Modelado de relaciones** habituales en BD orientadas a Documentos
 - Uno a muchos
 - Muchos a muchos
 - Jerarquías

- **Embeber documentos vs referencias**
 - Tamaño de N
 - Acceso a los datos

Bases de Datos Orientadas a Documentos. Índices.

- Al diseñar una BD hay que intentar **identificar el número ideal de índices**:
 - Excesivos \Rightarrow reducirá el rendimiento de las escrituras
 - Pocos \Rightarrow reducirán el rendimiento de lecturas
- Aspectos a tener en cuenta:
 - Ratio lectura/escritura (BD para análisis...)
 - Campos que se utilizarán en las búsquedas
 - BD con **fuerte carga de lecturas** suelen tener muchos índices, especialmente si no hay patrones claros de consultas.
 - BD con **fuerte carga de escrituras** (p.e. sensor): minimizar número de índices (identificadores), equilibrio
 - Si aún así es importante la latencia de consultas, puede plantearse la creación de una segunda BD con datos agregados para lecturas (DW)

Índice

- *Introducción a BD orientadas a Documentos*
- **Introducción a MongoDB**
- Configuración del Entorno de Trabajo
- Operaciones CRUD con MongoDB
- Creación y Uso de Índices
- Aggregation Framework

Introducción a MongoDB

- **¿Qué es MongoDB?**

- Mongo proviene de *humongous* (enorme)
- Base de Datos **NoSQL** (Almacenamiento de Datos No-Relacionales) **orientada a documentos**.

- **Características**

- BD NoSQL más popular del mercado (<http://db-engines.com/en/ranking>)
- Almacenamiento de documentos JSON (almacenados en BSON)
 - Estructuras más cercanas a los programas

```
{ nombre : "Ana", edad : "30", ciudad:"Móstoles", provincia : "Madrid", telf : ["916678989","678998877"] }
```

- Implementación **fácil**
- **Rápida** (“*eliminación de necesidad de joins*”)
- **Robusta y escalable**
- Esquemas dinámicos / **flexibles** (*schemaless*): dos documentos no tienen porqué tener el mismo esquema. *No* es necesario definirlo *antes* de añadir datos.
 - Al añadir un documento se crea la estructura de datos subyacente necesaria

Introducción a MongoDB:

Documentos JSON

- Los documentos contienen tanto información de la estructura como datos (*nombre – valor*):
 - El **nombre** en un par nombre-valor indica un **atributo**.
 - El **valor** en un par nombre-valor es el **dato asignado al atributo**.
- **Ejemplo:** Dos documentos de la colección Personas.

Documento JSON 1:

```
{ "Nombre": "Pedro",  
  "Apellidos": "Martínez Campo",  
  "Edad": 22,  
  "Aficiones": ["fútbol", "tenis", "ciclismo"],  
  "Amigos": [  
    { "Nombre": "María", "Edad": 22 },  
    { "Nombre": "Luis", "Edad": 28 } ] }
```

- ⇒ Es un documento JSON clásico: Tiene strings, 2 arrays (Aficiones y Amigos, que a su vez contiene 2 documentos embebidos) y números.

Documento JSON 2:

```
{ "Nombre": "Luis",  
  "Apellidos": "Jiménez Pérez",  
  "Estudios": "Administración y Dirección de Empresas",  
  "Edad": 52,  
  "Amigos": 12 }
```

- ⇒ Este documento no sigue el mismo esquema que el primero. Tiene menos campos, algún campo nuevo que no existe en el documento anterior e incluso un campo de distinto tipo.

Esto no es posible en una base de datos relacional, aunque es algo totalmente válido en MongoDB.

Índice

- *Introducción a BD orientadas a Documentos*
- *Introducción a MongoDB*
- **Configuración del Entorno de Trabajo**
- Operaciones CRUD con MongoDB
- Creación y Uso de Índices
- Aggregation Framework

Configuración del entorno de trabajo

- Instalación de **MongoDB Community Edition 4.2.3** (64 bits) (open source)
 - Cambiar PATH
 - Crear **directorio datos** en el mismo disco que se ha instalado MongoDB (\data\db)
 - mongod (servidor)
 - mongo (shell)

<https://www.mongodb.com/download-center/community>

<https://docs.mongodb.com/manual/administration/install-community/>

- Instalación de herramienta gráfica **MongoDB Compass Community Edition** (opción de la instalación de MongoDB)
- Instalación de Editor de MongoDB: **Robo 3T** versión 1.3. (antes Robomongo)

<https://robomongo.org/download>

Configuración del entorno de trabajo

- Arrancar el servidor

- > `cd C:\Program Files\MongoDB\Server\4.2.3\bin`

- > `mongod.exe`

- MongoDB usará “C:\data\db” como directorio de datos
→ Si no existe da error

- > `mongod.exe --dbpath “C:\datosmongo\db”`

- Si queremos cambiar el path de datos

Por defecto: **puerto 27017.**

Configuración del entorno de trabajo

- La Consola o Shell de MongoDB
 - > `cd C:\Program Files\MongoDB\Server\ 4.2.3\bin`
 - > `mongo.exe`
 - Por defecto, nos conectamos a la BD **prueba** o **test**.
- Para cambiar de BD o crear una nueva:
 - use BDNueva
 - Identificador tiene longitud máxima de 64 caracteres*
- Para saber a qué BD se está conectado:
 - db
- Para listar las BD existentes
 - show databases
- Para borrar una BD (a la que estemos conectados)
 - db.dropDatabase()
- Para ver la versión de MongoDB que tenemos instalada
 - db.version()

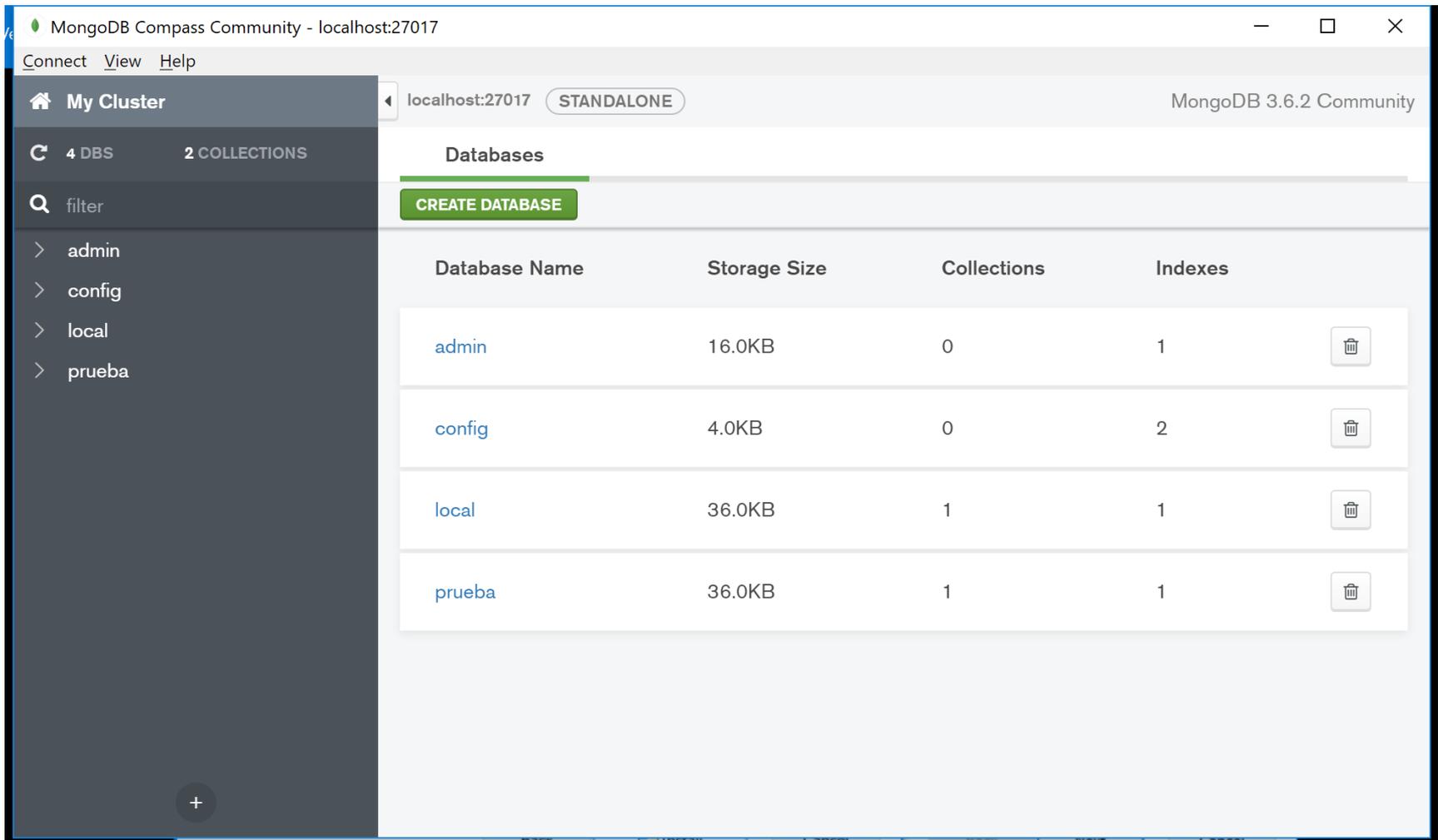
Configuración del entorno de trabajo

- **MongoDB Compass Community Edition**

Permite:

- Visualizar, añadir y borrar BD y colecciones
- Visualizar y interactuar con documentos (CRUD)
- Construir y ejecutar consultas
- Visualizar y optimizar rendimiento de consultas (explain plans visuales)
- Gestión de índices

Configuración del entorno de trabajo



The screenshot shows the MongoDB Compass Community interface for a local instance (localhost:27017). The left sidebar displays a tree view of databases: admin, config, local, and prueba. The main area shows a table of databases with columns for Database Name, Storage Size, Collections, and Indexes. A 'CREATE DATABASE' button is visible above the table.

Database Name	Storage Size	Collections	Indexes
admin	16.0KB	0	1
config	4.0KB	0	2
local	36.0KB	1	1
prueba	36.0KB	1	1

Índice

- *Introducción a BD orientadas a Documentos*
- *Introducción a MongoDB*
- *Configuración del Entorno de Trabajo*
- **Operaciones CRUD con MongoDB**
- Creación y Uso de Índices
- Aggregation Framework

Operaciones CRUD con MongoDB

- Añadir un nuevo documento a una colección
- Modificar documentos existentes
- Borrar documentos de una colección
- Recuperación de documentos

Operaciones CRUD con MongoDB

Antes de comenzar:

- Nombres de la BD y las colecciones son *case sensitive*.
- Tamaño máximo del nombre de la BD 64 caracteres.
- Nombres de las colecciones **no** pueden contener:
 - \$
 - “ ”
 - Comenzar con espacio en blanco o con *system*.
- **Recomendaciones:**
 - Nombre de las BD con apéndice DB o BD.
 - Nombre de colecciones no deben ser excesivamente largos.
 - Nombre de colecciones mejor en minúsculas y plural.
 - No usar separadores en nombre de colecciones (_).

Operaciones CRUD con MongoDB:

Añadir Documentos a una Colección

- **Inserción de un documento:**

db.NombreColeccion.insert({"clave":"valor"})

⇒ Si la colección NombreColeccion no existe, se creará.

- **Inserción de un array de documentos:**

*db.NombreColeccion.insert([
{documento1},...,{documentoN}])*

⇒ Añade una clave “_id” a cada documento (si no se añade explícitamente), que será un identificador de objeto (ObjectId, 12 bytes)

⇒ Si se inserta un array de documentos, estos se insertan por defecto en **orden**. Si se opta por una inserción sin orden (ordered = FALSE), si ocurre un error, se continúan insertando el resto de documentos.

⇒ Tamaño máximo de un documento BSON: **16 MB**

⇒ Función insert no es compatible con el operador

db.collection.explain()

Operaciones CRUD con MongoDB

Añadir Documentos a una Colección

- Inserción no ordenada
 - Si ocurre un error, **continúa** con la inserción del resto de documentos.

Ya existe en la BD.
ERROR: Pero continua insertando siguiente

```
db.alumnos.insert(  
  [{"_id":11,"nombre":"Jorge","apellido":"Ramon"},  
  {"_id":13,"nombre":"Maira","apellido":"Alonso"}],  
  {ordered: false})
```

Operaciones CRUD con MongoDB: Añadir Documentos a una Colección

- **Inserción de UN documento en una colección:**

`db.NombreColección.insertOne({"clave":"valor"})`

Operador disponible a partir de versión 3.2

- ⇒ Añade una clave “_id” al documento (si no se añade explícitamente), que será un identificador de objeto.
- ⇒ Devuelve el _id insertado en la colección.
- ⇒ Con insertOne() no se puede utilizar el operador `db.collection.explain()`

Operaciones CRUD con MongoDB:

Añadir Documentos a una Colección

- **Inserción de múltiples documentos en una colección:**

db.NombreColección.insertMany([doc1,...,doc2])

Operador disponible a partir de versión 3.2

- ⇒ Añade una clave “_id” a cada documento (si no se añade explícitamente), que será un identificador de objeto.
- ⇒ Devuelve la lista de _id insertados en la colección.
- ⇒ Permite la inserción ordenada y no ordenada:
ordered:boolean (por defecto ordered=TRUE)
- ⇒ Con insertMany() no se puede utilizar el operador db.NombreColección.explain().

Operaciones CRUD con MongoDB.

Añadir Documentos a una Colección

- Insertar documentos con `insert()` o `save()`
db.Nombre_Colección.insert()
 - Como parámetro pasaremos un documento (el objeto JSON) o un array de documentos que queremos insertar.
 - Añade una clave “_id” a cada documento (si no se lo pasamos explícitamente)

db.Nombre_Colección.save ()

- Como parámetro pasaremos el documento o array de documentos JSON que queremos insertar
- Puede funcionar como un **insert** (si no existe el _id en la colección) o como un **update**

Operaciones CRUD con MongoDB.

Añadir Documentos a una Colección

- Diferencia entre SAVE e INSERT
 - **Save:** Si se proporciona un identificador `_id` se hace una **actualización (sustitución) del documento** (invoca el método `upsert()`), si no se inserta (invoca el método `insert()`).
 - **Insert:** Se inserta siempre un nuevo documento. Si el identificador `_id` ya existe, da error de identificador duplicado.

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Para recuperar el **nombre** de todas las **colecciones** de una base de datos:

```
db.getCollectionNames()
```

```
show Collections
```
- Para **recuperar todos los documentos** de una colección:

```
db.NombreColeccion.find()
```

```
db.NombreColeccion.find().pretty()
```
- Para **renombrar** una **colección**:

```
db.NombreColeccion.renameCollection(NNombre)
```

→ La colección NNombre no debe existir en la BD

Operaciones CRUD con MongoDB.

Recuperación de Documentos

Ejemplo sencillo con `db.colección.find().pretty()`

```
>>
> db.clientes.insert(<<_id:"222", "nombre":"Alvaro", "edad":65>>)
WriteResult<< "nInserted" : 1 >>
> db.clientes.find<>.pretty<>
<
  "_id" : ObjectId<"56ccd5027cf92176cbd58f4b">,
  "nombre" : "Juan",
  "apellido" : "Perez"
>
< "_id" : "111", "apellido" : "Ramirez" >
<
  "_id" : ObjectId<"56ccd56b7cf92176cbd58f4c">,
  "nombre" : "Elvira",
  "apellido" : "Martin"
>
< "_id" : "222", "nombre" : "Alvaro", "edad" : 65 >
>>
```

Operaciones CRUD con MongoDB

Añadir Documentos a una Colección

Ejercicio:

Cree una nueva Base de Datos **BDClientes** y añada la colección **Clientes** en MongoDB e inserte en la misma la siguiente información:

id_cliente:187693 (_id)

nombre: Ana Pérez

dirección :

calle: C/ Mayor, 12

ciudad: Móstoles

provincia: Madrid

CP : 28933

teléfonos: 914556677,677445566

primer pedido : 15/01/2013

último pedido: 27/06/2014

nombre: Juan Martínez

dirección:

ciudad: Madrid

provincia: Madrid

profesión: Profesor

Recupere el contenido de la colección.

Renombre la colección a “clientes”.



Operaciones CRUD con MongoDB:

Modificar Documentos existentes

- Método `update` (query, update operator, *options*)
Modifica un documento o documentos existentes en una colección:

- Parámetros:
 - Consulta (*qué documento*)
 - Operación (conjunto de claves y valores a actualizar, es decir, *cuáles son los valores nuevos*)
 - Opciones
- Por defecto, se actualiza un único documento.

```
db.alumnos.update (  
  {"_id": 11},  
  {$set : {"titulación" : "GII" }})
```

¿Qué documentos?

¿Cómo los modifico?

Operador de UPDATE

⇒ Si el campo “titulación” no existe en el documento, éste se añade, en caso contrario, se actualiza su valor.

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

Comparison Query Operators (*para especificar qué documentos*)

Name	Description
\$eq	Matches values that are equal to a specified value.
\$gt	Matches values that are greater than a specified value.
\$gte	Matches values that are greater than or equal to a specified value.
\$lt	Matches values that are less than a specified value.
\$lte	Matches values that are less than or equal to a specified value.
\$ne	Matches all values that are not equal to a specified value.
\$in	Matches any of the values specified in an array.
\$nin	Matches none of the values specified in an array.

Sintaxis: { atributo: { **\$operador**: valor } }

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

Update Operators (*qué operación de actualización*)

Fields Name	Description
\$set	Sets the value of a field in a document.
\$unset	Removes the specified field from a document.
\$setOnInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
\$inc	Increments the value of the field by the specified amount.
\$mul	Multiplies the value of the field by the specified amount.
\$rename	Renames a field.
\$min	Only updates the field if the specified value is less than the existing field value.
\$max	Only updates the field if the specified value is greater than the existing field value.
\$currentDate	Sets the value of a field to current date, either as a Date or a Timestamp.

Sintaxis: { **\$operador**: { atributo: valor } }

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

- Actualización de documento o Inserción si el documento no existe en la colección: **update**
 - Es necesario poner la opción **upsert** al valor **true** (por defecto, FALSE). De esta forma se actualizará, si existe, el documento y en caso contrario, se insertará.

```
db.alumnos.update(  
  { "Edad": { $gte: 18 } },  
  { $set: { "mayorEdad": true }, $inc: { "Edad": 1 } },  
  { upsert: true }  
)
```

Se pueden incluir todos los elementos a modificar.
\$inc: si no existe el campo se incluye con el valor 1

Si no existe ningún alumno con una edad ≥ 18 inserta un nuevo documento.

Sólo lo actualiza para el primer documento que encuentra:
multi: false (valor por defecto)

- Actualización o Inserción db.colección.save()

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

- Actualización de **múltiples documentos con update()**
 - Es necesario poner la opción **multi** al valor **true**. De esta forma se actualizarán todos los documentos que satisfagan la condición.

```
db.alumnos.update(  
  { "Edad": { $gte: 18 } },  
  { $set: { "mayorEdad": true }, $inc: { "Edad": 1 } },  
  { multi: true }  
)
```

Operador de Comparación

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

- Método `updateOne` (query, update operator, *options*)
Modifica el primer documento existente en una colección que satisfaga la condición :

- Parámetros:

- Consulta (*qué documento*) o {} si se quiere modificar **el primer documento**
- Operación (conjunto de claves y valores a actualizar, es decir, *cuáles son los valores nuevos*)
- Opciones

```
db.alumnos.updateOne (  
  {"_id": 11},  
  {$set : {"titulación" : "GIS", "escuela" : "ETSII" }})
```

Operador de UPDATE

⇒ Como el campo “titulación” no existe en el documento, éste se añade.

Operaciones CRUD con MongoDB:

Modificar Documentos existentes

- Método `updateMany` (query, update operator, *options*)

Modifica todos los documentos de una colección que satisfagan la condición :

- Parámetros:
 - Consulta (*qué documentos*) o {} si se quiere modificar **todos los documentos**
 - Operación (conjunto de claves y valores a actualizar, es decir, *cuáles son los valores nuevos*)
 - Opciones

```
db.alumnos.updateMany (  
  {"_id": 566},  
  {$set : {"campus" : "Móstoles" }})
```

Operador de UPDATE

Operaciones CRUD con MongoDB:

Borrar Documentos de una Colección

- Método `deleteOne` (query operator, *options*)

Borra el primer documento existente en una colección que satisfaga la condición :

- Parámetros:
 - Consulta (*qué documento*) o {} si se quiere borrar **el primer documento**
 - Opciones

```
db.alumnos.deleteOne (  
  {"_id": 11})
```

⇒ Se borra el (primer) documento de la colección alumnos con identificador 11.

Operaciones CRUD con MongoDB:

Borrar Documentos de una colección

- Método `deleteMany` (query operator, *options*)

Borra todos los documentos de una colección que satisfagan la condición:

- Parámetros:
 - Consulta (*qué documentos*) o {} si se quiere modificar **todos los documentos**
 - Opciones

```
db.alumnos.deleteMany (  
  {"campus": "Móstoles"})
```

Operaciones CRUD con MongoDB:

Borrar Documentos de una Colección

- `db.colección.remove({"clave":"valor"})`
 - ⇒ Borra los documentos de la colección, donde la clave "clave" sea igual al "valor".
 - ⇒ Si se especifica la opción `{justOne:"True"}` solo se borra el primer elemento.
- `db.colección.remove({})`
 - ⇒ Borra todos los documentos de una colección
- `db.colección.drop ()`
 - ⇒ Borra la colección de la BD (**sin parámetros**)
 - ⇒ **Mejor rendimiento drop** que **remove** pero no permite criterio de borrado

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Recuperar todos los documentos de una colección
- Especificar una condición de igualdad
- Especificar condiciones usando operadores de consulta
- Especificar condiciones AND
- Especificar condiciones OR
- Especificar condiciones AND y OR
- Documentos embebidos
- ARRAYS

Operaciones CRUD con MongoDB:

Recuperación de Documentos

- **Recuperar todos los documentos de una colección**

```
db.alumnos.find()
```

- **Especificar una condición de igualdad (ambas son equivalentes)**

⇒ Documentos cuyo alumno se llame “Juanito”

```
db.alumnos.find({"author": "Juanito"})
```

```
db.alumnos.find({"autor": {$eq: "Juanito"}})
```

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Especificar una condición LIKE (contiene una cadena de caracteres)

```
db.alumnos.find({"nombre": /A./})
```

```
db.clientes.find({"nombre": /.u./})
```
- Especificar condiciones usando operadores de consulta

```
db.alumnos.find({"Edad": {$gte:18}})
```

Operaciones CRUD con MongoDB:

Recuperación de Documentos

- **Para indicar qué campos queremos recuperar**

```
db.clientes.find({"nombre":{"$eq:"Alfredo"}},{_id:1, ciudad:1})
```

- Solo mostrará, de los clientes que se llamen Alfredo, su identificador y el atributo ciudad (1)
- Aunque no se incluya el `_id`, este siempre aparece.

- **Exclusión de un atributo del resultado de la consulta**

```
db.clientes.find({"nombre":{"$eq:"Alfredo"}},{ciudad:0})
```

- Se mostrará para los clientes que se llamen Alfredo todos los atributos menos la ciudad (0)

```
db.clientes.find({},{_id:0, nombre:1})
```

- Se mostrará para todos los clientes su nombre (1) pero no el atributo `_id` (0)

```
db.clientes.find({},{_id:0})
```

- Se mostrará para los clientes todos los atributos meno el atributo `_id` (0)

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Especificar condiciones AND

Sintaxis: `{ $and: [{ <expresión1> }, { <expresión2> }, ... , { <expresiónN> }] }`

\$and realiza una operación AND sobre un array de dos o más expresiones y recupera los documentos que satisfagan **todas** las expresiones.

```
db.alumnos.find ({$and: [{"Edad":{$gte:18}}, {"nombre" : "Juanito"}]})
```

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- **Especificar condiciones OR**

Sintaxis: { \$or: [{ <expresión1> }, { <expresión2> }, ... , { <expresiónN> }] }

- **\$or** realiza una operación OR sobre un array de dos o más expresiones y recupera los documentos que satisfagan **alguna** de las expresiones

```
db.alumnos.find ({$or: [{"Edad":{$gte:18}}, {"nombre": "Belen"}]})
```

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- **Especificar condiciones AND y OR:**

Se pueden combinar los operadores AND y OR:

```
db.alumnos.find( {  
  $and : [  
    { $or : [ { "nombre" : "Juanito" }, { "nombre" : "Belen" } ] },  
    { $or : [ { "Edad" : 18 }, { "Edad": { $gt : 18 } } ] }  
  ]  
})
```

Operaciones CRUD con MongoDB

- Ejemplo colección Clientes con documento embebido y array:

```
{
  "id_cliente":187693,
  "nombre": "Ana Pérez",
  "dirección" : {
    "calle" : "C/ Mayor, 12",
    "ciudad" : "Móstoles",
    "provincia" : "Madrid",
    "CP" : "28933"
  },
  "telefonos":["914556677","677445566"],
  "primer_pedido" : "15/01/2013",
  "ultimo_pedido" : "27/06/2014"
}
```

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- **Documentos embebidos**
- `db.clientes.find({"dirección": {"calle": "C/Mayor, 12", "ciudad": "Móstoles", "provincia": "Madrid", "CP": "28933"} }).pretty()`

Devuelve el documento de la colección clientes que **tenga exactamente** el documento embebido indicado (en el mismo orden y todos los elementos)

- `db.clientes.find({"dirección.provincia": "Madrid"})`

Devuelve los documentos de la colección clientes que tengan el par provincia:Madrid en el documento embebido "dirección".

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- **ARRAYs**

```
db.clientes.find({"telefonos":["914556677","677445566"]})
```

Devuelve el documento de la colección *clientes* que tenga exactamente el array especificado (exactamente los mismos elementos).

```
db.clientes.find({"telefonos":"914556677"})
```

Devuelve los documentos de la colección *clientes* que contengan en el array *telefonos* el elemento especificado.

Operaciones CRUD con MongoDB.

Recuperación de Documentos

Ordenación de Documentos

- Por defecto las consultas devuelven los datos en el mismo orden en que fueron insertados en la base de datos.
- Si se desea ordenar el resultado debe usarse el método `sort()`
`db.coleccion.sort({campo1:tipoordenacion,campo2:tipoordenacion,...,campoN:tipoordenacion});`
- **Tipo de ordenación:**
 - Ascendente: 1
 - Descendente: -1

```
db.clientes.find({"nombre":"Mario"}).sort({"_id":1,"edad":1})
```

Devuelve los documentos de la colección *clientes* que cuyo nombre sea “Mario” ordenado por el identificador de objeto y la edad de forma ascendente.

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Método `count()`

Este método devuelve el número de documentos de una colección que satisfacen una condición.

Sintaxis: `db.nombreColección.count(condición, opciones)`

- `db.clientes.count()` **equivalente** `db.clientes.find().count()`
- `db.clientes.find({"nombre":"Eva"}).count()`
- `db.clientes.count({"nombre":"Eva"})`
- `db.clientes.count({"nombre":{"$eq:"Eva"}})`

- Método `limit()`

Este método limita el número de documentos de una colección que devuelve una consulta.

- `db.clientes.find().limit(2)`

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Operador **\$exists**

Sintaxis: {campo: {\$exists:<boolean>}}

Este operador devuelve todos los documentos que **tienen o no un determinado campo**.

Ejemplo:

```
db.alumnos.find({"titulación": {$exists:true}})
```

```
db.alumnos.find({"edad": {$exists:false}})
```

Operaciones CRUD con MongoDB.

Recuperación de Documentos

- Operador **\$type**

Sintaxis: {campo: {\$type:<tipo>}}

Este operador devuelve los documentos **cuyo campo sea de un determinado tipo.**

Ejemplo:

```
db.alumnos.find ({"edad": {$type: "int"}})
```

```
db.pacientes.find({"dirección": {$type:"string"}})
```

Operaciones CRUD con MongoDB.

Ejercicio 1

- Uso básico de operaciones CRUD



Índice

- *Introducción a MongoDB*
- *Configuración del Entorno de Trabajo*
- *Operaciones CRUD con MongoDB*
- **Creación y Uso de Índices**
- Aggregation Framework

Creación y Uso de Índices

- Consultar una colección con millones de documentos **sin tener índices** sobre uno o varios campos es **impensable en muchos casos**.
 - Gran diferencia entre realizar una consulta sobre campo con índice, y realizarla sin él.
- ⇒ **Importante:** Creación, configuración y uso de índices en nuestras colecciones.

Creación y Uso de Índices

- **Creación de un índice mediante:**

db.coleccion.createIndex(claves, opciones)

⇒ ALIAS (hasta versión 3.0.0): `db.collection.ensureIndex()`

Crea el índice si no existe previamente. Si ya existe, **no** se sobrescribe.

`_id`: **índice único** creado por defecto

Ejemplo: `db.clientes.createIndex({"Edad":1}) => name: Edad_1`

`db.clientes.createIndex({"nombre":1,apellido:1},{name:" nombre y apellidos"})`

- **Borrado de un índice existente (por nombre o especificación del índice):**

db.coleccion.dropIndex("nombreIndice") o

db.coleccion.dropIndex("especificación del campo": -1)

El índice creado para `_id` no se puede borrar.

- **Borrado de todos los índices de una colección:**

db.coleccion.dropIndexes()

Creación y Uso de Índices

- Para ver los índices definidos sobre una colección (obtener su nombre):

```
db.pacientes.getIndexes()
```

```
> db.pacientes.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "prueba.pacientes"
  },
  {
    "v" : 1,
    "key" : {
      "nombre" : 1
    },
    "name" : "nombre_1",
    "ns" : "prueba.pacientes"
  }
]
```

Versión de índices (0 ó 1 o 2)->

Base de datos y colección ->

- Campos indexados

- Nombre del índice

Creación y Uso de Índices

Tipos de Índices

- **Existen diferentes tipos de índices en MongoDB**
 - A la hora de configurar un índice, tenemos varias opciones donde elegir:

Creación y Uso de Índices

Tipos de Índices

Índices simples o de un solo campo:

- Estos índices se aplican a **un solo campo de los documentos** de una colección.
- Para declarar un índice de este tipo debemos usar la siguiente sentencia:

```
db.pacientes.createIndex( { "nombre" : 1 } )  
db.clientes.createIndex( { "nombre" : -1 } )
```

- El número 1 indica que queremos que el índice se ordene de forma ascendente. Si quisiéramos un orden descendente, el parámetro será un -1.
- Se pueden definir tanto sobre los documentos de primer nivel como sobre campos de **documentos embebidos**.

```
db.clientes.createIndex( { "dirección.provincia" : 1 } )
```

Creación y Uso de Índices

Tipos de Índices

Índices compuestos:

- Estos índices se aplican a **varios campos de los documentos** de una colección.
- Para declarar un índice de este tipo debemos usar un comando similar a este:

```
db.pacientes.createIndex( { "nombre" : 1 , "edad":-1} )
```

- El índice que se generará con la instrucción anterior, agrupará los datos primero por el campo **nombre** y luego por el campo **edad**. Es decir, se generaría algo así:

"Álvaro", 35

"Álvaro", 18

"María", 30

"María", 21

- Los índices compuestos se pueden usar para consultar uno o varios de los campos, sin que sea necesario incluirlos todos.
 - El índice compuesto ordenará el primer campo de la misma manera que si creásemos un índice simple. Lo que no podemos hacer es buscar únicamente sobre los siguientes campos (p.e. edad). Sería necesario crear un índice específico.
 - Si el índice compuesto tuviera tres campos, podríamos utilizarlo al consultar sobre el primer campo, sobre el primer y segundo campo, o sobre los tres campos.

Creación y Uso de Índices

Tipos de Índices

Índices únicos:

- Los índices simples y compuestos, pueden estar obligados a contener **valores únicos**. Esto lo conseguimos añadiendo el parámetro **unique** a la hora de crearlos.
- `db.pacientes.createIndex({ "nombre" : 1 } , {"unique":true})`
- En el caso de que tratemos de insertar valores repetidos en algún campo con un índice único, MongoDB nos devolverá un error.

Creación y Uso de Índices

Tipos de Indices

Índices *sparse*:

- Los índices vistos hasta ahora incluyen **todos** los documentos, también los que **no** tienen el **campo indexado**. Para los documentos que no tienen el campo indexado almacena un **valor nulo** en el índice.
- Para crear índices que **solo incluyan los documentos cuyo campo indexado existe**, utilizaremos la **opción sparse** (índice disperso).
- `db.pacientes.createIndex({ "nombre" : 1 } , {"sparse":true})`
- En este caso lo que hacemos es crear un índice que no tiene porqué contener todos los documentos de la colección.

Creación y Uso de Índices

Tipos de Índices

Índices parciales:

- Los índices parciales sólo indexan los documentos de una colección que **satisfacen una condición** (expresión).
- Requieren **menos espacio de almacenamiento** y **menos coste de creación y mantenimiento**.
- Para crear índices que solo incluyan los documentos que satisfagan la condición de filtrado, utilizaremos la **opción `partialFilterExpression`** .

```
db.pacientes.createIndex( { "nombre":1} , {partialFilterExpression: { "edad": { $gt: 18}}})
```

```
db.pacientes.createIndex( { "nombre":1} , {partialFilterExpression: { "nombre": { $exists: true}}})
```

- Esta opción existe a partir de la versión de MongoDB 3.2: se recomienda elegir esta opción (superconjunto) frente la opción de *sparse* (más funcionalidad).

Creación y Uso de Índices

Tipos de Índices

Índices parciales:

```
db.pacientes.createIndex( { "nombre":1} , {partialFilterExpression: {"edad":{ $gt:18}}})
```

- La opción **partialFilterExpression** acepta un documento que especifica una condición usando:
 - Expresiones **\$eq** de igualdad
`{ "nombre": "Juan" } o { "nombre": { $eq: "Juan" } }`
 - Expresión **\$exists: true** -> Sólo los documentos donde exista un determinado campo
`{ partialFilterExpression: { edad: { $exists: true } } }`
 - Expresiones: **\$gt, \$gte, \$lt, \$lte**
`{ "edad": { $gte: 18 } }`
 - Expresión **\$type** -> Sólo los documentos donde un campo sea de un tipo concreto
`{ "nombre": { $type: "string" } }`
 - Operador **\$and** solo en el nivel superior del documento

Creación y Uso de Índices

Intersección de índices

- **Antes de la versión 2.6 de MongoDB**, cada consulta podía utilizar como máximo un índice. Esto era un problema en algunos casos.

- **Ejemplo:**

Una consulta que ordena por nombre de forma ascendente y edad de forma descendente. Creando un índice compuesto, como hemos visto antes, tendríamos el problema solucionado.

```
db.pacientes.createIndex( { "nombre" : 1, "edad":-1 } )
```

¿Pero y si necesitamos también ejecutar consultas que devuelvan los resultados ordenados por nombre ascendente y edad también ascendente?

Nos veríamos obligados a crear otro índice, muy similar al anterior.

```
db.pacientes.createIndex( { "nombre" : 1, "edad":1 } )
```

- Para paliar estos problemas, aparecen los **índices cruzados**. De esta manera MongoDB puede usar **varios índices en una consulta**, para así mejorar el rendimiento de la misma.

Creando dos índices, tendríamos el problema solucionado.

```
db.pacientes.createIndex( { "nombre" : 1 } )
```

```
db.pacientes.createIndex( { "edad" : -1 } )
```

Creación y Uso de Índices

Indexación de subdocumentos

- **Ejemplo:**

```
db.pacientes.insert({"_id":"100",  
"nombre":"Sergio",  
"edad":30,  
"direccion": {"TipoVia":"calle",  
"Via":"Tulipan s/n",  
"CP":"28933",  
"ciudad":"Móstoles"}})
```

- Si necesitamos crear un índice sobre el campo ciudad, podríamos hacerlo sin problema:

```
db.pacientes.createIndex( { "direccion.ciudad" : 1 } )
```

- También podríamos crear el índice sobre el nodo principal **direccion**, pero esto es algo que solo nos ayudaría en el caso de que se realizaran consultas sobre el subdocumento completo y con los campos exactamente en el mismo orden.

Creación y Uso de Índices

Indexación de arrays

- Indexar arrays también es algo posible con MongoDB aunque es algo que deberemos hacer con cuidado y teniendo en cuenta algunas limitaciones:

1) La primera limitación es que **sólo uno de los campos del índice puede ser un array.**

```
db.users.createIndex( { "categories" : 1, "tags": 1 } )
```

Si intentamos insertar un documento con dos arrays, obtendremos un error:

```
db.users.insert({"categories":["game","book","movie"], "tags":["horror","scifi","history"]})  
cannot index parallel arrays [categories] [tags]
```

⇒ Habría que crear una entrada en el índice para el producto cartesiano de elementos. En el caso de que los arrays fuesen grandes, sería una operación inmanejable.

2) La otra limitación es que los elementos indexados del array **no tienen en cuenta el orden.** Por tanto, si realizamos consultas posicionales sobre el array no se usará el índice.

Creación y Uso de Índices

Consultas totalmente cubiertas por índices:

- Aunque tengamos varios índices creados, no todas las consultas van a ser igual de eficientes. Las consultas más rápidas serán las denominadas **consultas totalmente cubiertas**. Son aquellas consultas cuyos **campos consultados y devueltos** están incluidas en el índice.
- Ejemplo de índice compuesto:

```
db.clientes.createIndex( { "nombre" : 1, "edad":1 } )
```
- Si hacemos una búsqueda que utilice en la consulta nombre y/o edad, y además, con los valores devueltos nombre y/o edad, estaríamos hablando de una consulta totalmente cubierta.
- MongoDB no tendrá que buscar en disco el documento completo, ya que los valores serán devueltos directamente desde el índice. Siempre que podamos, deberemos hacer consultas de este tipo.

Creación y Uso de Índices

Analizando planes de ejecución en MongoDB

- MongoDB tiene una utilidad para obtener información sobre el plan de ejecución de una consulta **explain()**.

`db.colección.find().explain()` o `db.colección.explain().find()`

- Devolverá un documento JSON con información útil:
 - número de documentos procesados para devolver el resultado
 - si la consulta ha utilizado los índices o si scan de la colección
 - si ha tenido que ordenar los resultados en memoria
 - información sobre el servidor, etc.

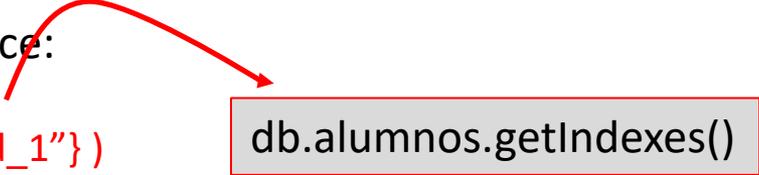
Creación y Uso de Índices

- Método `hint ()`
- Este método permite indicar en una consulta **qué índice utilizar**.
O bien mediante la especificación del índice

Ejemplo: `db.alumnos.find().hint({ edad: 1 })`

O bien indicando el nombre del índice:

Ejemplo : `db.alumnos.find().hint({ "edad_1" })`



```
db.alumnos.getIndexes()
```

- Para obligar a realizar en una consulta un *collection scan* (no usar ningún índice):
`{ $natural : 1 }`

Ejemplo (forward collection scan): `db.alumnos.find().hint({ $natural : 1 })`

Ejemplo (reverse collection scan): `db.alumnos.find().hint({ $natural : -1 })`

Creación y Uso de Índices

- Recomendación **inserción masiva** en una colección:
 1. Borrar los índices con `db.colección.dropIndexes()`
 2. Inserción de documentos
 3. Re-Creación de índices `db.colección.createIndex()`

El tiempo de creación de los índices es mucho menor que el tiempo necesario en actualizar el mismo índice ya creado en cada inserción de cada documento.

Creación y Uso de Índices

- Recreación de los índices de una colección mediante el método `reIndex()`

Ejemplo: `db.clientes.reIndex()`

- Este método **borra todos los índices** de una colección y los **recrea**.
- Se trata de una **operación costosa** para colecciones grandes y/o con un número elevado de índices.
- Apropiado cuando el tamaño de la colección ha **cambiado mucho** o cuando el **disco usado** por los índices es desproporcionadamente elevado.

Creación y Uso de Índices

Tipos de Indices

- **Otros tipos de índices**
- Existen también otros tipos de índices como son los índices *Hash*, los índices para geoposicionamiento y los índices de texto.

Índice

- *Introducción a MongoDB*
- *Configuración del Entorno de Trabajo*
- *Operaciones CRUD con MongoDB*
- *Creación y Uso de Índices*
- **Aggregation Framework**

Aggregation Framework

- Las agregaciones procesan registros y devuelven resultados calculados.
- Las **operaciones de agregación** agrupan valores de varios documentos, y realizan un conjunto de operaciones en los datos agrupados para devolver un resultado único.
- MongoDB proporciona tres formas de realizar agregaciones:
 - Métodos de agregación sencillos: `count()`, `distinct()`...
 - La función map-reduce
 - **El Aggregation Framework**

Aggregation Framework

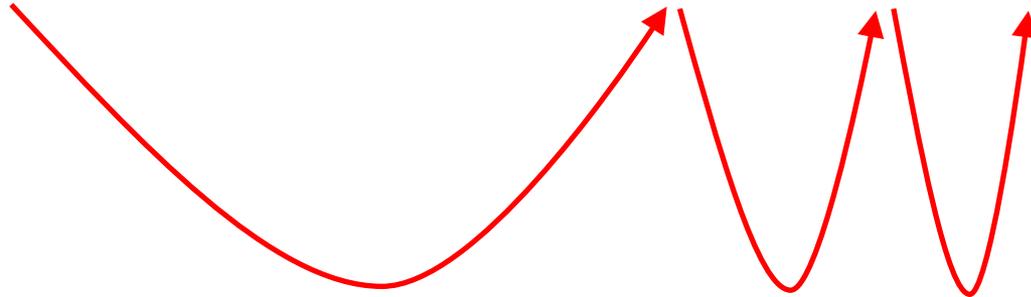
- Es un framework para agregación de datos basado en el concepto de **pipeline**.
- Los documentos **entran** en un pipeline de **varias etapas** que **transforma** los documentos en **resultados agregados**.
- Tiene algunas limitaciones: tipos de valores y tamaño del resultado.

Aggregation Framework

- Los pipelines del Aggregation Framework funcionan como un **conjunto de etapas** que transforman los documentos.
- Las **etapas** pueden funcionar como filtros (consultas), agrupar y ordenar documentos, resumir contenidos de arrays, etc...
- Es el método preferido para agregar datos en MongoDB.

Aggregation Framework

- El pipeline consta de etapas.
- Cada etapa transforma los documentos a medida que pasan por las etapas.
- Las etapas pueden repetirse en el pipeline
 - `db.collection.aggregate([<etapa1>,...])`



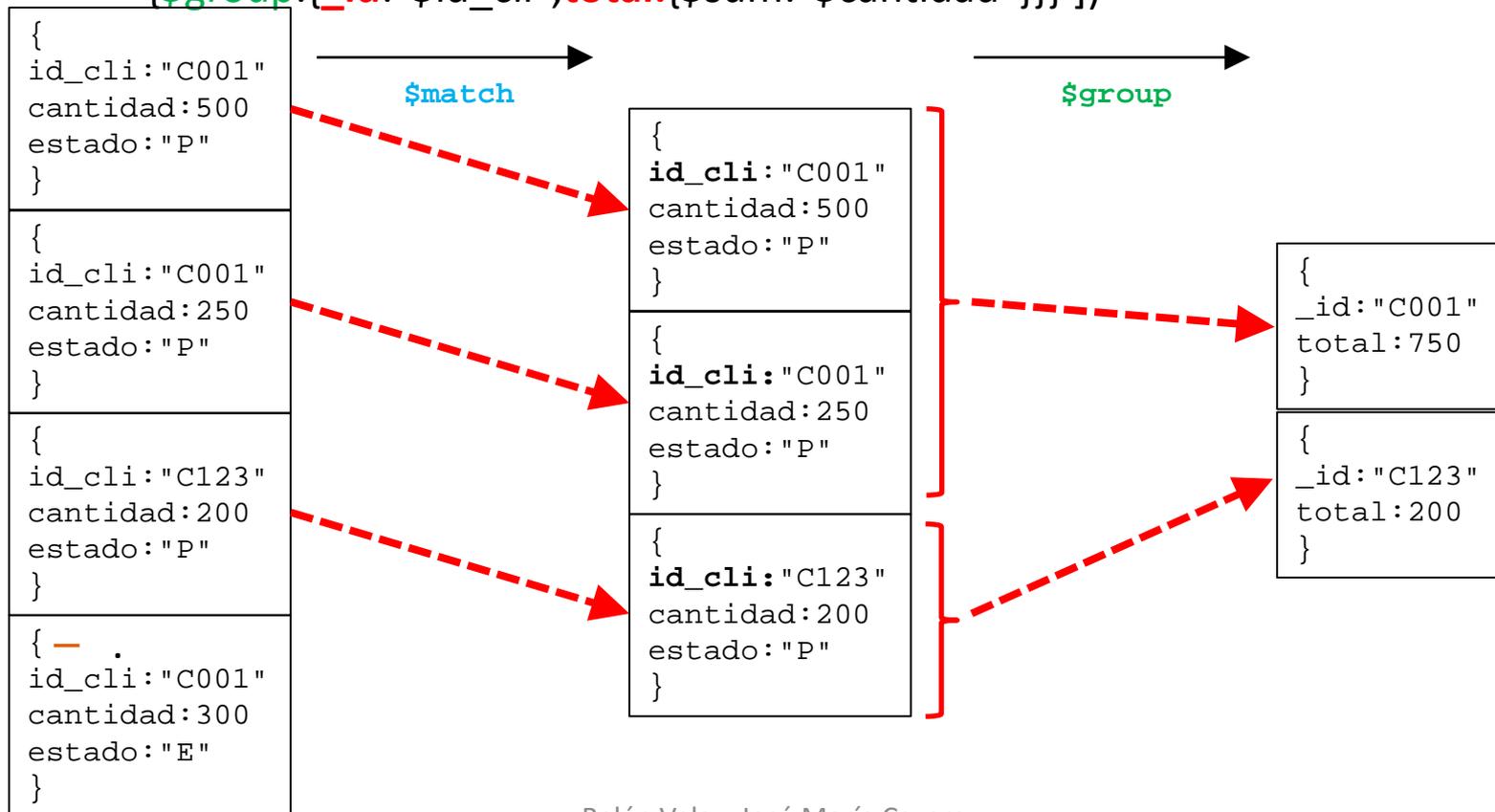
Aggregation Framework

- Las **etapas** pueden incluir **expresiones** con operadores que reciban un argumento o un array de argumentos:
`db.collection.aggregate([<etapa1>, ..., <etapaN>])`
 - **<operador>**: [<argumento1>, <argumento2>...]
 - » \$and, \$or, \$not, \$setEquals, \$setIntersection, \$setUnion, \$setDifference, \$setIsSubset, \$cmp, \$eq, \$gt, \$abs, \$add, \$subtract, \$concat, \$toUpper, \$size...
 - » **Acumuladores**: en la etapa **\$group**: \$sum, \$avg, \$first, \$last, \$max, \$min, \$push, \$addToSet, ...
 - Algunos de estos operadores también se pueden usar en \$Project (no como acumuladores).
- Cada operador puede producir 0, 1 o n **documentos de salida** a partir de 1 o n **documentos de entrada** (1:1, 1:0..1, 1:0..n, n:1,...)

Aggregation Framework

```
db.ventas.aggregate([  
  { $match: { estado: "P" } },  
  { $group: { _id: "$id_cli", total: { $sum: "$cantidad" } } } ])
```

Acumulador



Aggregation Framework

Para importar a partir de un fichero una colección de documentos en una base de datos:

- `mongoimport --host <host><:puerto>`
 - `--db <base de datos>`
 - `--collection <colección>`
 - `--file <fichero>`

Ejemplo: Importar colección de Provincias en BDNC

- `mongoimport`
 - `--db BDNC`
 - `--collection provincias`
 - `--file "PATH/provincias.json"`

Aggregation Framework

2.4	2.6	3.2	3.4	3.6
<u>\$project</u>	<u>\$out</u>	<u>\$sample</u>	<u>\$collStats</u>	<u>\$currentOp</u>
<u>\$match</u>	<u>\$redact</u>	<u>\$lookup</u>	<u>\$facet</u>	<u>\$listLocalSessions</u>
<u>\$limit</u>		<u>\$indexStats</u>	<u>\$bucket</u>	<u>\$listSessions</u>
<u>\$skip</u>			<u>\$bucketAuto</u>	
<u>\$unwind</u>			<u>\$sortByCount</u>	
<u>\$group</u>			<u>\$addFields</u>	
<u>\$sort</u>			<u>\$replaceRoot</u>	
<u>\$geoNear</u>			<u>\$count</u>	
			<u>\$graphLookup</u>	

Operadores

- **\$sample**

- Devuelve un **conjunto de N documentos de una colección al azar.**

```
db.libros.aggregate(  
    [{$sample: { size: <número>}}])
```

- **Ejercicio:**

- Obtener tres provincias al azar.

Operadores

- **\$match**

- Filtra los documentos que cumplen las condiciones

- { \$match: { <consulta> } }
 - » La sintaxis de la consulta es como una lectura
 - » Esta etapa conviene ponerla al principio.

```
db.libros.aggregate( [{ $match : { autor : "Cervantes" } } ] )
```

```
db.provincias.aggregate ( [ { $match: { Superficie: { $lt: 100 } } } ] )  
db.provincias.find( { Superficie: { $lt: 100 } } )
```

- **Ejercicio:**

- Obtener cinco provincias de Andalucía (CA: “Andalucía”).

Operadores

- **\$project**

- Pasan a la siguiente etapa con los campos especificados. Los campos pueden ser preexistentes o creados en esta etapa.

- {\$project: { <especificaciones de campos> }}
- Pueden añadirse o eliminarse campos (incluido el `_id`).

Operadores

- \$project

- Sintaxis:

- <campo>: <1 o true> -- se incluye el campo (también el _id)
 - <campo>: <0 o false> -- se excluye (e impide cualquier otro tipo de especificación, salvo excluir campos, en este caso por defecto sí aparecen)
 - _id: <0 o false> -- se suprime el _id (por defecto, sí se incluye)
 - <campo>: <expresión> -- añade o resetea el valor
 - <camponuevo>: \$<campo> -- para renombrar un campo
 - En embebidos, “a.b.c” : <1/0/exp> o a:{b:{c: <1/0/exp>}}
 - También se pueden quitar campos de manera opcional

Operadores

- \$project

- Ejemplos

- db.provincias.aggregate ([[{\$project:{"Nombre":1}}])
 - Devuelve únicamente el Nombre de las provincias y su _id.
 - db.provincias.aggregate ([[{\$project:{"Nombre":1,"_id":0}}])
 - Devuelve únicamente el Nombre de las provincias (sin _id).
 - db.provincias.aggregate ([[{\$project:{"Provincia":"\$Nombre","_id":0}}]])
 - Se renombra el campo Nombre y se excluye el _id.

- **Ejercicio :**

- Obtener el nombre (que pasará a ser el identificador _id) y superficie de 3 ciudades de Andalucía.

Operadores

- **\$sort**

- Ordena los documentos

- `{ $sort: { <campo1>: <ordenación>, <campo2>: <ordenación> ... } }`

- Ejemplo

- `db.provincias.aggregate ([{$sort:{Superficie:1}}, {$project:{"Nombre":1}}])`

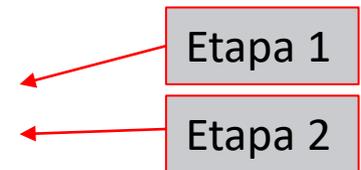
- Devuelve únicamente el Nombre de las provincias y su `_id`, ordenados ascendentemente por la Superficie de la provincia

- `¿db.provincias.aggregate ([$project:{"Nombre":1}], {$sort:{Superficie:1}})?`

- Al hacer la proyección los únicos dos campos que existen son el Nombre y el `_id`, por lo tanto, simplemente no ordena los documentos.

- **Ejercicio :**

- Obtener el nombre (renombrando el campo a `_id`) y la superficie de 5 ciudades de Andalucía al azar ordenadas por superficie.



Operadores

- **\$limit**

- Limita el número de documentos que pasan a la siguiente etapa, sin afectar al contenido.
 - { \$limit: <entero positivo> }
- Ejemplo
 - db.libros.aggregate({ \$limit : 5 });
 - » Nombre (identificador) y superficie de las 5 ciudades de Andalucía más extensas ordenadas por superficie
 - » Optimización \$sort + \$limit: solo ordena los documentos indicados por el limit.

- **\$skip**

- Ignora los primeros n documentos.
 - { \$skip: <entero positivo> }
- Ejemplo
 - db.libros.aggregate({ \$skip : 5 });

Operadores

- **\$out**

- Escribe la salida a una colección.
- Debe ser la **última etapa** del pipeline.
- { \$out: "<colección de salida>" }
- » Si la operación falla, la colección no se crea
- » Si la colección ya existe, se sustituye por la nueva

- **Ejercicio:**

Obtener una nueva colección provinciasCLM que contenga documentos con el nombre y la superficie de las provincias de Castilla-La Mancha..

Aggregation Framework

2.4	2.6	3.2	3.4	3.6
<u>\$project</u>	<u>\$out</u>	<u>\$sample</u>	<u>\$collStats</u>	<u>\$currentOp</u>
<u>\$match</u>	<u>\$redact</u>	<u>\$lookup</u>	<u>\$facet</u>	<u>\$listLocalSessions</u>
<u>\$limit</u>		<u>\$indexStats</u>	<u>\$bucket</u>	<u>\$listSessions</u>
<u>\$skip</u>			<u>\$bucketAuto</u>	
<u>\$unwind</u>			<u>\$sortByCount</u>	
<u>\$group</u>			<u>\$addFields</u>	
<u>\$sort</u>			<u>\$replaceRoot</u>	
<u>\$geoNear</u>			<u>\$count</u>	
			<u>\$graphLookup</u>	

Operadores

- **\$group**

- Agrupa los documentos de acuerdo a alguna expresión. El documento de salida contiene un **_id** que contiene el identificador de los distintos grupos. Puede contener campos calculados mediante algún acumulador.

- `{ $group: { _id: <expresión>, <campo1>: { <acumulador1> : <expresión1> }, ... } }`

- » El campo **_id** es **obligatorio**. Si se pone null, agrupa todos los documentos (`_id : null`)

- » Acumuladores: `$sum`, `$avg`, `$first`, `$last`, `$max`, `$min`, `$push`, `$addToSet`...

Operadores

- \$group

Ejemplo collection:

```
{
  estado: "New York",
  areaM: 218,
  region: "North East"}
{
  estado: "New Jersey",
  areaM: 90,
  region: "North East"}
{
  estado: "California",
  areaM: 300,
  region: "West"}
```

Operadores

- \$group

- Ejemplo:

```
db.coleccion.aggregate([{$group:
```

```
  {_id: "$region",
```

```
  avgAreaM: {$avg: "$areaM"},
```

```
  estados: {$push: "$estado"}}])
```

⇒ Campo por el que agrupo (region), será el _id

⇒ Calculo la media del área por region

⇒ Array estados con todos los valores de estado por region

- OBTENGO:

```
{_id: "Noth East",
```

```
  avgAreaM: 154,
```

```
  estados:["New York","New Jersey"]
```

```
}
```

```
{_id: "West",
```

```
  avgAreaM: 300,
```

```
  estados:["California"]
```

```
}
```

acumuladores

Operadores

- \$group

- Ejemplo:

```
db.provincias.aggregate ([{$group:  
  {"_id":"$CA",  
   "media":{"$avg":"$Superficie"}}}])
```

- **Ejercicios:**

- Superficie y número de provincias total.
 - Nueva colección con un documento por cada Comunidad, que contenga el número de provincias, el total de superficie y un array con las provincias.

Operadores

- **\$unwind**

- Descompone un documento en tantos documentos como elementos tenga el array.

Opción 1:

- {\$unwind: <campo>}

- Si no es un array, lo trata como un array de 1.
- Si está vacío o no existe, lo ignora.

- Ejemplo:

- {"_id": 1, "item": "ABC1", "tallas": ["S", "M", "L"]}
- db.inventory.aggregate([{\$unwind : "\$tallas" }])

```
{ "_id" : 1, "item" : "ABC1", "tallas" : "S" }  
{ "_id" : 1, "item" : "ABC1", "tallas" : "M" }  
{ "_id" : 1, "item" : "ABC1", "tallas" : "L" }
```

Operadores

- \$unwind

```
db.provincias.aggregate(  
  [{$unwind: "$Datos"}]) -- Datos es un Array de Año - Valor
```

```
db.provincias.aggregate(  
  [{$unwind: "$Datos"},  
    {$match: {"Datos.Anyo":2015}},  
    {$project:  
      {"Nombre":1,  
        "CA":1,  
        "Poblacion":"$Datos.Valor"}}])
```

- **Ejercicio:**

- Obtener una colección con un documento por provincia que contenga la población del último año guardado (sin asumir que es el 2015).

Operadores

- **Ejercicios:**

- Obtener una nueva colección CAconProvincias con un documento por cada Comunidad, que contenga el número de provincias, el total de superficie y un array con las provincias.
- Obtener el año con mayor población en España.
- Obtener un documento por provincia, con la población del primer año almacenado, el último y la diferencia.
- Obtener la comunidad autónoma que menos ha aumentado su densidad de población (o que más lo ha disminuido).

Operadores

- \$unwind

Opción 2:

```
{ $unwind: {  
    path: <campo>,  
    includeArrayIndex: <string>,  
    preserveNullAndEmptyArrays: <boolean>  
  }  
}
```

Donde:

campo: cadena que identifica un array (p.e. “\$Datos”)

includeArrayIndex: nombre del campo que contendrá el índice del valor que desagrega. Si no es un array, tendrá un null.

preserveNullAndEmptyArrays: Por defecto, es falso, y no incluye los valores de la colección en el caso de que <campo> no contenga nada. Si vale true, los incluye.

Operadores

Ejemplo :

```
db.productos.insertMany([
  { "_id" : 1, "item" : "camiseta", precio: 29, "tallas": [ "S", "M", "L" ] },
  { "_id" : 2, "item" : "sudadera", precio: 75, "tallas" : [ ] },
  { "_id" : 3, "item" : "camisa", precio: 59, "tallas": "L" },
  { "_id" : 4, "item" : "bufanda" ,precio: 40 },
])
```

```
db.productos.aggregate([
  {$unwind:
    {path:"$tallas",
    includeArrayIndex:"arrayIndex",
    preserveNullAndEmptyArrays:true}},
  ])
```

Resultado:

```
{ "_id" : 1, "item" : "camiseta", "precio" : 29, "tallas" : "S", "arrayIndex" : NumberLong(0) }
{ "_id" : 1, "item" : "camiseta", "precio" : 29, "tallas" : "M", "arrayIndex" : NumberLong(1) }
{ "_id" : 1, "item" : "camiseta", "precio" : 29, "tallas" : "L", "arrayIndex" : NumberLong(2) }
{ "_id" : 2, "item" : "sudadera", "precio" : 75, "arrayIndex" : null }
{ "_id" : 3, "item" : "camisa", "precio" : 59, "tallas" : "L", "arrayIndex" : null }
{ "_id" : 4, "item" : "bufanda", "precio" : 40, "arrayIndex" : null }
```

includeArrayIndex

preserveNullAndEmptyArrays

Operadores

Ejemplos:

```
db.provincias.aggregate([
  {$unwind: {path:"$Datos",
            includeArrayIndex:"arrayIndex"}},
  {$limit:10}])
```

Resultado:

```
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2015, "Valor" : 320663 }, "arrayIndex" : NumberLong(0) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2014, "Valor" : 319895 }, "arrayIndex" : NumberLong(1) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2013, "Valor" : 319927 }, "arrayIndex" : NumberLong(2) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2012, "Valor" : 320566 }, "arrayIndex" : NumberLong(3) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2011, "Valor" : 319846 }, "arrayIndex" : NumberLong(4) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2010, "Valor" : 317737 }, "arrayIndex" : NumberLong(5) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2009, "Valor" : 315280 }, "arrayIndex" : NumberLong(6) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2008, "Valor" : 310944 }, "arrayIndex" : NumberLong(7) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2007, "Valor" : 306475 }, "arrayIndex" : NumberLong(8) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco", "Superficie" : 3037,
  "Datos" : { "Anyo" : 2006, "Valor" : 302481 }, "arrayIndex" : NumberLong(9) }
```

Operadores

Ejemplos:

```
db.provincias.aggregate([
  {$unwind: {path:"$Datos",
            includeArrayIndex:"arrayIndex",
            preserveNullAndEmptyArrays:true}},
  {$match:{arrayIndex:2}},
  {$limit:5}])
```

Resultado:

```
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco",
  "Superficie" : 3037, "Datos" : { "Anyo" : 2013, "Valor" : 319927 }, "arrayIndex" : NumberLong(2) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdc"), "Nombre" : "Asturias", "CA" : "Principado de Asturias",
  "Superficie" : 10603, "Datos" : { "Anyo" : 2013, "Valor" : 1067802 }, "arrayIndex" : NumberLong(2) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdd"), "Nombre" : "Ávila", "CA" : "Castilla y León", "Superficie"
  : 8050, "Datos" : { "Anyo" : 2013, "Valor" : 169458 }, "arrayIndex" : NumberLong(2) }
{ "_id" : ObjectId("5e636e942354d38e20a48cde"), "Nombre" : "Badajoz", "CA" : "Extremadura",
  "Superficie" : 21766, "Datos" : { "Anyo" : 2013, "Valor" : 690894 }, "arrayIndex" : NumberLong(2) }
{ "_id" : ObjectId("5e636e942354d38e20a48cdf"), "Nombre" : "Islas Baleares", "CA" : "Islas Baleares",
  "Superficie" : 4991, "Datos" : { "Anyo" : 2013, "Valor" : 1110115 }, "arrayIndex" : NumberLong(2) }
```

Operadores

Ejemplos:

```
db.provincias.aggregate([
  {$unwind:
    {path:"$CA",
    includeArrayIndex:"arrayIndex",
    preserveNullAndEmptyArrays:true}},
  {$project:{Datos:0}},
  {$limit:5}])
```

includeArrayIndex
⇒ CA es un array con un único elemento

Resultado:

```
{ "_id" : ObjectId("5e636e942354d38e20a48cdb"), "Nombre" : "Araba/Álava", "CA" : "País Vasco",
"Superficie" : 3037, "arrayIndex" : null }
{ "_id" : ObjectId("5e636e942354d38e20a48cdc"), "Nombre" : "Asturias", "CA" : "Principado de Asturias",
"Superficie" : 10603, "arrayIndex" : null }
{ "_id" : ObjectId("5e636e942354d38e20a48cdd"), "Nombre" : "Ávila", "CA" : "Castilla y León", "Superficie"
: 8050, "arrayIndex" : null }
{ "_id" : ObjectId("5e636e942354d38e20a48cde"), "Nombre" : "Badajoz", "CA" : "Extremadura",
"Superficie" : 21766, "arrayIndex" : null }
{ "_id" : ObjectId("5e636e942354d38e20a48cdf"), "Nombre" : "Islas Baleares", "CA" : "Islas Baleares",
"Superficie" : 4991, "arrayIndex" : null }
```

Aggregation Framework

2.4	2.6	3.2	3.4	3.6
<u>\$project</u>	<u>\$out</u>	<u>\$sample</u>	<u>\$collStats</u>	<u>\$currentOp</u>
<u>\$match</u>	<u>\$redact</u>	<u>\$lookup</u>	<u>\$facet</u>	<u>\$listLocalSessions</u>
<u>\$limit</u>		<u>\$indexStats</u>	<u>\$bucket</u>	<u>\$listSessions</u>
<u>\$skip</u>			<u>\$bucketAuto</u>	
<u>\$unwind</u>			<u>\$sortByCount</u>	
<u>\$group</u>			<u>\$addFields</u>	
<u>\$sort</u>			<u>\$replaceRoot</u>	
<u>\$geoNear</u>			<u>\$count</u>	
			<u>\$graphLookup</u>	

Operadores

- **\$sortByCount**

- Agrupa los documentos y los cuenta por grupos, devolviendo en el resultado el `_id` correspondiente a la agrupación y un valor con el número de documentos.

```
{ $sortByCount: <expression> }
```

Es equivalente a:

```
{ $group: { _id: <expression>, count: { $sum: 1 } } },  
{ $sort: { count: -1 } }
```

Operadores

- \$sortByCount

- Ejemplo:

Dados los siguientes documentos:

```
{ "_id" : 1, "title" : "The Pillars of Society", "artist" : "Grosz", "year" : 1926, "tags" : [ "painting", "Expressionism" ] }
```

```
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch", "year" : 1902, "tags" : [ "woodcut", "Expressionism" ] }
```

```
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925, "tags" : [ "oil", "Surrealism", "painting" ] }
```

```
{ "_id" : 4, "title" : "The Persistence of Memory", "artist" : "Dali", "year" : 1931, "tags" : [ "Surrealism", "painting", "oil" ] }
```

```
{ "_id" : 5, "title" : "Composition VII", "artist" : "Kandinsky", "year" : 1913, "tags" : [ "oil", "painting", "abstract" ] }
```

```
{ "_id" : 6, "title" : "The Scream", "artist" : "Munch", "year" : 1893, "tags" : [ "Expressionism", "painting", "oil" ] }
```

Operadores

- `$sortByCount`

La siguiente consulta:

```
db.coleccion.aggregate([
  {$unwind: "$tags"},{$sortByCount: "$tags" }
])
```

Devuelve el siguiente resultado:

```
{ "_id" : "painting", "count" : 5 }
{ "_id" : "oil", "count" : 4 }
{ "_id" : "Expressionism", "count" : 3 }
{ "_id" : "Surrealism", "count" : 2 }
{ "_id" : "abstract", "count" : 1 }
{ "_id" : "woodcut", "count" : 1 }
```

- **Ejercicios:**

- Obtener el número de provincias de cada comunidad autónoma usando `sortByCount`.

Operadores

- **\$addField**

- Añade nuevos campos a documentos, manteniendo los que existen.
- Es equivalente a un \$project que mantenga todos los campos y añada otros nuevos.

```
{ $addField: { campo: <expresión>, ... } }
```

- Si el campo ya existe (incluyendo `_id`), se sobrescribe.

Operadores

- \$addFields

Ejemplo:

```
db.alumnos.insertMany([
  { _id: 1,
    estudiante: "Juan",
    deberes: [ 10, 5, 10 ],
    pruebas: [ 10, 8 ] },
  { _id: 2,
    estudiante: "María",
    deberes: [ 5, 6, 5 ],
    pruebas: [ 8, 8 ] }])
```

Si ejecutamos:

```
db.alumnos.aggregate( [
  { $addFields: { totaldeberes: { $sum: "$deberes" },
                 totalpruebas: { $sum: "$pruebas" } } },
  { $addFields: { notaFinal: { $add: [ "$totaldeberes",
                                       "$totalpruebas" ] } } } ] )
```

Operadores

- \$addField

Resultado:

```
{ "_id" : 1,  
  "estudiante" : "Juan",  
  "deberes" : [ 10, 5, 10 ],  
  "pruebas" : [ 10, 8 ],  
  "totaldeberes" : 25,  
  "totalpruebas" : 18,  
  "notaFinal" : 43 }
```

```
{ "_id" : 2,  
  "estudiante" : "María",  
  "deberes" : [ 5, 6, 5 ],  
  "pruebas" : [ 8, 8 ],  
  "totaldeberes" : 16,  
  "totalpruebas" : 16,  
  "notaFinal" : 32 }
```

Operadores

- **\$addField**

Se pueden añadir campos a un documento embebido:

Ejemplo:

```
db.vehiculos.insertMany([
  { _id: 1, tipo: "coche", especificaciones: { puertas: 4, ruedas: 4 } },
  { _id: 2, tipo: "motocicleta", especificaciones: { puertas: 0, ruedas: 2 } },
  { _id: 3, tipo: "moto de agua" } ])
```

```
db.vehiculos.aggregate( [
  { $addField: { "especificaciones.combustible": "sin plomo" } } ] )
```

Resultado:

```
{ _id: 1, tipo: "coche",
  especificaciones: { puertas: 4, ruedas: 4, combustible: "sin plomo" } }
{ _id: 2, tipo: "motocicleta",
  especificaciones: { puertas: 0, ruedas: 2, combustible: "sin plomo" } }
{ _id: 3, tipo: "moto de agua",
  especificaciones: { combustible: "sin plomo" } }
```

Se puede reescribir un campo, incluido el `_id`

```
db.vehiculos.aggregate( [
  { $addField: { _id: "$tipo",
  tipo: "vehículo" } } ] )
```

Resultado:

```
{ "_id" : "coche", "tipo" : "vehículo", "especificaciones" : { "puertas" : 4, "ruedas" : 4 } }
{ "_id" : "motocicleta", "tipo" : "vehículo", "especificaciones" : { "puertas" : 0, "ruedas" : 2 } }
{ "_id" : "moto de agua", "tipo" : "vehículo" }
```

- **Ejercicio:**

Obtener las provincias incluyendo un campo que contenga el nombre de la provincia y el de la CA (`$concat`).

Operadores

- **\$count**

- Devuelve un documento que contiene el número de documentos que entran en la etapa.

```
{ $count: <string> }
```

<string> es el nombre del campo que contendrá la salida (número de documentos).

- **Ejemplo: colección "notas" con los siguientes documentos:**

```
{"_id" : 1, "asignatura" : "Historia", "nota" : 45 }  
{"_id" : 2, "asignatura" : "Historia", "nota" : 79 }  
{"_id" : 3, "asignatura" : "Historia", "nota" : 97 }
```

- ```
db.notas.aggregate(
 [{ $match: {nota: {$gt: 50}}},
 { $count: "aprobados"}
])
```

Resultado:

```
{"aprobados" : 2}
```

- **Ejercicio:**

Número total de provincias utilizando \$count.

# Operadores

- **\$bucket**

- Clasifica los documentos en grupos (buckets)

db.colección.aggregate ([{

\$bucket:

{groupBy: <expresión>,

boundaries: [ limite1, limite2 ...],

default: <literal para fuera de rango>,

output:{ campo1: {<expresión con acum>},

campo2: {<expresión con acum>}...

}}})

El array *boundaries* [ 0, 5, 10 ] crea dos buckets:

• [0, 5): 0, límite inferior inclusivo y 5, límite superior exclusivo

• [5, 10): 5, límite inferior inclusivo y 10, límite superior exclusivo

# Operadores

- \$bucket

- Ejemplo de clasificación de los documentos en grupos (buckets)

```
db.provincias.aggregate ({
```

```
 $bucket:
```

```
 {groupBy: "$Superficie",
```

```
 boundaries: [0, 1000, 10000],
```

```
 default: "Otros",
```

```
 output: { "numProvincias": { $sum: 1 },
```

```
 "provincias": { $push: "$Nombre" }
```

```
 } } })
```

Resultado:

```
{ "_id" : 0, "numProvincias" : 2, "provincias" : ["Ceuta", "Melilla"] }
```

```
{ "_id" : 1000, "numProvincias" : 25, "provincias" : ["Araba/Álava", "Ávila", "Islas Baleares", "Barcelona",
"Bizkaia", "Cádiz", "Cantabria", "Castellón/Castelló", "Alicante/Alacant", "Coruña, A", "Almería", "Gipuzkoa",
"Girona", "Lugo", "Madrid", "Málaga", "Palencia", "Las Palmas", "Ourense", "Pontevedra", "La Rioja", "Segovia",
"Santa Cruz de Tenerife", "Valladolid", "Tarragona"] }
```

```
{ "_id" : "Otros", "numProvincias" : 25, "provincias" : ["Asturias", "Badajoz", "Burgos", "Cáceres", "Ciudad Real",
"Córdoba", "Albacete", "Cuenca", "Granada", "Guadalajara", "Huelva", "Huesca", "Jaén", "León", "Lleida",
"Murcia", "Salamanca", "Navarra", "Sevilla", "Soria", "Toledo", "Valencia/València", "Teruel", "Zamora",
"Zaragoza"] }
```

# Operadores

- **\$bucketAuto**

- Agrupa por rangos de valores:

```
db.provincias.aggregate([
 {$bucketAuto: {
 groupBy: "$Superficie",
 buckets: 5,
 output: {
 "numProvincias":{$sum: 1 },
 "provincias":{$push:"$Nombre"},
 "superficies":{$push:"$Superficie"}
 }
 }
]
})
```

Buckets: especificaremos el número de grupos que queremos que genere en base al campo de agregación.



# Operadores

- **\$bucketAuto**

- Resultado: Buckets: 5

```
{ "_id" : { "min" : 13, "max" : 5321 }, "numProvincias" : 10, "provincias" : ["Melilla", "Ceuta", "Gipuzkoa", "Bizkaia", "Araba/Álava", "Santa Cruz de Tenerife", "Las Palmas", "Pontevedra", "Islas Baleares", "La Rioja"], "superficies" : [13, 18, 1997, 2217, 3037, 3381, 4065, 4494, 4991, 5045] }
{ "_id" : { "min" : 5321, "max" : 7950 }, "numProvincias" : 10, "provincias" : ["Cantabria", "Alicante/Alacant", "Girona", "Tarragona", "Castellón/Castelló", "Segovia", "Ourense", "Málaga", "Cádiz", "Barcelona"], "superficies" : [5321, 5817, 5909, 6302, 6636, 6920, 7273, 7306, 7440, 7733] }
{ "_id" : { "min" : 7950, "max" : 10561 }, "numProvincias" : 10, "provincias" : ["Coruña, A", "Madrid", "Ávila", "Palencia", "Valladolid", "Almería", "Lugo", "Huelva", "Soria", "Navarra"], "superficies" : [7950, 8027, 8050, 8052, 8110, 8775, 9856, 10127, 10306, 10391] }
{ "_id" : { "min" : 10561, "max" : 14022 }, "numProvincias" : 10, "provincias" : ["Zamora", "Asturias", "Valencia/València", "Murcia", "Lleida", "Guadalajara", "Salamanca", "Granada", "Jaén", "Córdoba"], "superficies" : [10561, 10603, 10807, 11313, 12172, 12214, 12349, 12646, 13496, 13771] }
{ "_id" : { "min" : 14022, "max" : 21766 }, "numProvincias" : 12, "provincias" : ["Burgos", "Sevilla", "Teruel", "Albacete", "Toledo", "León", "Huesca", "Cuenca", "Zaragoza", "Ciudad Real", "Cáceres", "Badajoz"], "superficies" : [14022, 14036, 14809, 14926, 15369, 15580, 15636, 17140, 17274, 19813, 19868, 21766] }.
```

# Operadores

---

- **\$lookup**

- Permite hacer joins.
- Añade un array a cada documento para aquellos elementos que coinciden con la colección enlazada.

{**\$lookup**:

**from**: <colección2 a enlazar>,

**localField**: <campo de colección origen>,

**foreignField**: <campo de colección "from" >,

**as**: <array de salida>

}}

Array de documentos combinados.

Campos por los que hacemos el left outer join.

# Operadores

---

- \$lookup

- Ejemplo

- db.asignatura.insert({num : 1, nombre: "mongodb"})
    - db.asignatura.insert({num : 2, nombre: "relacional"})
    - db.temario.insert({asig : 1, tema: "CRUD"})
    - db.temario.insert({asig : 1, tema : "Aggregation Framework"})

- Mezclamos las asignaturas con los temarios:

- db.asignatura.aggregate  
([{\$lookup:{from : "temario",  
          localField : "num",  
                      foreignField : "asig",  
                      as : "temasdeasignatura"}} ])

# Operadores

- \$lookup

Resultado:

```
db.asignatura.aggregate([{$lookup:{from : "temario", localField : "num",foreignField : "asig", as : "temasdeasignatura"}}])
```



**Left outer join!!**

# Operadores

---

- \$lookup

- **Ejercicio:**

Genere las colecciones Comunidades y soloProvincias

```
db.provincias.aggregate ([
 {$group:{"_id":"$CA",
 "numProvincias":{$sum:1}}},
 {$out: "Comunidades"}])
```

```
db.provincias.aggregate([
 {$project:{Nombre:1,CA:1}},
 {$out:"soloProvincias"}])
```

Realice un lookup entre ambas colecciones.

# Limitaciones del Aggregation Pipeline

---

- Restricciones en el tamaño del resultado
  - Cada documento del resultado está limitado a **16 Mb**. No afecta a los resultados intermedios.
- Restricciones de memoria
  - Cada etapa del pipeline tiene un **límite de 100 Mb de RAM**. Para evitarlo, se puede usar la opción **allowDiskUse** para permitir la escritura en ficheros temporales.
- Más

# Aggregation Framework

---

- Otros aspectos:
  - Los operadores `$match` y `$sort` pueden aprovechar la existencia de **índices** cuando aparecen al principio del pipeline.
  - Si la operación de agregación requiere solo un subconjunto de los datos de la colección, conviene utilizar `$match`, `$limit`, y `$skip` para restringir los documentos al principio del pipeline, para optimizar el uso de índices.
  - El pipeline tiene una fase interna de optimización que proporciona un rendimiento mejorado para ciertas secuencias de operadores (Ej: `$sort + $match`, `$project+$match`, [etc.](#))

# Anexo: GridFS

---

- Forma de almacenar grandes ficheros en MongoDB.
- Divide el fichero en partes (chunks), almacenándolos por separado.
- GridFS almacena la información en dos colecciones:
  - Chunks (almacena los chunks): fs.chunks
  - Files (almacena los metadatos del fichero): fs.files

# Anexo: GridFS

---

- Chunks:

```
{ "_id" : <ObjectId>,
 "files_id" : <ObjectId>,
 "n" : <num>,
 "data" : <binary> }
```

- Donde:

- “\_id”: identificador
- “files\_id”: identificador del padre (en files)
- “n”: número de orden
- “data”: chunk

# Anexo: GridFS

---

- Files:

```
{
 "_id" : <ObjectId>,
 "length" : <num>,
 "chunkSize" : <num>,
 "uploadDate" : <timestamp>,
 "md5" : <hash>,
 "filename" : <string>,
 "contentType" : <string>,
 "aliases" : <string array>,
 "metadata" : <dataObject>,
}
```

# Anexo: GridFS

---

- mongofiles <options> <commands> <filename>
- Donde:
  - <options>: host, puesto, usuario, bd, nombre local del fichero (sobreescribe a <filename>),...
  - <commands>: cargar un fichero, leerlo, borrarlo, buscar, consultar...
  - <filename>: nombre del fichero o identificador
- Versión reducida:
  - mongofiles --local "ruta del fichero en local" -d basededatos put "nombre del fichero en GridFS"
  - mongofiles --local "ruta del fichero en local" -d basededatos get "nombre del fichero en GridFS"

# Anexo: GridFS

---

- Ejemplo

- mongofiles --local "C:/TEMP/Eurovision.mp4" -d prueba put "eurovisionOT"

- db.fs.files.find()

- db.fs.chunks.find({}, {\_id:1, files\_id:1, n:1})

- mongofiles --local "C:/TEMP/recuperado.mp4" -d prueba get "eurovisionOT"