



Acceso a Datos MongoDB

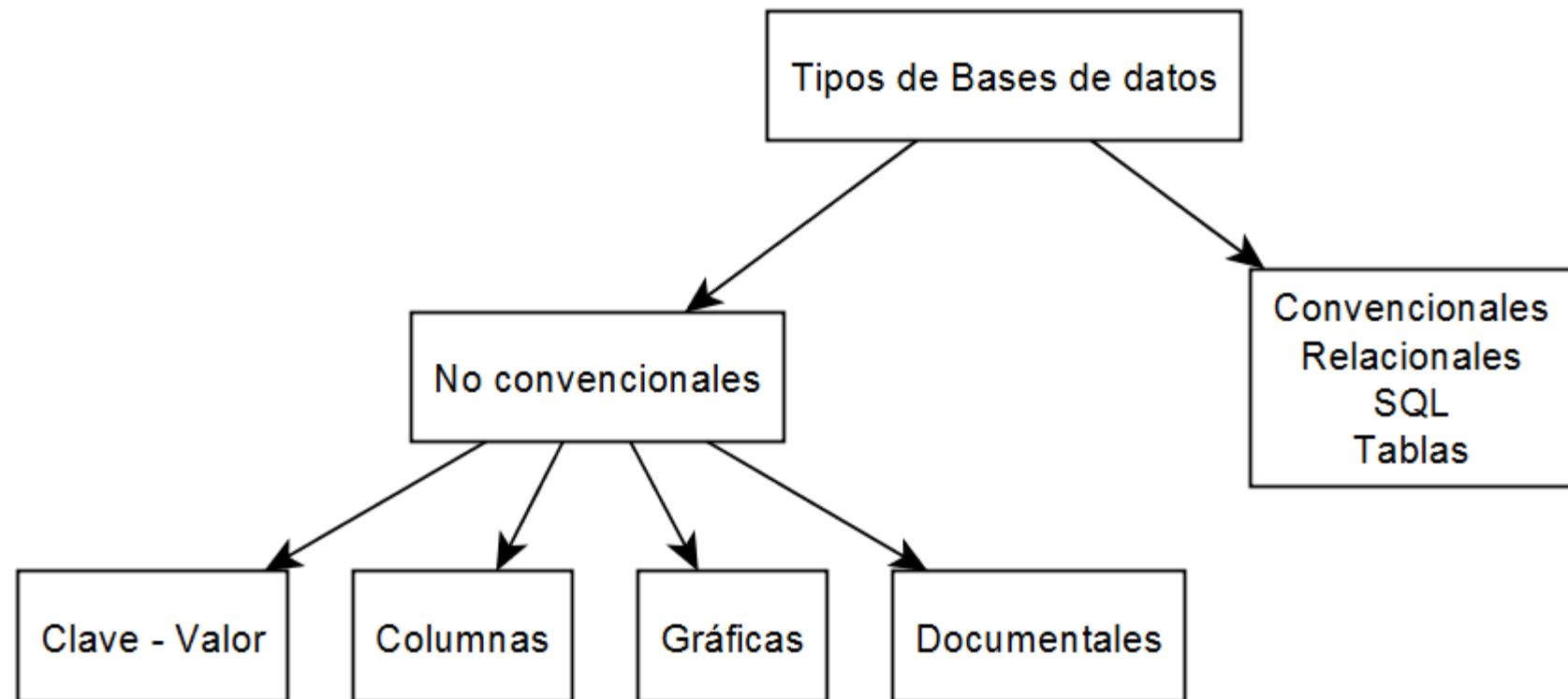
Mongo DB

1. Introducción
2. Operaciones básica con el Shell
3. Operaciones avanzadas con el Shell (Agregación MapReduce)
4. Conexión a PHP y Java



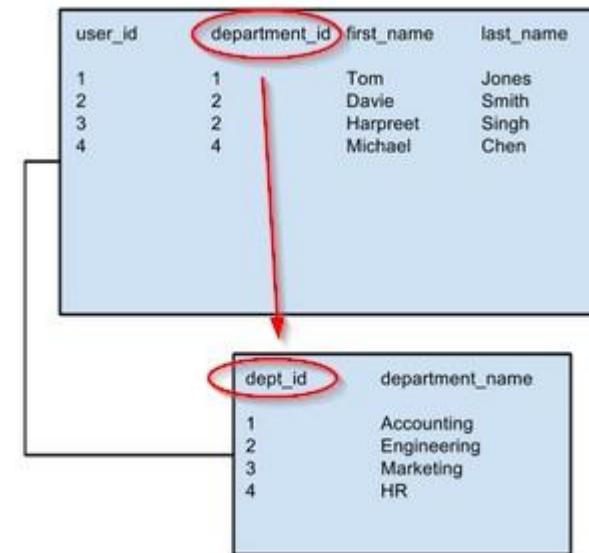
1. Introducción a Mongo Data Base

Introducción a los sistemas de bases de datos documentales



Bases de datos relacionales

- Los datos se almacenan en tablas → Registros → Campos
 - Las tablas pueden **relacionarse** entre sí
- Es necesario definir el modelo de datos (**esquema**) a priori.
- Modelo de **transacciones** ACID (Atomicity, Consistency, Isolation, Durability) maduro
- Resultan difíciles de **escalar**
 - Escalabilidad vertical



Bases de datos clave → valor

Los datos se almacenan en pares clave (key) → valor (value)

Carecen de esquema

El valor puede ser de cualquier tipo (texto, colecciones, imágenes, vídeo...)

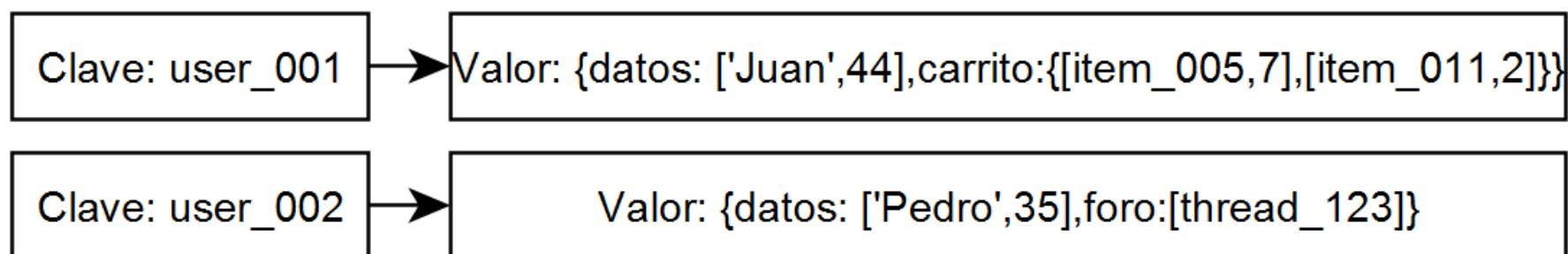
Sólo se pueden consultar los datos a través de las claves, que son únicas

Simples, rápidas y fácilmente escalables

No se pueden realizar consultas sobre el contenido de los valores (son entidades opacas)

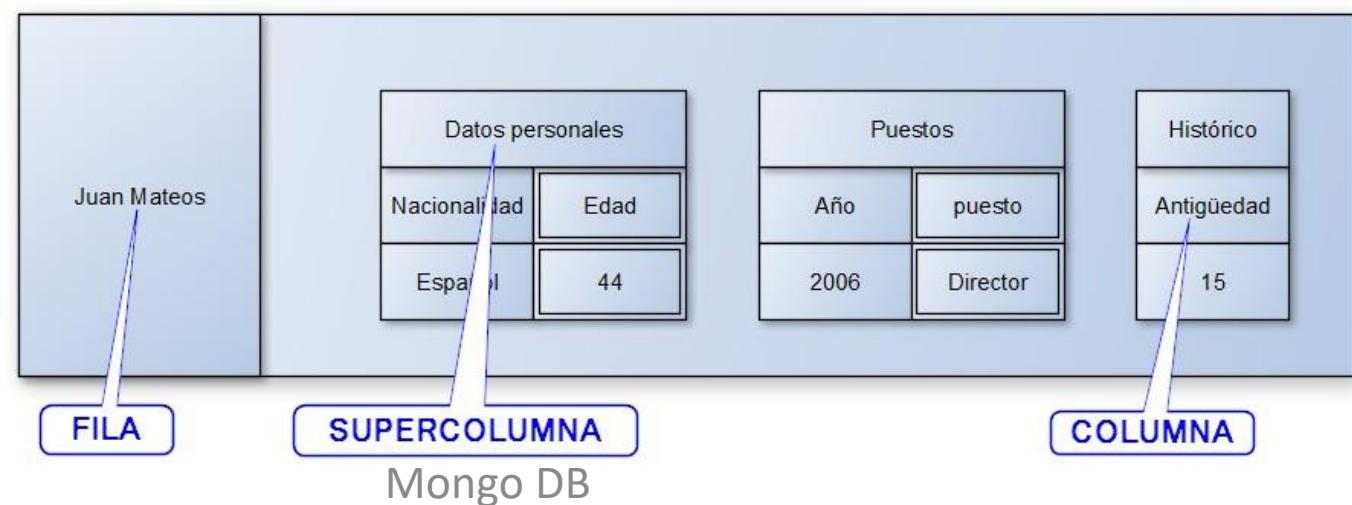
Caso de uso: Información de sesión en aplicaciones web, carritos de la compra...

Ejemplos: **Amazon Dynamo, Riak y Redis**



Bases de datos de columnas 1/2

- Los datos se almacenan en:
 - **Columnas:** Un par clave → valor
 - **Super-columnas:** Combinación de columnas
 - **Familia de columnas:** Combinación de columnas almacenadas en una fila e identificadas por una clave de fila
 - **Familia de super-columnas:** Combinación de columnas y super-columnas almacenadas en una fila e identificadas por una clave de fila
- La clave para acceder a un dato tiene el formato:
 - fila :: [familia de columnas] :: columna

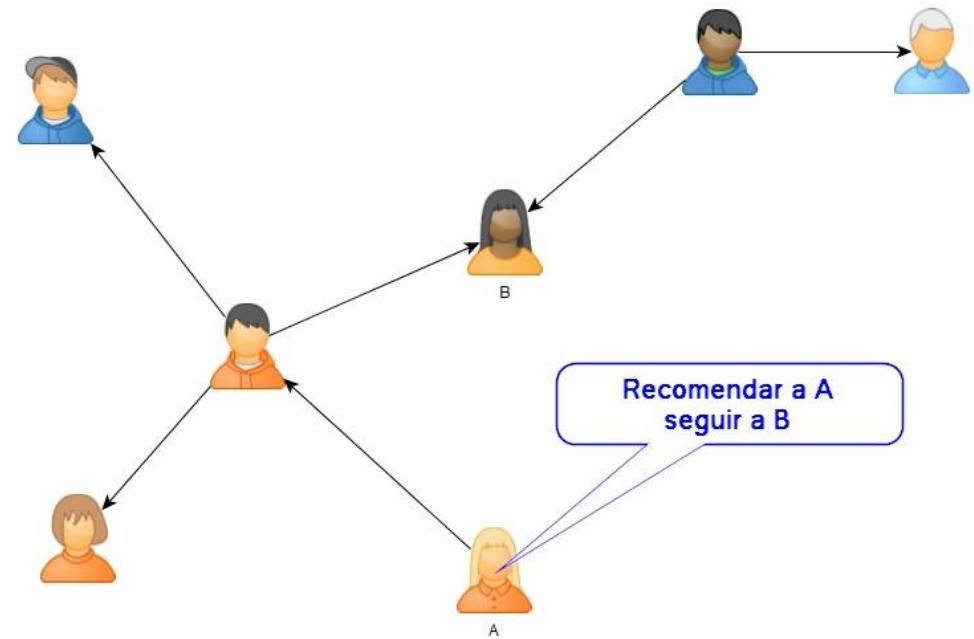


Bases de datos de columnas 2/2

- Escalan muy bien
- Muy rápidas en consultas de agregación (sumas, medias, máximos...)
- Todos los datos de una columna deben tener el mismo formato
 - Esto facilita la compresión
 - Pero resta flexibilidad
- **Caso de uso:** Contadores, registro de eventos...
- Ejemplo: **Cassandra**

Bases de datos de gráficos

- Los datos se almacenan en **nodos**, pero también se almacenan las relaciones entre ellos (estas relaciones se denominan **bordes**).
- Al no tener que calcular los bordes al efectuar consultas, son muy rápidas para obtener relaciones entre los datos
 - Por ejemplo: Los amigos de mis amigos de Facebook
- Su escalabilidad puede resultar compleja
- **Casos de uso:** Motores de recomendaciones en función de las preferencias de mis amigos o de mi historial de compras
- Ejemplo: **Neo4j**



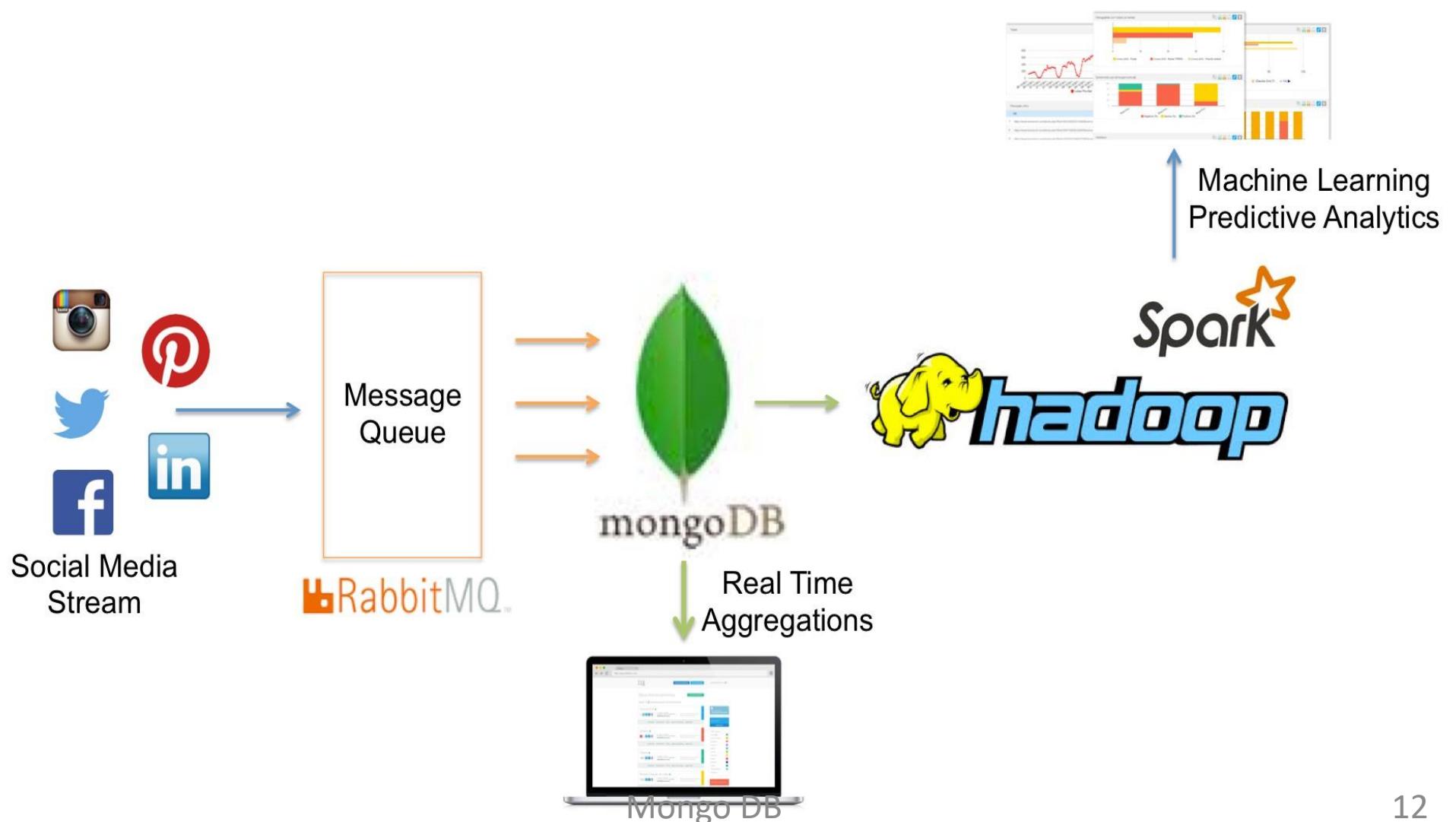
Bases de datos documentales

- Los datos se almacenan en Bases de Datos → Colecciones → Documentos
 - Los documentos pueden ser de cualquier tipo: textos, vídeos...
 - Y pueden tener cualquier estructura, incluso dentro de una misma colección
 - Los documentos pueden contener otros documentos incrustados (o referenciados)
- Permiten realizar búsquedas muy rápidas
- Casos de uso: **Analíticas web**
- Ejemplos: CouchDB y MongoDB

La historia de MongoDB

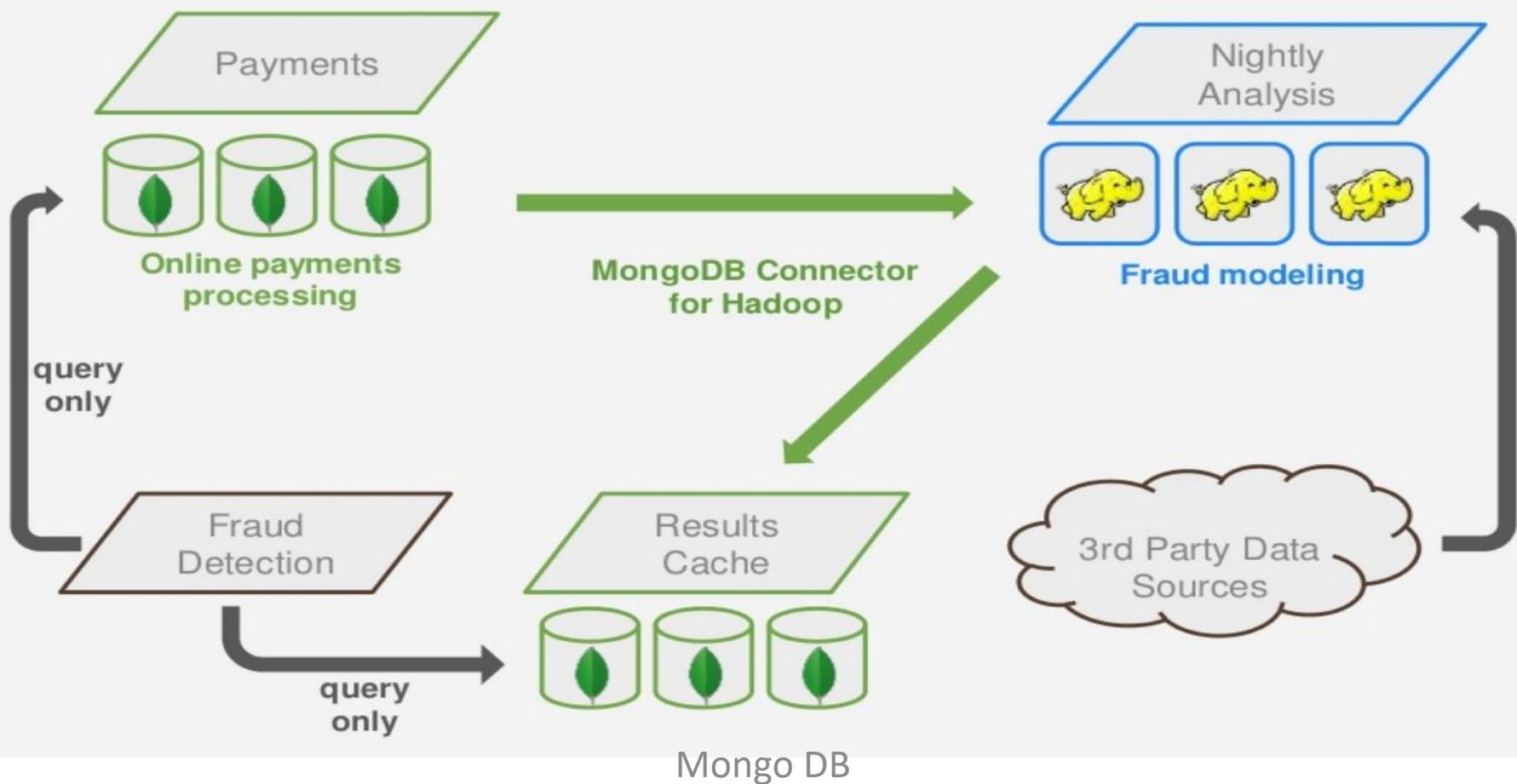
- 2007: 10gen (DoubleClick) inicia el desarrollo de MongoDB.
 - El nombre proviene de **Humongous** (enorme).
- 2009: MongoDb se empieza a distribuir con licencia de código abierto.
- 2011: Se lanza la versión 1.4, que es la primera considerada suficientemente estable para escenarios reales de explotación
- Existen básicamente 2 versiones disponibles:
 - Community
 - Enterprise (Compass → Visualización de datos mediante GUI)
- Última versión estable: 4.0.2 (29 de agosto de)

El lugar de MongoDB dentro del ecosistema Big Data 1/3



El lugar de MongoDB dentro del ecosistema Big Data 2/3

Fraud Detection



El lugar de MongoDB dentro del ecosistema Big Data 3/3

- Actores principales
 - **Hadoop**: Framework para aplicaciones distribuidas que almacena los datos en disco. Diseñado para manejar cantidades ingentes de información.
 - **Spark**: Framework de ejecución en clúster para analizar datos almacenados en memoria.
 - **Kafka**: Sistema que permite unificar mensajes provenientes de fuentes diferentes (streaming).
 - **MongoDB**

El teorema CAP (conjetura de Brewer)

Escenario	Prioridad	Descripción
Sin particiones	CA	El sistema es disponible y con una alta consistencia
Particionado con la mayoría de los nodos conectados	AP	Se elige un nuevo servidor primario y las escrituras no sincronizadas procedentes del anterior son ignoradas
Particionado con la mayoría de los nodos no conectados	CP	Sólo se permiten operaciones de lectura

C: Consistencia A: Disponibilidad P: Tolerancia

Compatibilidad ACID

- Son parámetros para caracterizar las transacciones:
 - **Atomicidad:** Se ejecutan todos los pasos o ninguno, pero la transacción nunca se queda a medias.
 - **Consistencia** (integridad): La transacción sólo se ejecuta si lleva los datos de un estado estable a otro también estable, es decir, entre estados que no violan las reglas de integridad de los datos.
 - **Aislamiento:** Dos transacciones ejecutadas en paralelo tienen que producir el mismo resultado que ejecutadas en serie; es decir, una transacción ejecutada en paralelo con otra no puede depender de ella.
 - **Durabilidad:** Una vez ejecutada la transacción, los resultados persistirán aunque el sistema sufra un fallo.

¿Por qué usar MongoDB?

- Manejo de cantidades ingentes de información (**terabytes y petabytes**) que varían rápidamente tanto en su contenido como en su estructura.
- Aplicaciones cuya base de usuarios puede crecer rápidamente hasta el orden de los millones.
- Desarrollo rápido (Agile) gracias a su paradigma de esquema flexible, que permite introducir nuevas características sin tiempos de parada.
- Reducción de costes por el uso de software Open Source.
- Programación orientada a objetos en lugar de un lenguaje de consulta.
- Escalabilidad horizontal en lugar de vertical.
- Alto rendimiento y alta disponibilidad.
- Indexación geoespacial.
- Integración con Bigdata Hadoop
- Creado para la nube (Azure, AWS...)

¿Quién usa MongoDB?



“Without MongoDB, it would've taken a much bigger team to get the app live so quickly.”

— Prashanth Kokati, Expedia

[Tweet](#)



“As we work with our user base to figure out killer features, rapid innovation cycles with MongoDB are a real benefit.”

— Luke Kolin, The Weather Channel

[Tweet](#)



“The flexibility of MongoDB has put a lot of power in the hands of our developers.”

— Brian Massey, Under Armour

[Tweet](#)



Mongo DB

¿Cuándo usar MongoDB? ¿y cuándo no?

- MongoDB no pretende ser una solución óptima para todos los casos de uso, pero se adapta muy bien a:
 - Aplicaciones móviles
 - Gestores de contenidos
 - Análisis de datos en tiempo real
 - IoT
- Los siguientes factores pueden desaconsejar su uso:
 - Carece de **JOINS hasta la versión 3.X.X**, por lo que las consultas que requieran relacionar datos de documentos distintos pueden resultar lentas.
 - Las transacciones a nivel de documento se han añadido en la versión 4 (junio de), por lo que no se consideran aún maduras. A nivel de sarding están previstas para la versión 4.2.
 - Necesita integrarse con otros sistemas basados en SQL/OLAP/OLTP
 - OLAP: On line analytical processing (lectura)
 - OLTP: On line transactional processing (escritura)

¿Cómo usar MongoDB? BSON 1/2

- BSON (binary JSON): <http://bsonspec.org>
 - Basado en JSON, diseñado para facilitar el intercambio de información entre aplicaciones y clientes web (navegadores)
 - Optimizado para lograr mayor velocidad que JSON (a costa de requerir más espacio de almacenamiento) y permitir una fácil conversión al sistema nativo de datos de muchos lenguajes de programación (Python, Ruby, PHP, C...)
- Clave: valor (tipo)

```
{"nombre": "Antonio",
"edad": 44,
"emails": ["juanma@gmail.com", "juanma@educa.madrid.org"\]}
```

```
Bson:
\x16\x00\x00\x00          // total document size
\x02                      // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00    // field value (size of value, value, null terminator)
\x00                      // 0x00 = type EOO ('end of object')
```

¿Cómo usar MongoDB? BSON 2/2

- Algunos de los tipos de datos BSON
 - Cadena: {"nombre": "Antonio"}
 - Entero (32 y 64 bits): {"edad": 44}
 - Real (IEEE 754)
 - Fecha (milisegundos desde el 1 de enero de 1970 – UNIX Epoch)
 - Array de bytes (datos binarios)
 - Booleano (true o false)
 - Null
 - Objeto BSON
 - Array BSON
 - Código JavaScript
 - Expresiones regulares

¿Cómo usar MongoDB? Motor de base de datos

- MMAP (Obsoleto en MongoDB 4): Memory Mapped Storage Engine
 - Escritura en disco cada minuto
 - Escritura en el diario (journal) cada 100ms
- WiredTiger (opción recomendada)
 - **Concurrencia** a nivel de documento: Varios clientes pueden escribir simultáneamente documentos distintos de una misma colección
 - Compresión de datos
 - Soporte para **transacciones**

¿Cómo usar MongoDB? Índices

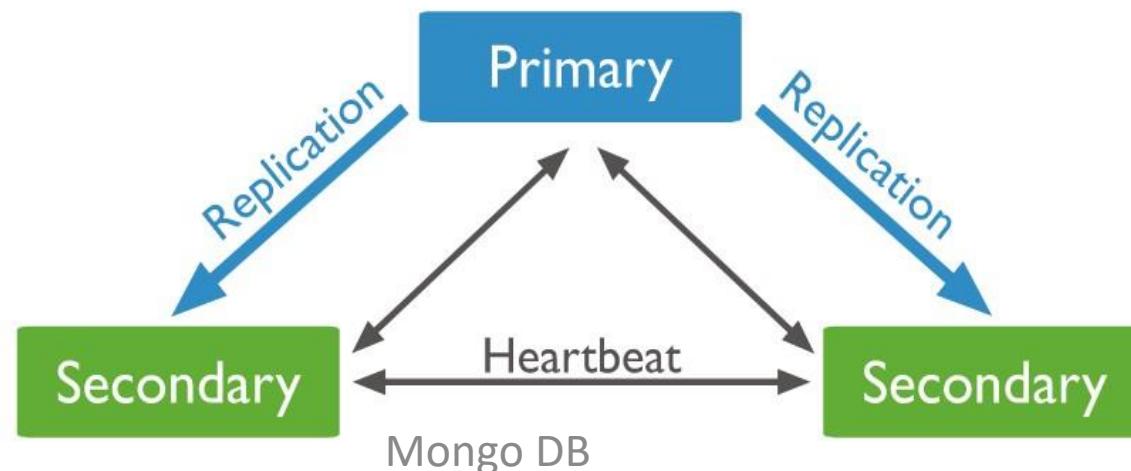
- Sin los índices, las consultas tendrían que “leer” todo el contenido de los documentos para localizar datos
 - Sistema Dewey de las bibliotecas
- Todos los documentos de MongoDB se indexan automáticamente por la clave `_id`, que asegura su unicidad.
- MongoDB admite:
 - Índices sobre documentos incrustados
 - Índices compuestos (usar varias claves para el índice: nombre + apellido)
 - Índices geoespaciales: ¿gasolineras más próximas a mi ubicación?
- Cada colección puede tener como máximo 64 índices
- Los índices aceleran las búsquedas pero relentizan las inserciones y borrados

¿Cómo usar MongoDB? GridFS

- El tamaño máximo de un documento en MongoDB es 16MB, pero con GridFS no existe límite porque la información (por ejemplo un clip de sonido) se divide en:
 - **Metadata:** que se almacena en la colección **files**
 - **Contenido:** que se fragmenta y se almacena en la colección **chunks**.

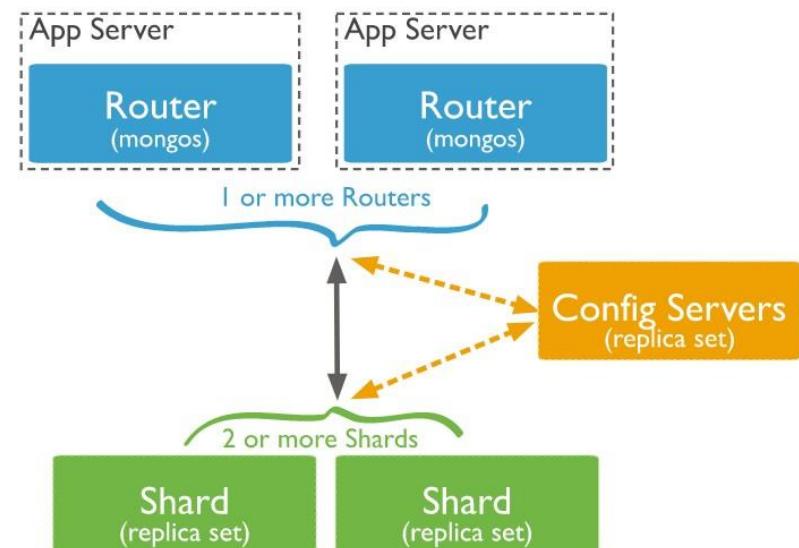
¿Cómo usar MongoDB? Replicado

- Para el replicado inicialmente se utilizaba un sistema master-slave, pero ahora está obsoleto en favor del sistema replica-set:
 - Un servidor principal procesa todas las operaciones de escritura y las consigna en el **oplog** (registro de operaciones).
 - El oplog se replica en los servidores secundarios, que lo utilizan para mantener una copia actualizada del servidor principal.
 - Si el servidor principal falla, los secundarios realizan una elección para elegir un nuevo principal, pero sólo si el número de secundarios disponibles (saludables) es mayoría (la mitad más uno).
 - Puede haber un nodo **árbitro**, que no mantiene una copia de los datos, sino que sólo responde a los heartbeat y desempata en caso de que el número de secundarios sea par.



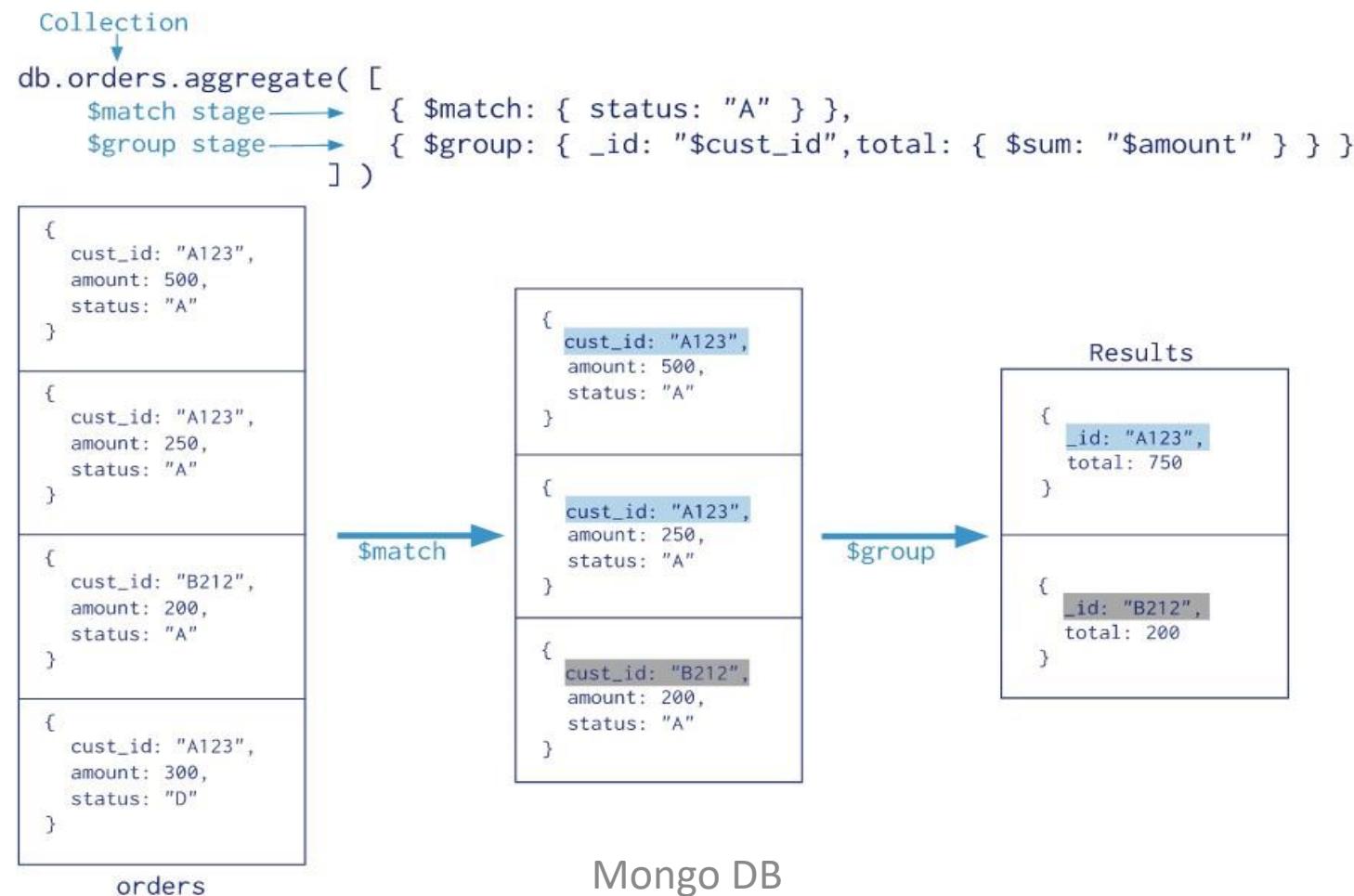
¿Cómo usar MongoDB? Sharding

- Tipos de escalado:
 - **Vertical:** Es el más habitual en los sistemas de bases de datos SQL. Para ampliar el sistema hay que actualizar el servidor a un modelo más potente (mejor procesador, más memoria, discos duros de más capacidad...). Es un modelo de crecimiento limitado.
 - **Horizontal:** Los datos y la carga de trabajo se reparte entre varios servidores. Es el sistema que utiliza MongoDB.
 - **Mongos:** Enrutan las peticiones
 - **Shards:** Contienen las colecciones divididas.
 - **Config servers:** Conocen la división en chunks de las colecciones y pasan la información a los Mongos para que sepan dónde dirigir cada petición.



¿Cómo usar MongoDB? Pipeline aggregation

- Es un sistema optimizado para generar datos agregados (sumas, máximos...) de una forma sencilla y rápida

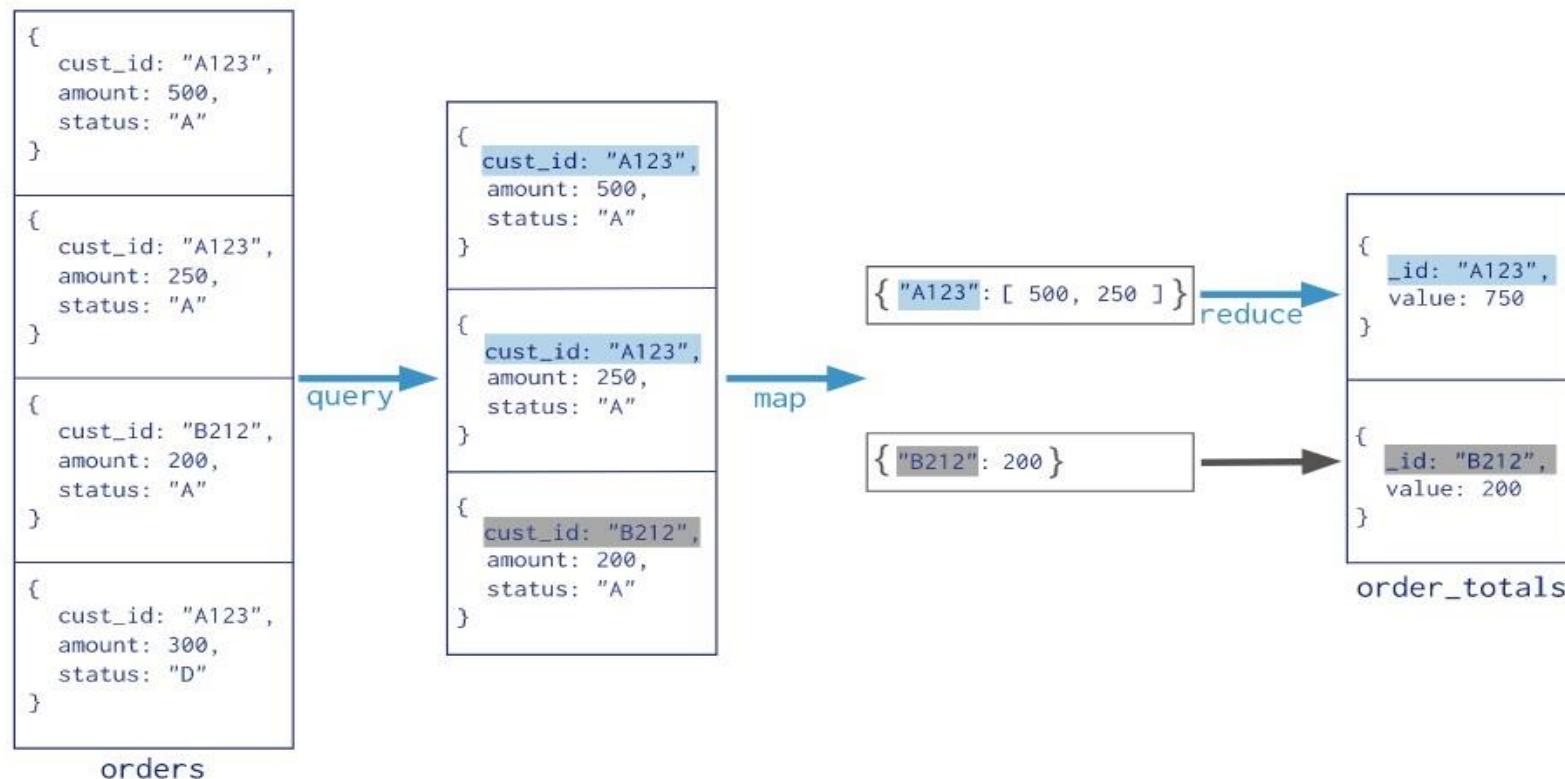


¿Cómo usar MongoDB? Map and Reduce 1/2

- Es un sistema de agregación más complejo, pero también más lento y complicado de utilizar.
 - La función **map** genera un documento intermedio con los datos de interés...
 - que son agregados por la función **reduce**.

¿Cómo usar MongoDB? Map and Reduce 2/2

```
Collection  
↓  
db.orders.mapReduce(  
    map → function() { emit( this.cust_id, this.amount ); },  
    reduce → function(key, values) { return Array.sum( values ) },  
    query → { query: { status: "A" } },  
    output → { out: "order_totals" }  
)
```



Instalación de MongoDB en Windows

- La versión 4 sólo funciona en sistemas de 64 bits
- Windows 7/Server 2008 R2 o posterior
- Descargar e instalar el archivo msi
- Añadir al path (C:\Program Files\MongoDB\Server\4.0\bin) a las “Variables de entorno del sistema”
- Arrancar MongoDB escribiendo **mongod** en la línea de comandos (cmd)
- Arrancar el shell de MongoDB abriendo otra línea de comandos (cmd) y escribiendo **mongo**
- Para arrancar y detener el servicio MongoDB, desde una línea de comandos (cmd) ejecutada como administrador:

```
net start mongodb
```

```
net stop mongodb
```
- Revisar el archivo de configuración en C:\Program Files\MongoDB\Server\4.0\bin\mongod.cfg



2.

Operaciones básicas del Shell

Operaciones básicas con el shell

- Mostrar las bases de datos
 - **show dbs**
 - Por defecto tenemos 3 bases de datos (admin, config y local) que utiliza internamente MongoDB.
- Activar/crear una base de datos
 - **use centro**: Al activar una base de datos se crea si no existía previamente. MongoDB es sensible al uso de mayúsculas y minúsculas. Si mostramos ahora las bases de datos no aparecerá aun **centro** porquhele no tiene contenido.
- Obtener ayuda
 - **help**: muestra ayuda general
 - **db.help()**: muestra ayuda sobre los métodos relacionados con las bases de datos
 - **db.<colección>.help()**: muestra ayuda sobre los métodos relacionados con las colecciones
- Averiguar qué base de datos está activa
 - **db**

Operaciones básicas con el shell

- **Insertar documentos en una colección**
 - `db.personas.insertOne({Tipo: "Profesor", Nombre: "Juan Félix"})`
 - Si la colección no existe, se crea automáticamente.
 - El espacio de nombres de la colección
(nombre_base_de_dados.nombre_colección) no puede superar los 120 caracteres.
 - No está permitido crear una colección con el nombre **system**
 - Cada documento recibe automáticamente un `_id` compuesto por:
 - 4bytes: Segundos desde la época UNIX
 - 5bytes: Aleatorio (primeros 3 bytes del MD5 maquina[3] seguido de PID[2])
 - 3 byte: Contador incremental
- **Mostrar todas las colecciones de la base de datos activa**
 - `show collections`

Operaciones básicas con el shell

- Insertar varios documentos en una colección

```
db.personas.insertMany([
```

```
{Tipo: "Profesor", Nombre: "Ana", Apellidos: "Palma", Asignaturas: ["Matemáticas", "Física", "Química"]},
```

```
{Tipo: "Alumno", Nombre: "Felipe", Apellidos: "Martín", Calificaciones: {"Matemáticas": 8, "Física": 5.5}},
```

```
{Tipo: "Profesor", Nombre: "Irene", Apellidos: "Pérez", Asignaturas: ["Inglés", "Física", "Dibujo"]},
```

```
{Tipo: "Alumno", Nombre: "Raquel", Apellidos: "López", Calificaciones: {"Matemáticas": 9, "Física": 7.5, "Inglés": 6}},
```

```
{Tipo: "Administrativo", Nombre: "Luis", Nominas:[{Mes: "Octubre", "Cantidad": 1850.90},{Mes: "Noviembre", "Cantidad": 1600.50}]},
```

```
{Tipo: "Administrativo", Nombre: "Marisa", Nominas:[{Mes: "Octubre", "Cantidad": 1420.30},{Mes: "Noviembre", "Cantidad": 1850.90}]}
```

```
])
```

Operaciones básicas con el shell

- **Eliminar la base de datos activa**
 - `db.dropDatabase()`
- **Eliminar una colección**
 - `db.<colección>.drop()`
- **Renombrar una base de datos (fuera del shell)**
 - Crear una carpeta en la que almacenar un volcado (`c:\dump`) y colocarse en ella
 - `mongodump --db <base_de_datos>`
 - `mongorestore --dir c:\dump --nsInclude 'base_de_datos' --nsFrom 'centro.*' --nsTo 'nuevocentro.*'`
- **Renombrar una colección**
 - `db.<colección>.renameCollection('nuevoNombre', [dropTarget=true/false])`

Operaciones básicas con el shell

- **Buscar documentos**
 - `db.personas.find()`
- **find(consulta,proyeccion)**
 - **Consulta**
 - `db.personas.find({Tipo: "Profesor"})`
 - **Proyección**
 - `db.personas.find({Tipo: "Alumno"},{_id:0})`
 - **Consulta en array**
 - `db.personas.find({Asignaturas: "Física"},{Nombre: 1})`
 - **No se pueden combinar proyecciones de exclusión e inclusión**
 - ~~`db.personas.find({Asignaturas: "Física"},{Nombre: 1, Apellidos: 0})`~~
 - **Salvo la de exclusión de _id**
 - `db.personas.find({Asignaturas: "Física"},{Nombre: 1, _id: 0})`

Operaciones básicas con el shell

- **Buscar en documentos incrustados (sintaxis de punto; usar obligatoriamente comillas dobles)**
 - `db.personas.find({"Calificaciones.Matemáticas": 8})`
- **Si no usamos la sintaxis de punto, el subdocumento tiene que coincidir exactamente, incluso en el orden de los campos**
 - `db.personas.find({"Calificaciones": {"Matemáticas": 8}})`
 - `db.personas.find({"Calificaciones": {"Matemáticas": 8, "Física": 5.5}})`
- **Buscar en arrays de documentos incrustados**
 - Sin sintaxis de punto, la coincidencia debe ser exacta, incluso en el orden de los campos
 - `db.personas.find({"Nominas": {"Mes": "Octubre", "Cantidad": 1850.90}})`
 - `db.personas.find({"Nominas.Cantidad": 1850.90})`
 - También podemos buscar por el índice del subdocumento
 - `db.personas.find({"Nominas.0.Cantidad": 1850.90})`

Operaciones básicas con el shell

- **Ordenación: 1=Ascendente / 0=Descendente**
 - `db.personas.find().sort({"Nombre":1})`
- **limit**
 - `db.personas.find().limit(2)`
- **skip**
 - `db.personas.find().skip(2)`
- **Combinación de comandos**
 - `db.personas.find().sort({"Nombre": 1}).limit(1).skip(1)`

Operaciones básicas con el shell

- **Proyección \$:** Devuelve sólo el primer elemento de un array que esté filtrado en la consulta de find() y que cumpla la consulta
 - `db.personas.find({"Asignaturas": "Física"}, {"Asignaturas.$":1})`
- **Proyección \$elemMatch:** Devuelve sólo el primer elemento de un array que cumpla la condición especificada en el propio \$elemMatch, aunque no esté filtrado en la consulta
 - `db.personas.find({Tipo: "Administrativo"}, {Nombre:1, Nominas: {$elemMatch:{"Cantidad": {$gt:1400}}}})`
- **Proyección \$slice:** Devuelve el número de elementos especificado de un array (si es negativo, empezando por el final)
 - `db.personas.find({Tipo: "Administrativo"}, {Nominas:{$slice: -1}})`

Operaciones básicas con el shell

- Operadores de comparación
 - \$eq
 - \$gt
 - \$gte
 - \$in
 - \$lt
 - \$lte
 - \$ne
 - \$nin
- db.personas.find({Tipo:{\$in:["Profesor","Alumno"]}})
- db.personas.find({"Nominas.Cantidad":{\$gte:1850.9,\$lt:2000}})

Operaciones básicas con el shell

- **Operadores lógicos**
 - \$and
 - \$or
 - \$nor
 - \$not
- **db.personas.find({\$and:[{Tipo:"Alumno"},{Apellidos: "López"}]})**
- **db.personas.find({\$nor:[{Tipo: "Profesor"},{Tipo: "Alumno"}]})**

Operaciones básicas con el shell

- Operadores de array
 - \$all
 - \$elemMatch: Obliga a que se cumplan todas las condiciones en un mismo elemento del array
 - \$size
- db.personas.find({Asignaturas:{\$all:["Matemáticas","Física"]}})
- db.personas.find({Asignaturas:{\$size: 3}})
- db.personas.find({Asignaturas: {\$elemMatch: {\$eq:"Matemáticas"}}})
- db.personas.find({"Nominas.Mes": "Octubre","Nominas.Cantidad": 1850.9})
- db.personas.find({Nominas: {\$elemMatch:{Mes: "Octubre", "Cantidad": 1850.9}}})

Operaciones básicas con el shell

- El operador **\$exists**: boolean permite consultar los documentos en los que existe o no el campo especificado
 - `db.personas.find({"Calificaciones.Matemáticas":{$exists: true}})`
 - `db.personas.find({"Calificaciones.Matemáticas":{$exists: false}})`
- Comandos básicos de agregación: **count()** y **distinct()**
 - `db.personas.count()`
 - `db.personas.find({"Tipo": "Profesor"}).count()`
 - `db.personas.distinct("Tipo")`
 - `db.personas.distinct("Nominas.Mes")`
 - `db.personas.distinct("Calificaciones")`

Operaciones básicas con el shell

- **Expresiones regulares**
 - `db.personas.find({"Nombre": {$regex: '^J'}})`
 - `db.personas.find({"Nombre": {$regex: 'j'}})`
 - **No sensible a mayúsculas/minúsculas con la opción i**
 - `db.personas.find({"Nombre": {$regex: 'j',$options: 'i'}})`
 - **Combinar varios criterios en la consulta**
 - `db.personas.find({"Tipo":{$regex: '^A',$ne: "Alumno"}})`
 - `db.personas.find({"Nombre":{$regex: 'i'}}, {"Nombre": 1})`

Operaciones básicas con el shell

- Operaciones de actualización
 - `updateOne({filtro}, {actualización}, {opciones})`
 - `updateMany({filtro}, {actualización}, {opciones})`
 - `replaceOne({filtro}, {actualización}, {opciones})`
 - Campos: La `{actualización}` puede contener los siguientes operadores
 - `$currentDate`
 - `$inc`
 - `$min`: Elige el mínimo entre el valor actual y el especificado
 - `$max`
 - `$mul`: Multiplica por el valor indicado
 - `$rename`
 - `$set`
 - `$setOnInsert`: Inserta el campo sólo si no existe ningún documento que cumpla el criterio del filtro y upsert es true (+2 diapositivas)
 - `$unset`

Operaciones básicas con el shell

- Actualizar el valor de un campo
 - `db.personas.updateOne({Nombre: "Juan Félix"},{$set:{"Tipo":"Alumno"})`
- Actualizar el valor de un subdocumento con notación de punto
 - `db.personas.updateOne({Nombre: "Felipe"},{$set: {"Calificaciones.Matemáticas": 7.5}})`
- Actualizar el valor de un array utilizando `arrayFilters[]`
 - El identificador del elemento (en este caso "materia" no puede empezar con mayúscula)
 - `db.personas.updateOne({Nombre: "Ana"},{$set:{"Asignaturas.$[materia]":"Dibujo"}}, {arrayFilters:[{"materia":"Física"}]})`
- Actualizar el valor de un campo en un array de sub-documentos
 - `db.personas.updateOne({"Nombre": "Marisa"},{$set:{"Nominas.$[mensualidad].Cantidad": 1500}}, {arrayFilters: [{"mensualidad.Mes":"Octubre"}]})`

Operaciones básicas con el shell

- Al actualizar un campo que no existe, se crea ese campo en el documento
 - `db.personas.updateOne({Nombre: "Juan Félix"},{$set:{"Apellidos": "Mateos"})}`
 - `db.personas.updateOne({Nombre: "Juan Félix"},{$set:{"Asignaturas": ["Inglés"]}})`
- El operador \$exists puede utilizarse para actualizar un campo sólo si existe
 - `db.personas.updateOne({"Nombre": "Juan Félix","Apellidos":{$exists: true}},{$set:{"Apellidos": "Barrado"})}`
- Por defecto, al actualizar un campo, si no existe ningún documento que cumpla el criterio del filtro, no ocurre nada.
 - `db.personas.updateOne({"Nombre": "Almudena"},{$set:{"Tipo": "Profesor"})}`
 - Pero con la opción upsert: true se creará un documento nuevo si ninguno de los existentes cumple el filtro
 - `db.personas.updateOne({"Nombre": "Almudena"},{$set:{"Tipo": "Profesor"}}, {upsert: true})`

Operaciones básicas con el shell

- Renombrar campos con el operador \$rename
 - db.personas.updateMany({},{\$rename:{'Apellidos': "Primer apellido"}})
- Multiplicar con \$mul
 - db.personas.updateMany({"Tipo":"Administrativo"},{\$mul:{'Nominas.\$[mensualidad].Cantidad':1.05}},{\$arrayFilters:[{"mensualidad.Mes": "Noviembre"}]})
 - db.personas.updateMany({"Tipo":"Administrativo"},{\$mul:{'Nominas.\$[mensualidad].Cantidad':1.05}},{\$arrayFilters:[{"mensualidad.Mes":{\$not:{\$eq:"Noviembre"}}}]}))
- Actualizar todos los elementos de un array con el operador posicional todos \$[]
 - db.personas.updateMany({"Tipo":"Administrativo"},{\$set:{'Nominas.\$[].Cantidad': 1000}})

Operaciones básicas con el shell

- **Borrar un campo con \$unset**
 - `db.personas.updateOne({Nombre: "Juan Félix"},{$unset:{"Primer apellido":""}})`
- **Operadores de actualización de arrays**
 - `$`
 - `$[]`
 - `$[<identificador>]`
 - **\$addToSet: Añade un elemento al array sólo si no existe ya en él**
 - **\$pop: Elimina el primer o último elemento de un array**
 - **\$pull: Elimina todos los elementos que cumplen un criterio**
 - **\$push: Añade un elemento a un array**
 - **\$pullAll: Elimina todos los elementos que coincidan con los especificados en un array**

Operaciones básicas con el shell

- Añadir un elemento a un array si no existe en él
 - `db.personas.updateMany({"Tipo": "Profesor"},{$addToSet:{"Asignaturas": "Inglés"})}`
- Añadir un elemento a un array independientemente de que ya exista uno igual o no
 - `db.personas.updateMany({Tipo: "Profesor"},{$push:{"Asignaturas": "Inglés"})}`
- Eliminar el primer (-1) o el último (1) elemento de un array con \$pop
 - `db.personas.updateMany({"Tipo": "Administrativo"},{$pop:{Nominas: 1}})`
- Eliminar elementos de un array
 - `db.personas.updateMany({Tipo: "Profesor"},{$pull: {"Asignaturas": "Física"})}`
 - `db.personas.updateMany({Tipo: "Profesor"},{$pullAll:{"Asignaturas": ["Inglés", "Dibujo"]})}`

Operaciones básicas con el shell

- Actualizaciones atómicas
 - El problema ABA
 - Nuestro objetivo es incrementar un contador y, a continuación, leer su nuevo valor
 - Supongamos que hacemos lo siguiente
 - 1) Incrementamos el contador con updateOne
 - 2) Leemos el contador con findOne
 - Supongamos ahora que entre 1 y 2, otro usuario incrementa el contador ¿Qué valor leeremos si el contador inicial era 1? 3
 - Para evitar esto disponemos de findAndModify(), que es capaz de actualizar y leer el valor anterior o el actual de forma atómica

```
db.collection.findAndModify({  
    query: <documento>,  
    remove: <booleano>, → Indica si queremos eliminar el documento que cumple la consulta  
    update: <documento>, → Documento de actualización con $set, $inc, $mul...  
    new: <booleano>,  
    fields: <documento>, → Los campos que devuelve  
    upsert: <booleano>  
});
```

Operaciones básicas con el shell

```
function obtenerSiguiente(identificadorContador) {  
    return db.contadores.findAndModify(  
    {  
        query: {_id : identificadorContador},  
        update: {$inc : {contador: 1}},  
        upsert: true,  
        fields:{contador:1, _id:0},  
        new: true  
    }  
    ).contador  
}
```

obtenerSiguiente("contadorClientes")

obtenerSiguiente("contadorProveedores")

Operaciones básicas con el shell

- **Métodos para eliminar colecciones**
 - **deleteOne()**
 - **deleteMany()**
 - Si ejecutamos `db.personas.deleteMany({})` se borrarán todas las colecciones pero no los índices

Operaciones básicas con el shell

- Llegados a este punto vamos a borrar toda nuestra colección y a rellenarla de nuevo para seguir con los ejemplos:

```
db.personas.drop()
```

```
db.personas.insertMany([
```

```
  {Tipo: "Profesor", Nombre: "Juan Félix"},
```

```
  {Tipo: "Profesor", Nombre: "Ana", Apellidos: "Palma", Asignaturas:  
   ["Matemáticas", "Física", "Química"]},
```

```
  {Tipo: "Alumno", Nombre: "Felipe", Apellidos: "Martín", Calificaciones: {"Matemáticas": 8, "Física":  
   5.5}},
```

```
  {Tipo: "Profesor", Nombre: "Irene", Apellidos: "Pérez", Asignaturas: ["Inglés", "Física", "Dibujo"]},
```

```
  {Tipo: "Alumno", Nombre: "Raquel", Apellidos: "López", Calificaciones: {"Matemáticas": 9, "Física":  
   7.5, "Inglés": 6}},
```

```
  {Tipo: "Administrativo", Nombre: "Luis", Nominas:[{Mes: "Octubre", "Cantidad": 1850.90},{Mes:  
   "Noviembre", "Cantidad": 1600.50}]},
```

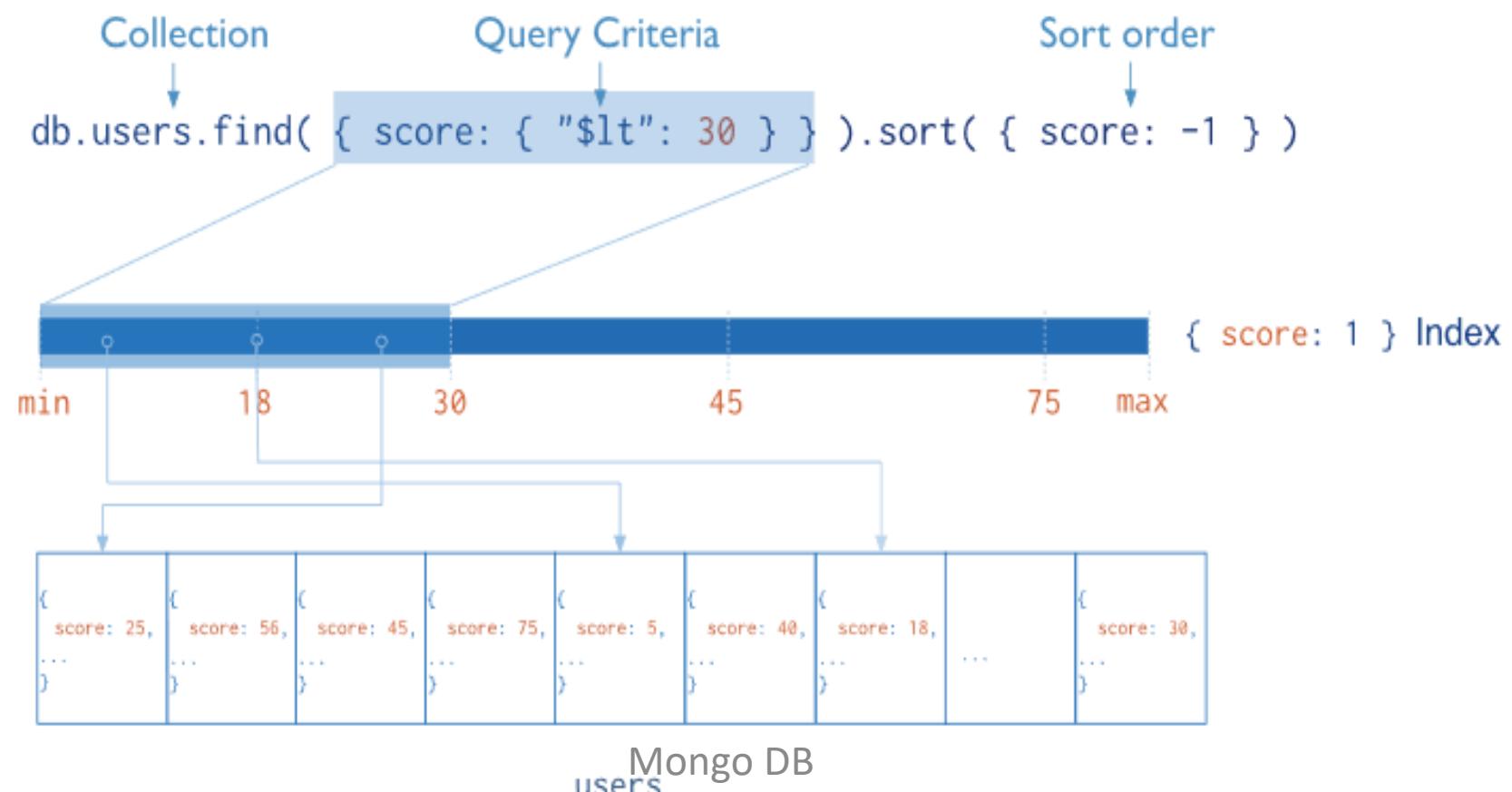
```
  {Tipo: "Administrativo", Nombre: "Marisa", Nominas:[{Mes: "Octubre", "Cantidad": 1420.30},{Mes:  
   "Noviembre", "Cantidad": 1850.90}]}  
])
```

Operaciones básicas con el shell

- Los campos de tipo fecha contienen el número de milisegundos desde la época Unix (positivos o negativos, si son fechas anteriores a la época Unix).
- Para insertar un campo de tipo Date podemos usar los métodos JavaScript:
 - `new Date(número de milisegundos)`
 - `new Date("AAAA-MM-DD")`
 - `new Date("AAAA-MM-DDHH:MM:SS")`
- `db.personas.updateOne({"Nombre":"Juan Félix"},{$set:{"Fecha nacimiento":new Date()}})`
- `db.personas.updateOne({"Nombre":"Juan Félix"},{$set:{"Fecha nacimiento":new Date(24*60*60*1000)}})`
- `db.personas.updateOne({"Nombre":"Juan Félix"},{$set:{"Fecha nacimiento":new Date("1973-10-05")}})`

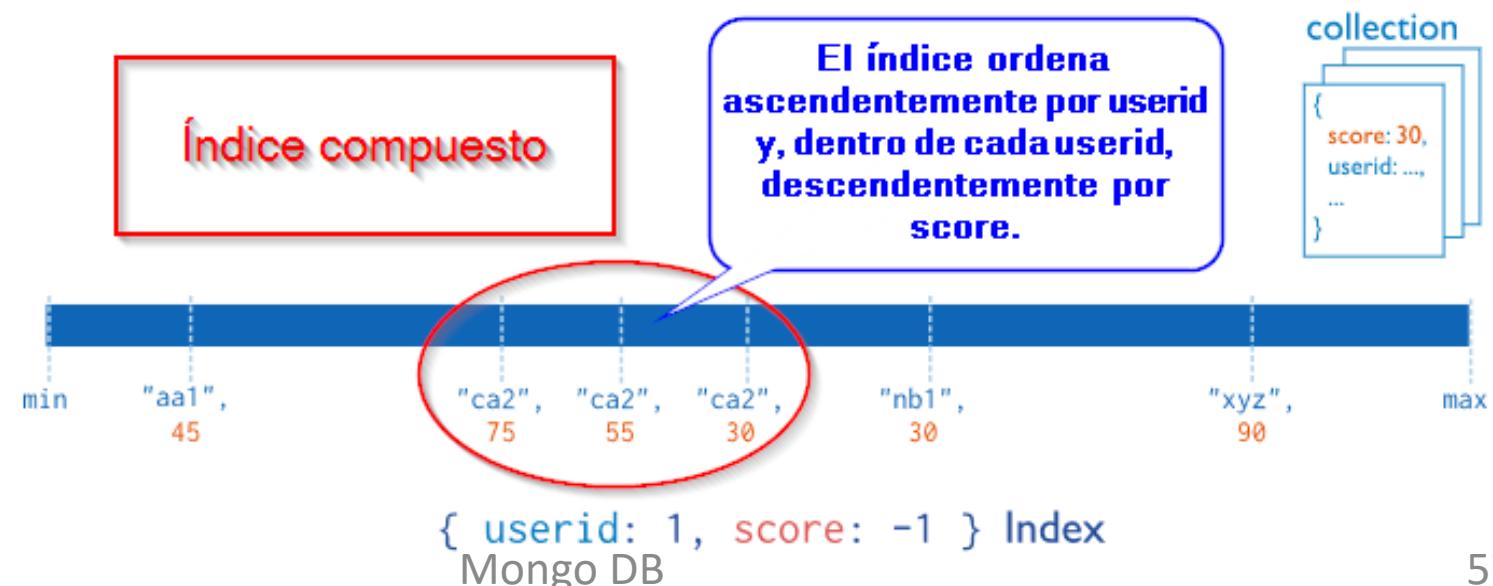
Operaciones básicas con el shell

- **Los índices sirven para que las consultas de lectura sean más rápidas.**
 - Un índice es una parte de los valores de los campos de una colección, almacenados en una estructura de tipo **B-tree** y **ordenados**, permitiendo a las consultas una rápida localización de los valores cuando se utilizan criterios de igualdad o de intervalo (rango).



Operaciones básicas con el shell

- Todas las colecciones poseen por defecto un índice sobre el campo `_id`, que asegura su unicidad, y que no podemos eliminar.
- Tipos de índice:
 - De un solo campo (puede ser un campo de un subdocumento)
 - Compuestos: Índice sobre varios campos (máx: 31), cada uno con su propio criterio de ordenación (1: Ascendente; -1: Descendente)
 - Multicluve: Se aplican a los campos de tipo array
 - Geoespaciales



Operaciones básicas con el shell

- Vamos a comparar la diferencia entre usar un índice o no usarlo
 - db.personas.find({"Nombre":{\$lt:"L"}}, {"Nombre":1}).explain("executionStats")

```
> db.personas.find({"Nombre":{$lt:"L"}}, {"Nombre":1}).explain("executionStats")
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "centro.personas",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "Nombre" : {
                "$lt" : "L"
            }
        },
        "winningPlan" : {
            "stage" : "PROJECTION",
            "transformBy" : {
                "Nombre" : 1
            },
            "inputStage" : {
                "stage" : "COLLSCAN",
                "filter" : {
                    "Nombre" : {
                        "$lt" : "L"
                    }
                },
                "direction" : "forward"
            }
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 2,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 7,
        "executionStages" : {
            "stage" : "PROJECTION",
            "nReturned" : 4,
            "executionTimeMillisEstimate" : 0
        }
    }
}
```

COLLSCAN: Indica que se ha consultado la colección

No se ha usado ningún índice

Se han examinado 7 documentos: todos

Operaciones básicas con el shell

- Crear un índice sobre el campo Nombre con orden ascendente
 - db.personas.createIndex({"Nombre":1})
 - db.personas.find({"Nombre":{\$lt:"L"}}, {"Nombre":1}).explain("executionStats")

```
    "winningPlan" : {  
        "stage" : "PROJECTION",  
        "transformBy" : {  
            "Nombre" : 1  
        },  
        "inputStage" : {  
            "stage" : "FETCH",  
            "inputStage" : {  
                "stage" : "IXSCAN",  
                "keyPattern" : {  
                    "Nombre" : 1  
                },  
                "indexName" : "Nombre_1",  
                "isMultiKey" : false,  
                "multiKeyPaths" : {  
                    "Nombre" : [ ]  
                },  
                "isUnique" : false,  
                "isSparse" : false,  
                "isPartial" : false,  
                "indexVersion" : 2,  
                "direction" : "forward",  
                "indexBounds" : {  
                    "Nombre" : [  
                        "[\\\"\\\", \\"L\\\"]"  
                    ]  
                }  
            }  
        },  
        "rejectedPlans" : [ ]  
    },  
    "executionStats" : {  
        "executionSuccess" : true,  
        "nReturned" : 4,  
        "executionTimeMillis" : 0,  
        "totalKeysExamined" : 4,  
        "totalDocsExamined" : 4,  
        "executionStages" : {  
            "stage" : "PROJECTION",  
            "inputStage" : {  
                "stage" : "IXSCAN",  
                "keyPattern" : {  
                    "Nombre" : 1  
                },  
                "indexName" : "Nombre_1",  
                "isMultiKey" : false,  
                "multiKeyPaths" : {  
                    "Nombre" : [ ]  
                },  
                "isUnique" : false,  
                "isSparse" : false,  
                "isPartial" : false,  
                "indexVersion" : 2,  
                "direction" : "forward",  
                "indexBounds" : {  
                    "Nombre" : [  
                        "[\\\"\\\", \\"L\\\"]"  
                    ]  
                }  
            }  
        }  
    }  
}
```

IXSCAN: En este caso se ha usado un índice

Sólo ha sido necesario examinar 4 documentos

Mongo DB

Operaciones básicas con el shell

- Ejercicio (índices sobre campos de subdocumentos)
 - Crear una colección llamada clientes con colation en y strength 1 (no discrimina mayúsculas/minúculas, acentos...)
 - db.createCollection("Clientes", {collation:{locale:"en",strength:1}})
 - Importar la colección clientes
 - mongoimport C:\borrar\customers-test-data.json -d centro -c clientes --jsonArray

```
db.Clientes.find({  
    "Name.Last Name" : "Johnston"  
}, {  
    "_id" : NumberInt(0),  
    "Name.First Name" : NumberInt(1),  
    "Name.Last Name" : NumberInt(1)  
}).sort({  
    "Name.Last Name" : NumberInt(1)  
}).explain("executionStats")
```

Operaciones básicas con el shell

- Ejercicio (cont.)
 - Crear un índice para el Last Name
 - db.Clientes.createIndex({ "Name.Last Name" : 1 })
 - Repetir la consulta anterior y comprobar cómo desciende el tiempo de ejecución (de 68 a 5)
 - Borrar el índice de Last Name
 - db.Clientes.getIndexes()
 - db.Clientes.dropIndex({"Name.Last Name" : 1})
 - Crear un nuevo índice que incluya tanto el Last Name como el First Name (índice compuesto)
 - db.Clientes.createIndex({ "Name.Last Name" : 1,"Name.First Name" : 1 })
 - Repetir la consulta anterior y comprobar cómo desciende el tiempo de ejecución (de 5 a 2). En este caso tarda menos porque ni siquiera es necesario consultar la colección, pues toda la información necesaria está en el propio índice.
 - Esta situación se denomina "Covered Query"

Operaciones básicas con el shell

- Ejercicio (La importancia del orden en los índices compuestos)
 - Realizar esta consulta:
 - db.Clientes.find().sort({"Name.Last Name":1,"Name.First Name":-1})
 - A pesar de que tenemos un índice para Last Name y First Name, no se puede ejecutar la consulta porque se supera el límite de 32MB para operaciones sort
 - Borrar todos los índices y crear el siguiente
 - db.Clientes.dropIndexes()
 - db.Clientes.createIndex({ "Name.Last Name" : 1,"Name.First Name" : -1 })
 - Comprobar que ahora se puede realizar tanto la consulta anterior como ésta:
 - db.Clientes.find().sort({"Name.Last Name":-1,"Name.First Name":1})
 - Borrar todos los índices y crear el siguiente
 - db.Clientes.createIndex({"Name.First Name" : -1, "Name.Last Name" : 1 })
 - ¿Se podrá realizar la consulta anterior?

Operaciones básicas con el shell

- Índices multiclave: Indexan automáticamente todos los elementos de un array
 - Comprobar que la siguiente consulta no utiliza ningún índice:
 - db.personas.find({Asignaturas: "Física"}).explain("executionStats")
 - Crear el siguiente índice
 - db.personas.createIndex({"Asignaturas": 1})
 - Verificar que ahora la misma consulta anterior sí utiliza este índice
- Crear el siguiente índice
 - db.personas.createIndex({Nominas: 1})
 - Comprobar que la siguiente consulta no lo aprovecha
 - db.personas.find({"Nominas.Mes": "Octubre"})
 - Pero esta sí:
 - db.personas.find({"Nominas": {"Mes":Octubre", "Cantidad": 1850.90}}).explain("executionStats")
 - Para que pudiera usar un índice tendríamos que crearlo, por ejemplo, así:
 - db.personas.createIndex({"Nominas.Mes":1,"Nominas.Cantidad":-1})

Operaciones básicas con el shell

- En MongoDB se pueden utilizar 2 formatos para los datos de tipo geoespacial:
 - **Objeto GeoJSON** (opción recomendada)
 - Es un documento compuesto por 2 campos:
 - type: Indica el tipo de geometría (punto, línea, polígono...)
 - coordinates: Es un array de arrays compuestos por 2 valores [longitud,latitud]
 - Ubicacion:{"type": "LineString", "coordinates":[[0,0],[90,45]]}
 - Podemos usar geojson.io para visualizarlo
 - Los cálculos de distancia se realizan sobre el geoide WGS84
 - **Pares de coordenadas** (legacy; compatibilidad formatos antiguos)
 - Ubicacion:[longitud,latitud]
 - Los cálculos de distancia se realizan sobre un plano euclídeo

Operaciones básicas con el shell

- **Operadores geoespaciales**

- **\$geoIntersects**: Selecciona las geometrías que tienen intersección con la indicada.
- **\$geoWithin**: Selecciona las geometrías que están dentro de la especificada de tipo Polygon o MultiPolygon (o \$centerSphere)
- **\$near**: Devuelve los documentos ordenados desde el más próximo al más alejado del punto indicado. Requiere un index de tipo geoespacial (2d o 2dsphere, en función de que el punto se especifique en formato legacy o GeoJSON, respectivamente). **Si el índice es 2D realiza los cálculos en proyección plana.**
- **\$nearSphere**: Devuelve los documentos ordenados desde el más próximo al más alejado del punto indicado, **utilizando siempre geometría esférica** (incluso si el índice es 2d). Requiere un index de tipo geoespacial (2d o 2dsphere).

Operaciones básicas con el shell

- Ejercicio: Añadir a los 3 profesores de personas un campo GeoJSON de tipo Point llamado Ubicación, con las coordenadas de Madrid, Santander y Cádiz
 - db.personas.updateOne({Nombre:"Juan Félix"},{\$set:{"Ubicacion":{type:"Point",coordinates:[-3.68,40.41]}}})
 - db.personas.updateOne({Nombre:"Ana"},{\$set:{"Ubicacion":{type:"Point",coordinates:[-3.81,43.46]}}})
 - db.personas.updateOne({Nombre:"Irene"},{\$set:{"Ubicacion":{type:"Point",coordinates:[-6.27,36.51]}}})
- Buscar todos los profesores cuya ubicación esté a menos de 300km de Córdoba
 - db.personas.find({Ubicacion:\$geoWithin:{\$centerSphere:[[-4.75,37.88],300/6370]}})

Operaciones básicas con el shell

- Tipos de geometrías del objeto GeoJSON:
 - **Point**
 - **LineString**
 - **Polygon**. Son polígonos cerrados; el último par de coordenadas debe coincidir con el primero
 - {type: "Polygon", coordinates: [[[0, 0], [3, 6], [6, 1], [0, 0]]]}
 - **MultiPoint**
 - **MultiLineString**
 - **MultiPolygon**
 - **GeometryCollection**
- **Se crean con el operador \$geometry**
 - **\$geometry: {type: "Point" ,coordinates: [<longitude> , <latitude>]}**

Operaciones básicas con el shell

- Además de \$geometry, podemos usar los siguientes especificadores de geometría:
 - **\$centerSphere** (funciona con índices **2d** y **2dsphere**)
 - { \$centerSphere: [[<x>, <y>], <radius>] }
 - El radio debe expresarse en radianes
 - **\$maxDistance**: Permite especificar una distancia máxima para los resultados de \$near y \$nearSphere (funciona con índices **2d** y **2dsphere**)
 - { \$near: [-74 , 40], \$maxDistance: 10 }
 - La distancia se expresa en metros
 - **\$minDistance**: Permite especificar una distancia mínima para los resultados de \$near y \$nearSphere (sólo funciona con índices **2dsphere**)
 - { \$near:{\$geometry: { type: "Point", coordinates: [-73.9667, 40.78] }, \$minDistance: 1000, \$maxDistance: 5000}}
 - La distancia se expresa en metros

Operaciones básicas con el shell

- **Índices geoespaciales:**
 - **2dsphere**: Calcula las distancias en geometría esférica
 - **2d**: Calcula las distancias en geometría plana
- Intentar ordenar los documentos por la distancia a Toledo de su Ubicacion:
 - db.personas.find({Ubicacion:{\$near:{\$geometry:{type:"Point",coordinates:[-4.02,39.87]}}}})
 - No es posible porque no tenemos un índice geoespacial sobre Ubicacion
- Crear un índice de tipo 2dsphere sobre el campo personas.Ubicacion
 - db.personas.createIndex({Ubicacion: "2dsphere"})
- Comprobar que ahora sí se puede ejecutar la consulta
 - db.personas.find({Ubicacion:{\$near:{\$geometry:{type:"Point",coordinates:[-4.02,39.87]}}}})

Operaciones básicas con el shell

- **Ejercicio: Búsqueda de restaurantes por barrios y por proximidad**
 - Importar los datasets de **barrios** y **restaurantes**
 - **mongoimport c:\borrar\neighborhoods.json -d centro -c barrios**
 - **mongoimport c:\borrar\restaurants.json -d centro -c restaurantes**
 - Observar que los restaurantes están ubicados por el campo **location**
 - Y que los barrios lo están por el campo **geometry**

```
> db.restaurantes.findOne()
{
  "_id" : ObjectId("55cba2476c522cafdb053ae4"),
  "location" : {
    "coordinates" : [
      -74.00528899999999,
      40.628886
    ],
    "type" : "Point"
  },
  "name" : "Regina Caterers"
}
```

Operaciones básicas con el shell

- Supongamos que estamos en las coordenadas [-73.856077, 40.848447].
- Para averiguar el barrio al que corresponden estas coordenadas recurrimos al operador \$geoIntersects
 - db.barrios.find({geometry:{\$geoIntersects:{\$geometry:{type:"Point",coordinates:[-73.856077, 40.848447]}}}}, {"name":1})
- Para averiguar todos los restaurantes que hay en un barrio usamos el operador \$geoWithin
 - var geometriaBarrio = db.barrios.findOne({geometry:{\$geoIntersects:{\$geometry:{type:"Point",coordinates:[-73.856077, 40.848447]}}}})
 - db.restaurantes.find({location:{\$geoWithin: {\$geometry:geometriaBarrio.geometry}}}, {name:1})

Operaciones básicas con el shell

- También podemos combinar **\$geoWithin** con **\$centerSphere** para averiguar qué restaurantes están a menos de 1000 metros de la posición (aunque sin ordenar)
 - db.restaurantes.find({location:{\$geoWithin:{\$centerSphere:[[-73.856077, 40.848447],1/6370]}}, {name:1}}).count()
- Para obtenerlos ordenados tenemos que usar **\$nearSphere**
 - db.restaurantes.find({location:{\$nearSphere:{\$geometry:{type:"Point", coordinates:[-73.856077, 40.848447]}, \$maxDistance:1000}}}, {name:1}).count()
- Pero no funcionará a menos que definamos un índice de tipo **2dsphere** sobre location
 - db.restaurantes.createIndex({location:"2dsphere"})

Operaciones básicas con el shell

- Capped Collections / Colecciones cerradas
 - Son colecciones con un tamaño máximo, en las que está garantizado que los resultados de las consultas utilizan el orden de inserción, que se denomina **orden natural**.
 - Si se alcanza el tamaño máximo, se van borrando los documentos más antiguos para dar cabida a los nuevos que se vayan insertando.
 - No permiten borrar documentos
 - No permiten actualizaciones de los documentos que varíen su tamaño
 - En definitiva, están pensadas para almacenar registros (por ejemplo, registro de errores en una aplicación)

Operaciones básicas con el shell

- Creación:
 - db.createCollection("registro", {capped: true, size: 10000})
 - db.createCollection("registro", {capped: true, size: 10000, max: 25})
 - **size** es obligatorio / **max** (número de documentos) es opcional
 - Si se alcanza size empiezan a borrarse los documentos más antiguos aunque no se haya alcanzado aún max
 - for(var i=1;i<=25;i++){ db.registro.insertOne({error: i}) }
 - db.registro.find()
 - db.registro.insertOne({error: 26})
 - db.registro.find()

Operaciones básicas con el shell

- Índices de tipo text
 - Sirven para realizar búsquedas "inteligentes" sobre textos, al estilo del buscador de Google
 - Estos índices se construyen mediante:
 - **Token**: En primer lugar el texto se descompone en tokens, que son aproximadamente equivalentes a las palabras
 - **Stemming**: Son algoritmos que reducen los tokens a la raíz de la que provienen. Por ejemplo, la raíz de "cocinar" sería "cocin"
 - **Stop words**: No se indexan ciertas palabras que por sí solas no aportan "significado" al texto, como artículos, adverbios, ciertos verbos.
 - https://github.com/mongodb/mongo/blob/master/src/mongo/db/fts/stop_words_spanish.txt
 - Son extremadamente complejos y pueden crecer "brutalmente" por lo que **sólo está permitido tener un índice de texto en cada colección** (aunque puede ser un índice compuesto; es decir, que indexe varios campos)

Operaciones básicas con el shell

- Importar el dataset bibliotecas
 - mongoimport c:\borrar\bibliotecas.json -c bibliotecas -d centro --jsonArray
- Explorar el esquema de la colección observando que hay un campo **description**, otro **title** y otro **@type**
 - db.bibliotecas.find().limit(1).pretty()
- MongoDB permite elegir un idioma predeterminado para el índice de texto.
 - <https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages>
 - db.bibliotecas.createIndex({description: "text"}, {default_language: "spanish"})

Operaciones básicas con el shell

- Para realizar una búsqueda hay que combinar los operadores **\$text** y **\$search**
 - db.bibliotecas.find({\$text:{\$search:"cocina"}}, {description:1,score:{\$meta: "textScore"}})
- Cada coincidencia recibe una puntuación de similitud, que podemos ver mediante el operador **\$meta** configurado con el valor **textScore** (que usa un orden **descendente** en combinación con sort)
 - db.bibliotecas.find({\$text:{\$search:"cocina"}}, {description:1,puntuacion:{\$meta: "textScore"}})

Operaciones básicas con el shell

- Por defecto, las búsquedas realizadas con \$text no tienen sensibilidad de mayúsculas/minúsculas ni diacrítica, pero podemos activarla asignando el valor true a las opciones **\$caseSensitive** y **\$diacriticSensitive**:
 - db.bibliotecas.find({\$text:{\$search:"Cuentos"},\$caseSensitive: true},{description:1,puntuacion:{\$meta: "textScore"}}).count()
 - db.bibliotecas.find({\$text:{\$search:"Cuentos"}}, {description:1,puntuacion:{\$meta: "textScore"}}).count()
 - db.bibliotecas.find({\$text:{\$search:"poesia"},\$diacriticSensitive: true},{description:1,puntuacion:{\$meta: "textScore"}})
 - db.bibliotecas.find({\$text:{\$search:"poesía", \$diacriticSensitive: true}}, {description:1,puntuacion:{\$meta: "textScore"}})

Operaciones básicas con el shell

- Cuando indicamos varios términos en \$search, se consideran como un OR
- Para realizar búsquedas de frases exactas tenemos que escapar las comillas con el carácter \'
 - db.bibliotecas.find({\$text:{\$search:"\'tradición oral española\'"}}, {description:1,puntuacion:{\$meta: "textScore"}})
 - db.bibliotecas.find({\$text:{\$search:"tradición oral española"}}, {description:1,puntuacion:{\$meta: "textScore"}}).count()
- Se puede incluir un - delante de un término para excluirlo de la búsqueda
 - db.bibliotecas.find({\$text:{\$search:"rimas -colmos"}}, {description:1,puntuacion:{\$meta: "textScore"}})
 - db.bibliotecas.find({\$text:{\$search:"rimas"}}, {description:1,puntuacion:{\$meta: "textScore"}}).count()

Operaciones básicas con el shell

- Índices de textos compuestos
 - En estos índices podemos ponderar el peso de cada campo mediante la opción weight
 - db.bibliotecas.createIndex({title: "text", description: "text"})
 - Recuerde que sólo podemos tener un índice de texto en cada colección
 - db.bibliotecas.dropIndexes()
 - db.bibliotecas.find({\$text:{\$search:"historia"}}, {title:1,description:1,puntuacion:{\$meta: "textScore"}}).sort({ puntuacion: { \$meta: "textScore" } }).limit(1)
 - "puntuacion" : **1.0865800865800865**
 - db.bibliotecas.createIndex({title: "text", description: "text"},{weights: {title:2,description:1}})
 - db.bibliotecas.find({\$text:{\$search:"historia"}}, {title:1,description:1,puntuacion:{\$meta: "textScore"}}).sort({ puntuacion: { \$meta: "textScore" } }).limit(1)
 - "puntuacion" : **1.658008658008658**

Operaciones básicas con el shell

- Si el campo puede contener texto en diferentes idiomas, podemos indexarlo correctamente si además contiene un campo que indique el idioma correspondiente; por defecto el campo debe usar la clave language, pero podríamos usar otro a través de la opción { language_override: "<campo_idioma>" }

```
db.citas.insertMany( [  
    {_id: 1,language: "portuguese",original: "A sorte protege os audazes.",translation:  
        [ {language: "english",quote: "Fortune favors the bold." },{language:  
            "spanish",quote: "La suerte protege a los audaces." } ] },  
    {_id: 2,language: "spanish",original: "Nada hay más surrealista que la  
realidad.",translation: [ { language: "english", quote: "There is nothing more surreal  
than reality." }, { language: "french", quote: "Il n'y a rien de plus surréaliste que la  
réalité." } ] },  
    {_id: 3,original: "is this a dagger which I see before me.",translation:{ language:  
        "spanish", quote: "Es este un puñal que veo delante de mí."} }  
]
```

Operaciones básicas con el shell

- Al crear el índice se usará el valor del campo language
 - db.citas.createIndex({ original: "text", "translation.quote": "text" })
- Y al realizar las búsquedas tendremos que indicar el idioma:
 - db.citas.find({\$text:{\$search: "puñalada", \$language:"spanish"}})

Operaciones básicas con el shell

- Documentos referenciados
 - Referencias manuales: Es la opción más sencilla
 - Referencias con DBRef: Puede resultar más cómoda para utilizar con drivers
- Referencia manual
 - `juanfe=db.personas.findOne({"Nombre": "Juan Félix"})`
 - `db.comentarios.insertOne({"ref_personas":juanfe._id,"Comentario":"Esto es un comentario en un foro"})`
 - `comentario=db.comentarios.findOne()`
 - `db.personas.findOne({"_id":comentario.ref_personas})`

Operaciones básicas con el shell

- Referencia con DBRef
 - DBRef es simplemente un formato de documento que establece una relación estandarizada con otro documento. Su formato es éste, y el orden importa. Obsérvese que el nombre del campo que contiene el id del objeto referenciado es **\$id**

```
{  
    "$ref" : "<colección>",  
    "$id" : "<_id del documento referenciado>",  
    "$db" : "<base de datos>"  
}
```

Operaciones básicas con el shell

- db.comentarios.insertOne({"ref_autor":new DBRef("personas",juanfe._id,"centro"),comentario:"Esto es otro comentario"})
- db.comentarios.find()
- otro_comentario=db.comentarios.findOne({"comentario": "Esto es otro comentario"})
- db.personas.findOne({_id:otro_comentario.ref_autor.**\$id**})



3. Operaciones avanzadas del Shell

Operaciones avanzadas con el shell

- Agregación
 - Básicamente hay 2 formas de realizar agregaciones en MongoDB:
 - **Aggregation Pipeline Framework:**
 - Optimizado para mayor **velocidad** y análisis en **tiempo real**
 - Puede aprovechar los índices
 - Limitado a datasets de 100M
 - La salida es un índice o un documento o una colección
 - Sólo se pueden utilizar los comandos existentes
 - **MapReduce**
 - Optimizado para trabajar sobre **conjuntos de datos muy grandes**
 - Más flexible porque permite escribir funciones personalizadas de map y reduce en JavaScript
 - Más opciones de salida: inline, colección, merge, replace...

Operaciones avanzadas con el shell

- Aggregation Pipeline Framework
 - Consiste en hacer pasar el conjunto de datos original (**procedente de una misma colección**) por etapas sucesivas de filtrado, agregación, ordenación...
 - Cada etapa actúa sobre el resultado de la etapa anterior
 - Se ejecuta con db.<colección>.aggregate([array de etapas])
 - **[array de etapas]** está compuesto por documentos que describen las etapas a aplicar mediante comandos



Operaciones avanzadas con el shell

- Comandos fundamentales de etapas:
 - **\$group**: Admite operadores de acumulación como \$sum, \$avg, \$max, \$min...
 - **\$match**: Sólo permite pasar a la siguiente etapa a los documentos que cumplen el criterio especificado
 - **\$limit**: Limita el número de documentos que pasan a la siguiente etapa
 - **\$skip**: Pasa a la siguiente etapa todos los documentos menos los n primeros
 - **\$sort**: Ordena los documentos según los criterios especificados
 - **\$project**: Añade o elimina campos de los documentos antes de pasar a la siguiente etapa
 - **\$count**: Devuelve el número de documentos que contiene la etapa actual; podría considerarse como una combinación de \$group y \$project
 - **\$unwind**: Descompone un array de modo que cada uno de sus datos se almacena en un documento diferente para la siguiente etapa
 - **\$lookup**: Cree una outer left join sobre otra colección de la misma base de datos, añadiendo a cada documento de la etapa anterior y array con las coincidencias en la segunda colección
 - **\$out**: Guarda el resultado de la agregación en una colección (tiene que ser el último comando del pipeline)
 - **\$geoNear**: Ordena los documentos del más próximo al más alejado de unas coordenadas especificadas

Operaciones avanzadas con el shell

- Equivalencia SQL → Comandos de agregación MongoDB

WHERE	<code>\$match</code>
GROUP BY	<code>\$group</code>
HAVING	<code>\$match</code>
SELECT	<code>\$project</code>
ORDER BY	<code>\$sort</code>
LIMIT	<code>\$limit</code>
SUM()	<code>\$sum</code>
COUNT()	<code>\$sum</code> <code>\$sortByCount</code>
join	<code>\$lookup</code>

Mongo DB

Operaciones avanzadas con el shell

- Importar la colección zips.json
 - mongoimport c:\borrar\zips.json -c zips -d centro
 - El campo pop indica la población

```
> db.zips.findOne()
{
    "_id" : "01007",
    "city" : "BELCHERTOWN",
    "loc" : [
        -72.410953,
        42.275103
    ],
    "pop" : 10579,
    "state" : "MA"
}
```

Operaciones avanzadas con el shell

- **\$group**
 - El documento de salida debe contener obligatoriamente un campo **_id** con la clave del campo que queramos usar como clave única de agrupación
 - La clave única de agrupación puede estar compuesta por varias claves de campo para crear grupos dentro de grupos
 - Adicionalmente puede contener otros campos creados a través de operadores de acumulación (**\$sum**, **\$avg**, **\$max**, **\$push**, **\$first**, **\$last**...)
 - Mostrar los estados
 - `db.zips.aggregate([{$group:{_id:"$state"} }])`
 - Obsérvese el **\$** delante de **\$state**
 - Mostrar los estados y sus ciudades
 - `db.zips.aggregate([{$group:{_id:{estado:"$state",ciudad:"$city"} } }])`

Operaciones avanzadas con el shell

- \$group
 - Contar los estados
 - db.zips.aggregate([{\$group:{_id:"\$state"}},{\$group:{_id: null, total: {\$sum:1}}}])
 - Obsérvese cómo se usa _id: null para indicar que se sumen todos documentos de la etapa anterior
 - Obsérvese cómo se usa \$sum: 1 para sumar una unidad por cada coincidencia. Con \$sum: 2 nos saldrían el doble de estados
 - Contar las ciudades
 - db.zips.aggregate([{\$group:{_id:{estado:"\$state",ciudad:"\$city"} }},{\$group: {_id: "\$_id.estado",numeroCiudades:{\$sum:1}}}])
 - Obsérvese cómo se usa la sintaxis de punto para especificar la clave única en la segunda etapa de agrupación

Operaciones avanzadas con el shell

- \$group
 - Contar los códigos postales en cada estado
 - db.zips.aggregate([{\$group:{_id:"\$state", total: {\$sum:1}}}])
 - Averiguar la población total
 - db.zips.aggregate([{\$group:{_id:null, total: {\$sum:"\$pop"}}}])
 - Obsérvese cómo se indica a \$sum que acumule los valores de los campo pop
 - Averiguar la población total por estado
 - db.zips.aggregate([{\$group:{_id:"\$state", total: {\$sum:"\$pop"}}}])
 - Mostrar cada estado y un array de los zips que le pertenecen
 - db.zips.aggregate([{\$group:{_id:"\$state", zips: {\$push:"\$_id"}}}])
 - **\$first:<clave>** devuelve el primer valor de la clave especificada de cada agrupamiento de \$group
 - Mostrar cada estado y la primera de sus ciudades
 - db.zips.aggregate([{\$group:{_id:"\$state",ciudad:{\$first:"\$city"}}}])

Operaciones avanzadas con el shell

- \$match
 - Mostrar todos los documentos de Ohio [OH]
 - db.zips.aggregate([{\$match:{state:'OH'}}])
 - Mostrar la población de las ciudades de Ohio
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}}])
 - Mostrar todas las ciudades de Ohio con más de 100.000 habitantes
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}}, {\$match:{poblacion:{\$gt:100000}}}])
 - Averiguar la población total de todas las ciudades de Ohio con más de 100.000 habitantes
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}}, {\$match:{poblacion:{\$gt:100000}}}, {\$group:{_id: null, total: {\$sum:"\$poblacion"}}}])

Operaciones avanzadas con el shell

- \$limit y \$skip
 - Mostrar las ciudades de Ohio de 5 en 5
 - db.zips.aggregate([{\$match:{state:'OH'}},{\$limit:5}])
 - db.zips.aggregate([{\$match:{state:'OH'}},{\$limit:5},{\$skip:5}])
 - db.zips.aggregate([{\$match:{state:'OH'}},{\$skip: 5},{\$limit:5}])

Operaciones avanzadas con el shell

- \$sort
 - Cuál es la máxima población de una ciudad de Ohio
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}, {\$group:{_id:null,maximaPoblacion:{\$max: "\$poblacion"}}}}]
 - Pero ¿cuál es la ciudad que tiene esa población?
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}, {\$sort:{poblacion:-1}}, {\$limit:1}}])
 - ¿Cuál es la tercera ciudad más poblada de Ohio, y cuál es su población?
 - db.zips.aggregate([{\$match:{state:'OH'}}, {\$group:{_id:"\$city", poblacion: {\$sum:"\$pop"}}, {\$sort:{poblacion:-1}}, {\$skip:2}, {\$limit:1}}])

Operaciones avanzadas con el shell

- \$project
 - Básicamente permite indicar qué campos de la etapa anterior queremos mantener y/o eliminar, como en la proyección de un find.
 - Pero también sirve para añadir campos calculados gracias al uso de los operadores de agregación:
 - <https://docs.mongodb.com/manual/reference/operator/aggregation/#accumulators-project>
 - Supongamos que hay que proporcionar 3 dosis de vacuna a toda la población de la ciudad de YOUNGSTOWN; mostrar los ZIPS de YOUNGSTOWN ordenados ascendentemente, la población de cada ZIP, y las dosis necesarias para cada uno de ellos
 - db.zips.aggregate([{\$match:{city:"YOUNGSTOWN"}},{\$sort:{"_id":1}},{\$project:{poblacion:"\$pop",dosis:{\$multiply:["\$pop",3]}}}])
 - Incluir en el documento anterior un campo con la clave estado y el valor "OHIO"
 - db.zips.aggregate([{\$match:{city:"YOUNGSTOWN"}},{\$sort:{"_id":1}},{\$project:{poblacion:"\$pop",dosis:{\$multiply:["\$pop",3]},estado:{\$literal:"OHIO"}}}])

Operaciones avanzadas con el shell

- **\$count: <nombre del campo>**
 - Este comando de etapa sólo requiere el nombre del campo al que queremos asignar como valor el total de documentos de la etapa anterior
 - ¿Cuántos ZIPS hay en la colección?
 - db.zips.aggregate([{\$count:"totalZips"}])
 - Podría obtenerse el mismo resultado con:
 - db.zips.count()
 - ¿Cuántos estados hay?
 - db.zips.aggregate([{\$group:{_id:"\$state"}},{\$count:"totalEstados"}])



Operaciones avanzadas con el shell

- Ejercicio
 - 1) ¿Qué estados tienen más de 10 millones de habitantes?
 - 2) ¿Cuál es la población media de las ciudades de cada estado?
 - 3) ¿Cuáles son las ciudades de cada estado con mayor y menor población?

Operaciones avanzadas con el shell

- Solución 1

```
db.zips.aggregate( [ {  
    $group: { _id: "$state", poblacion: { $sum: "$pop" } } },  
    { $match: { poblacion: { $gte: 10*1000*1000 } } }  
] )
```

Operaciones avanzadas con el shell

- Solución 2

```
db.zips.aggregate( [  
    { $group: {  
        _id: { estado: "$state", ciudad: "$city" },  
        poblacion: { $sum: "$pop" } } },  
    { $group: {  
        _id: "$_id.estado",  
        poblacionMediaCiudades: { $avg: "$poblacion" } } }  
])
```

Operaciones avanzadas con el shell

- Solución 3

```
db.zips.aggregate([
    {$group:
        {_id:{estado:"$state",ciudad:"$city"},poblacion:{$sum:"$pop"}}
    },
    {$sort:{"_id.estado":1,poblacion:-1}},
    {$group:
        {_id:"$_id.estado",
        mayor:{$first:"$_id.ciudad"},
        menor:{$last:"$_id.ciudad"}}
    },
    {$sort:{_id:1}}
])
```

Operaciones avanzadas con el shell

- \$unwind
 - Descompone un campo de tipo array de modo que produce un documento para cada uno de los valores de ese campo
 - Cada uno de los documentos producidos es idéntico al original pero con el campo de tipo array sustituido por cada uno de los valores del array
 - Importar el dataset de los datos de bicimad de agosto de
 - mongoimport c:\borrar\bicimad.json -c bicimad -d centro
 - Estraer en un array las calles por las que ha transcurrido cada itinerario
 - db.bicimad.aggregate([{\$limit:10}, {\$unwind:"\$track.features"}, {\$project: {calle:"\$track.features.properties.var"}}, {\$group:{_id:"\$_id",itinerario: {\$push:"\$calle"}}}])

Operaciones avanzadas con el shell

- \$out: Permite crear una colección a partir de los documentos devueltos por un pipeline de agregación
 - mongoimport c:\borrar\bicimad_estaciones.json -c bicimad_estaciones -d centro
 - Crear una colección con las estaciones del primer documento de bicimad_estaciones
 - db.bicimad_estaciones.aggregate([{\$limit:1}, {\$unwind:"\$stations"}, {\$project: {_id:0, stations:1}}, {\$out:"bicimad_estacion"}])

Operaciones avanzadas con el shell

- `$lookup{from:"", localField:"", foreignFiled:"", as:""}`
 - Añade a cada documento de entrada un array con la clave as compuesto por los documentos de la colección from en los que el foreignField coincide con el localField del documento de entrada.
 - Obtener los nombres de las estaciones en las que se han alquilado las bicicletas (idunplug_station) y se han devuelto (idplug_station) para cada trayecto. Mostrarlos en 2 campos con las claves "inicio" y "fin"
 - db.bicimad.aggregate([{\$limit:1}, {\$lookup:
 {from:"bicimad_estacion", localField:"idunplug_station", foreignField:"stations.id", as:"inicio"}}, {\$lookup:
 {from:"bicimad_estacion", localField:"idplug_station", foreignField:"stations.id", as:"fin"}}, {\$project:{_id:0, "inicio.stations.name":1, "fin.stations.name":1}}, {\$project:
 {inicio:"\$inicio.stations.name", fin:"\$fin.stations.name"}}])

Operaciones avanzadas con el shell

- **\$geoNear:** Muestra una lista de los documentos ordenados desde el más próximo hasta el más alejado del punto especificado.
 - Opciones
 - **near:** Para especificar el punto respecto al que queremos medir las distancias en formato GeoJSON
 - posicion: {type: "Point",coordinates: [-73.856077, 40.848447]}
 - **distanceField:** Campo de salida en el que queremos almacenar la distancia

Operaciones avanzadas con el shell

- Vamos a modificar el esquema de la colección bicimad_estacion para que las ubicaciones esté en un campo con formato GeoJSON de la forma:

```
posicion: {  
    type: "Point",  
    coordinates: [-73.856077, 40.848447]  
}
```

- Podemos utilizar el operador \$toDecimal para convertir las coordenadas, que están en formato de cadena de caracteres, a números decimales.
- db.bicimad_estacion.aggregate([{\$group:{_id:"\$_id",nombre:{\$first:"\$stations.name"},longitud:{\$push:{\$toDecimal:"\$stations.longitude"}},latitud:{\$push:{\$toDecimal:"\$stations.latitude"}}}},{\$project:{nombre:"\$nombre","ubicacion.type":"Point","ubicacion.coordinates":{\$concatArrays:["\$longitud", "\$latitud"]}}},{\$out:"bicimad_estaciones_geojson"}])

Operaciones avanzadas con el shell

- Creamos un índice 2dsphere sobre el campo ubicación
 - db.bicimad_estaciones_geojson.createIndex({ubicacion:"2dsphere"})
- Obtenemos un listados de las estaciones de bicicletas ordenadas por proximidad al "IES EL Lago" $([-3.730441, 40.415247])$
 - db.bicimad_estaciones_geojson.aggregate([{\$geoNear:{near: {type:"Point",coordinates:[-3.730441,40.415247]},distanceField:"distancia"}}])

Operaciones avanzadas con el shell

- **\$bucket**
 - Agrupa los documentos de la etapa anterior en subdocumentos que cumplen una determinada condición, especificada en **groupBy**, clasificándolos en intervalos (buckets → cubos) cuyos extremos se configuran mediante **boundaries**.
 - Los intervalos (boundaries) son cerrados (inclusivos) por el extremo inferior y abiertos (exclusivos) por el extremo superior.
 - Todos los documentos de la etapa de entrada deben poder evaluar la condición de groupBy; en caso contrario (por ejemplo, si a alguno de los documentos le falta el campo que se utiliza en la condición de groupBy), deberemos indicar un nombre de intervalo en el que quedarán agrupados todos estos documentos inclasificables. Este nombre de intervalo se configura mediante la opción **default**.
 - Los documentos agrupados generados por \$bucket utilizarán como _id el límite inferior del intervalo, y sólo contendrán el número de documentos de la etapa anterior que caen dentro de ese intervalo. Si queremos incluir más campos podremos hacerlo mediante la opción **output**, en la que podemos utilizar **operadores de acumulación** como **\$sum**, **\$avg**,, **\$push**...

Operaciones avanzadas con el shell

- \$bucket
 - Vamos a clasificar las estaciones de bicimad en función de las bases que poseen:
 - db.bicimad_estacion.aggregate([{\$bucket:{groupBy:"\$stations.total_bases",boundaries:[0,5,10,15,18,20,22,24],default:"NC"}}])
 - Todas las estaciones que no caen en ninguno de los intervalos se almacenan en el documento NC
 - db.bicimad_estacion.aggregate([{\$bucket:{groupBy:"\$stations.total_bases",boundaries:[0,5,10,15,18,20,22,24,26,28,30,32,34,36,38,40],default:"NC"}}])
 - Muestra sólo el _id y count
 - db.bicimad_estacion.aggregate([{\$bucket:{groupBy:"\$stations.total_bases",boundaries:[0,5,10,15,18,20,22,24,26,28,30,32,34,36,38,40],default:"NC",output:{total:{\$sum:1},bases:{\$avg:"\$stations.total_bases"},estaciones:{\$push:"\$stations.name"}}}}]
 - Incluimos el total de estaciones, la media de bases y el nombre de las estaciones.
 - Al usar output tenemos que encargarnos expresamente de calcular el número de documentos que caen en cada colección (ya no se incluye count automáticamente)

Operaciones avanzadas con el shell

- **\$bucketAuto**
 - Funciona igual que \$bucket, pero en lugar de especificar nosotros los límites de los intervalos mediante **boundaries**, utilizamos la propiedad **buckets** para indicar cuántos intervalos deseamos.
 - Opcionalmente, si no queremos que los intervalos sean del mismo tamaño, podemos especificar una serie de números mediante la opción **granularity (números preferentes)**. Podemos elegir entre:
 - **Series de Renard** (R5, R10, R20, R40, R80 y 1-2-5)
 - Se basan en las raíces quinta, décima... de 10. Consecuentemente, los extremos son 1 y 10 (10.3 en el caso de R80). Si los valores de la etapa de entrada son mayores que 10, los límites de Renard se multiplican por la potencia de 10 necesaria para darles cabida.
 - **Series de número E** (E6, E12, E24, E48, E96, E192)
 - Igual que Renard, pero con las raíces sexta, duodécima...
 - **1-2-5** es una variante de R3, redondeada a las cifras 1, 2 y 5: 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500...
 - **Potencias de 2 (POWERSOF2)**
 - Utiliza como extremos las potencias de 2: 1, 2, 4, 8, 16, 32...

Operaciones avanzadas con el shell

- \$bucketAuto
 - Ejemplo de serie de Renard R5
 - $10^{(1/5)} = 1.5848$
 - El intervalo sería:
 - $10^{(0/5)}=1$
 - $10^{(1/5)}=1.5848\sim1.6$
 - $10^{(2/5)}=2.5118\sim2.5$
 - $10^{(3/5)}=3.981\sim4.0$
 - $10^{(4/5)}=6.3095\sim6.3$
 - $10^{(0/5)}=10$
 - ¿Por qué R80 termina en 10.3? Porque se aplica un redondeo

4.12	0.25
4.37	0.25
4.62	0.25
4.87	0.28
5.15	0.30
5.45	0.30
5.75	0.40
6.15	0.35
6.50	0.40
6.90	0.40
7.30	0.45
7.75	0.50
8.25	0.50
8.75	0.50
9.25	0.50
9.75	0.55
10.3	

Operaciones avanzadas con el shell

- \$bucketAuto
 - Vamos a clasificar las estaciones de bicimad en 3 buckets
 - db.bicimad_estacion.aggregate([{\$bucketAuto:
{groupBy:"\$stations.total_bases",buckets:3,output:{total:
{\$sum:1},media_bases:
{\$avg:"\$stations.total_bases"},estaciones:{\$push:"\$stations.name"},bases:
{\$push:"\$stations.total_bases"}}}])
 - ¿Por qué se crean sólo 2 buckets?
 - Por la cardinalidad
 - $[12-18]=1$ (12)
 - $[18-24]=4$ (18, 20, 21, 22)
 - $[24-30]=3$ (24, 27, 30)

Número de bases	Número de estaciones
12	1
18	5
20	1
21	13
22	1
24	138
27	11
30	2

Operaciones avanzadas con el shell

- \$bucketAuto
 - Probar con 9 buckets
 - db.bicimad_estacion.aggregate([{\$bucketAuto: {groupBy:"\$stations.total_bases",buckets:9,output:{total: {\$sum:1},media_bases:{\$avg:"\$stations.total_bases"},estaciones: {\$push:"\$stations.name"},bases:{\$push:"\$stations.total_bases"}}}}]
 - Ahora obtenemos 3 buckets
 - Probar con 18 buckets
 - db.bicimad_estacion.aggregate([{\$bucketAuto: {groupBy:"\$stations.total_bases",buckets:18,output:{total: {\$sum:1},media_bases:{\$avg:"\$stations.total_bases"},estaciones: {\$push:"\$stations.name"},bases:{\$push:"\$stations.total_bases"}}}}]
 - Ahora obtenemos 4 buckets
 - \$autoBucket intenta dividir los **valores únicos de los documentos** uniformemente en los buckets, pero si no puede, creará menos buckets.
 - Probar con 1000 bucket

Operaciones avanzadas con el shell

- \$bucketAuto
 - Para que bucketAuto sea capaz de crear el número de buckets solicitado es necesario que la cardinalidad de la colección sea lo suficientemente alta.
 - Ejercicio
 - Probar a crear 10 buckets con las poblaciones de los códigos postales de la colección zips

Operaciones avanzadas con el shell

- \$bucketAuto
 - Solución del ejercicio
 - db.zips.aggregate([{\$bucketAuto:{groupBy:"\$pop",buckets:10,output:{total:{\$sum:1},media_pop:{\$avg:"\$pop"}}}}])

Operaciones avanzadas con el shell

- \$facet
 - Sirve para categorizar y agrupar los documentos de la etapa anterior atendiendo a diversos criterios, que son pipelines de agregación que deben incluir una etapa de tipo \$bucket o \$bucketAuto
 - Devuelve un único documento en el que cada campo tiene como valor el resultado de aplicar cada uno de esos criterios
 - Su sintaxis es:

```
{ $facet:  
  {  
    <outputField1>: [ <stage1>, <stage2>, ... ],  
    <outputField2>: [ <stage1>, <stage2>, ... ],  
    ...  
  }  
}
```

Operaciones avanzadas con el shell

- \$facet

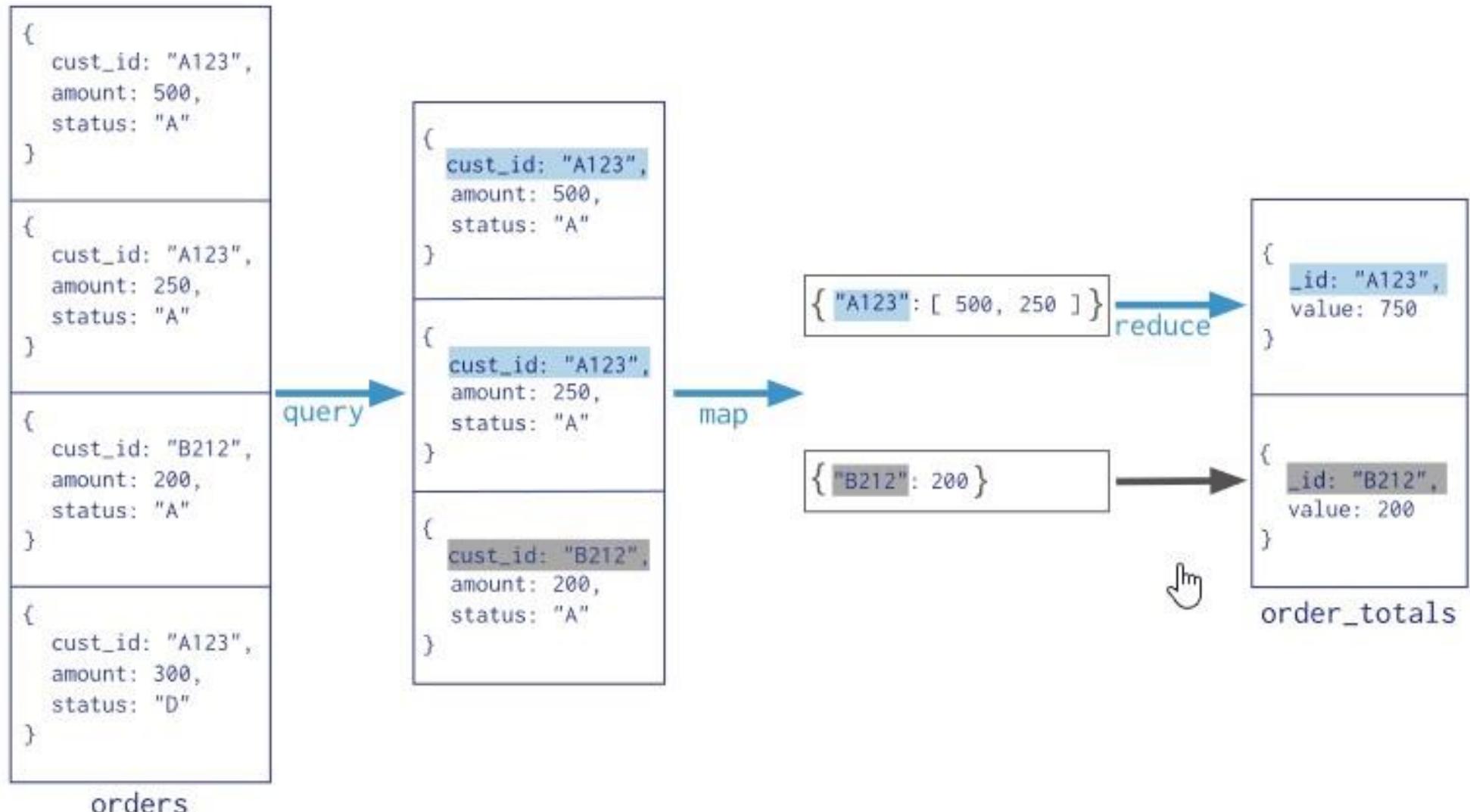
- Vamos a agrupar y clasificar las estaciones de bicicletas por 2 facetas:
 - Número de bases total (total_bases)
 - Número de bases libres (free_bases)

```
db.bicimad_estacion.aggregate([  
    {$facet:{  
        bases_total:[{$bucket: {  
            groupBy: "$stations.total_bases",  
            boundaries: [ 0, 5, 10, 15, 20, 25, 30, 35 ]}]}],  
        bases_libres:[{$bucket: {  
            groupBy: "$stations.free_bases",  
            boundaries: [ 0, 5, 10, 15, 20, 25, 30, 35 ]}]}]  
    }  
])
```

Operaciones avanzadas con el shell

- **mapReduce** es un sistema de agregación diseñado para manejar cantidades enormes de datos (petabytes), por lo que suele ejecutarse en sistemas de procesamiento en paralelo, como Hadoop.
- No obstante, MongoDB también incluye una implementación de mapReduce.
- Su funcionamiento básico consiste en utilizar 2 funciones escritas en JavaScript:
 - **map**: Esta función extrae de la colección pares clave: valor. La clave debe ser un campo de la colección. El valor será un array con los valores de todos los documentos que comparten la misma clave.
 - **reduce**: Esta función es la que realmente agrega los valores de los pares anteriores actuando sobre el array de valores.
 - Adicionalmente admite otros parámetros, como:
 - **finalize**: Se aplica a continuación de reduce para alterar la presentación de los datos.
 - **out**: Permite especificar el destino del resultado de la operación mapReduce: la consola o una colección.
 - **query**: Sirve para limitar el número de documentos que debe procesar la función map.

Operaciones avanzadas con el shell



MUY IMPORTANTE

El tipo de dato devuelto por **reduce** tiene que ser idéntico al tipo del valor emitido por la función map.

Es posible que la función reduce se ejecute **varias veces para una misma clave** (en tal caso la reducción anterior se toma como un valor de entrada más para la reducción actual)

Operaciones avanzadas con el shell

- Vamos a calcular la población total de cada ciudad de nuestra colección de zips (códigos postales)
 - var funcion_map=function(){emit(this.city,this.pop);}
 - var funcion_reduce=function(ciudad,poblacion){return Array.sum(poblacion);}
 - db.zips.mapReduce(funcion_map,funcion_reduce,{out:{inline:1}})
- Resultado

...

```
{  
    "_id" : "ZWOLLE",  
    "value" : 5325  
},  
"timeMillis" : 2381,  
"counts" : {  
    "input" : 29353,  
    "emit" : 29353,  
    "reduce" : 8386,  
    "output" : 16584  
},  
"ok" : 1  
}
```

Operaciones avanzadas con el shell

- ¿Y si quisiéramos pasar el nombre del estado?
 - Usamos el "truco" de emitir como valor un documento JSON que incluya los campos que nos interesen

```
var funcion_map=function(){  
    var estado_poblacion={estado: this.state,poblacion: this.pop};  
    emit(this.city,estado_poblacion);  
}  
var funcion_reduce=function(ciudad,estado_poblacion){  
    var resultado_reduce={ciudad:"",poblacion:0};  
    for(var i=0; i<estado_poblacion.length; i++){  
        resultado_reduce.poblacion+=estado_poblacion[i].poblacion;  
    }  
    resultado_reduce.ciudad=estado_poblacion[0].estado;  
    return resultado_reduce;  
}- db.zips.mapReduce(funcion_map,funcion_reduce,{out:{inline:1}})
```

Operaciones avanzadas con el shell

- Vamos a calcular la población media de las ciudades de cada estado
- Planteamiento
 - 1) Hacemos un primer mapReduce con una clave compuesta por el estado y la ciudad, para conseguir reducir la población total de cada ciudad sumando la de cada uno de sus zips y lo almacenamos en una colección llamada `zips_agregados_por_ciudad`.
 - 2) Realizamos un segundo mapReduce usando como clave el estado de la colección generada en el paso anterior, y acumulando en un contador el número de ciudades que tiene cada estado.

Operaciones avanzadas con el shell

- Paso 1

```
var funcion_map=function(){  
    emit({estado:this.state,ciudad:this.city},this.pop);  
}  
 
```

```
var funcion_reduce=function(estado_ciudad,poblacion){  
    return Array.sum(poblacion);  
}  
 
```

```
db.zips.mapReduce(funcion_map,funcion_reduce,  
{out:"zips_agregados_por_ciudad"})
```

Operaciones avanzadas con el shell

- Paso 2 (1/2)

```
var funcion_map2=function(){
    emit(this._id.estado,{poblacion: this.value, cuenta:1});
}

var funcion_reduce2=function(estado,poblacion){
    respuesta_reduce={poblacion: 0, cuenta:0};
    for(var i=0;i<poblacion.length;i++){
        respuesta_reduce.poblacion+=poblacion[i].poblacion;
        respuesta_reduce.cuenta+=poblacion[i].cuenta;
    }
    return respuesta_reduce;
}
```

Operaciones avanzadas con el shell

- Paso 2 (2/2)

```
funcion_finalize=function(estado,poblacion){  
    poblacion.media=poblacion.poblacion/poblacion.cuenta;  
    return poblacion;  
}  
  
db.zips_agregados_por_ciudad.mapReduce(funcion_map2,funcion_reduce2,{out:{inline:1},finalize:funcion_finalize})
```



4. Conexión a PHP y a JAVA



DRIVER PHP

- La extensión **mongo** se considera obsoleta
- El driver de PHP recomendado actualmente es la extensión **mongodb**
 - Esta actualización pretende aislar las operaciones de más bajo nivel en el dirver mongodb y abstraer las de más alto nivel a una biblioteca llamada **MongoDB PHP library**

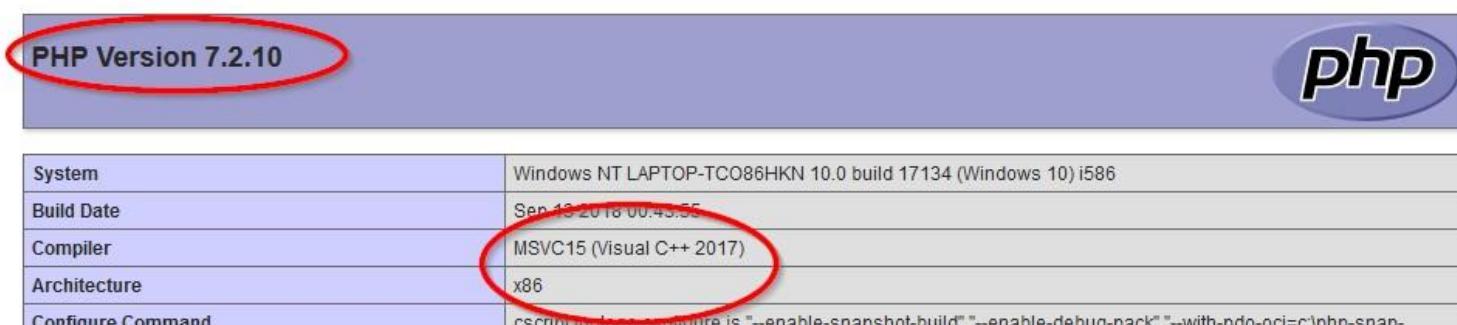


DRIVER PHP

- DRIVER PHP
 - 1. Instalar XAMPP
 - 2. Instalar la extensión mongodb (no confundir con la extensión mongo, que actualmente se considera obsoleta)
 - <https://pecl.php.net/package/mongodb>
 - 3. Opcionalmente, pero muy recomendado, instalar la PHP Library a través de Composer

DRIVER PHP

- INSTALAR LA EXTENSIÓN mongodb
 - 1. Para saber qué versión de la extensión debemos instalar, necesitamos averiguar la versión de PHP de nuestro servidor
 - 1.Crear la carpeta C:\xampp\htdocs\mongodb y, dentro de ella, un archivo llamado phpinfo.php con el siguiente contenido:
 - <?php phpinfo(); ?>
 - 2.Acceder con el navegador web a la dirección
<http://localhost/mongodb/phpinfo.php>



The screenshot shows a browser displaying the output of a PHP info page. At the top, it says "PHP Version 7.2.10". Below that is the PHP logo. The main content is a table with various system information. Two specific entries are circled with red circles: "Build Date" (Sep 13 2018 00:43:35) and "Compiler" (MSVC15 (Visual C++ 2017)).

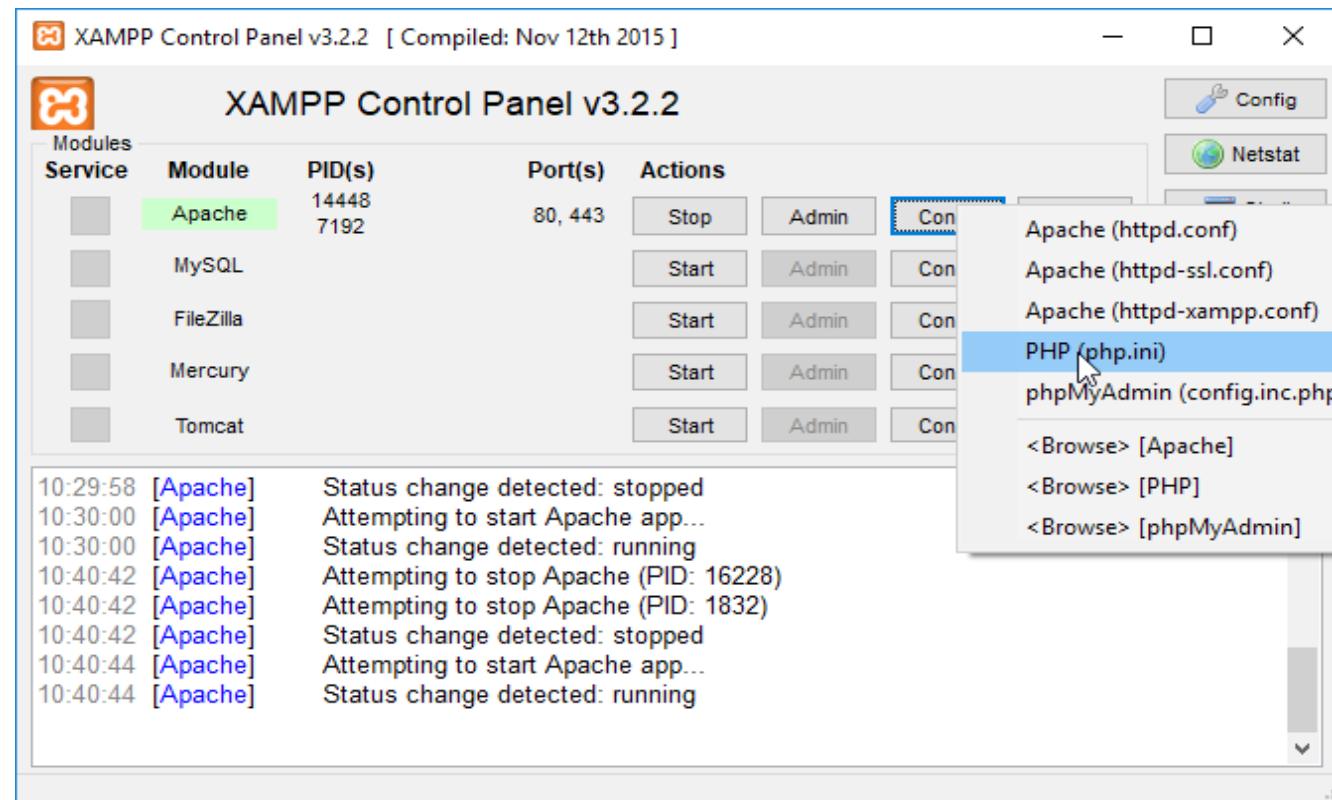
System	Windows NT LAPTOP-TC086HKN 10.0 build 17134 (Windows 10) i586
Build Date	Sep 13 2018 00:43:35
Compiler	MSVC15 (Visual C++ 2017)
Architecture	x86
Configure Command	cscript ..\configure.js --enable-snapshot-build --enable-debug-pack --with-pdo-oci=c:\php-snap-

DRIVER PHP

- INSTALAR LA EXTENSIÓN mongodb
 - Descargar de <https://pecl.php.net/package/mongodb> la dll correspondiente a nuestra versión de php
 - https://windows.php.net/downloads/pecl/releases/mongodb/1.5.3/php_mongodb-1.5.3-7.2-ts-vc15-x86.zip
 - Descomprimir el archivo anterior y copiar la dll en la carpeta de extensiones de xampp
 - C:\xampp\php\ext

DRIVER PHP

- INSTALAR LA EXTENSIÓN mongodb
 - Abrir con XAMPP el archivo php.ini e incluir en él la siguiente línea:
 - extension=php_mongodb.dll
 - Reiniciar el servidor Apache desde XAMPP



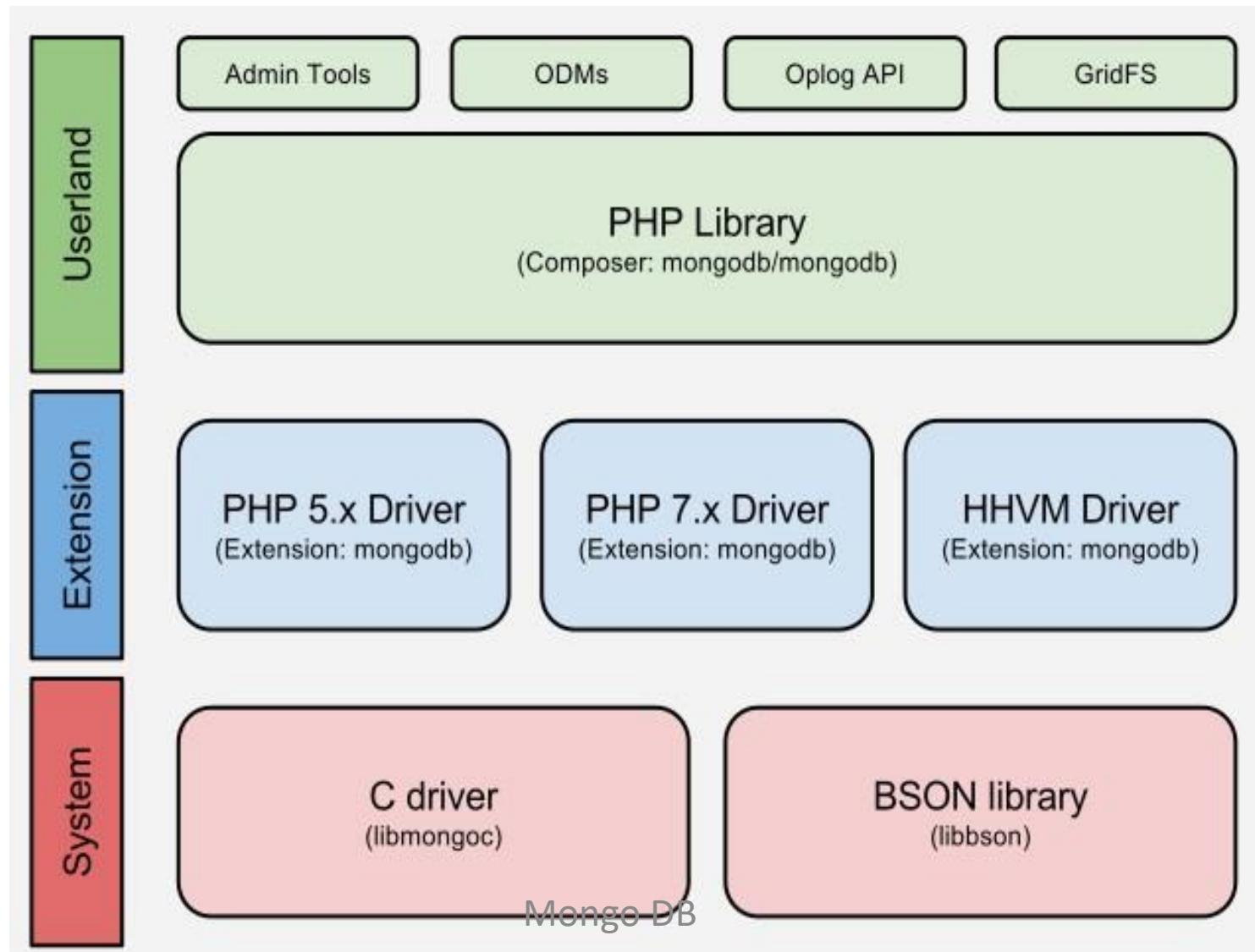
DRIVER PHP

- INSTALAR PHP Library
 - Instalar Composer
 - <https://getcomposer.org/>
 - Desde la línea de comandos, en el directorio en el que anteriormente creamos el archivo phpinfo.php, ejecutar:
 - composer require mongodb/mongodb

```
..\xampp\htdocs\mongodb> composer require mongodb/mongodb
Using version ^1.4 for mongodb/mongodb
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing mongodb/mongodb (1.4.2): Loading from cache
Writing lock file
Generating autoload filesMongo DB
```

DRIVER PHP

- ARQUITECTURA: La biblioteca "descansa" sobre el driver



DRIVER PHP

- Los documentos BSON de MongoDB se convierten en arrays asociativos de PHP

```
persona = ({  
    "nombre": "Juan Félix",  
    "tipo": "Profesor",  
    "asignaturas": ["Inglés", "Física"],  
    "horasLectivas": [  
        {"dia": "lunes", "horas": 4},  
        {"dia": "martes", "horas": 2}  
    ]  
})
```

```
$persona = array(  
    "nombre" => "Juan Félix",  
    "tipo" => "Profesor",  
    "asignaturas" =>  
        array("Inglés", "Física"),  
    "horasLectivas" => array(  
        array("dia" => "lunes", "horas" => 4),  
        array("dia" => "martes", "horas" => 2)  
    ))
```

DRIVER PHP

- Algunas de las clases más útiles de la librería son:
 - **Client:** Permite conectar con la base de datos
 - Constructor: Nos permite indicar la configuración de conexión
 - \$cliente = new MongoDB\Client("mongodb://localhost:27017")
 - Otros métodos de interés
 - listDatabases()
 - selectDatabase()
 - selectCollection()
 - **Database:** Permite gestionar las colecciones
 - listCollections()
 - selectCollection()
 - createCollection()
 - dropCollection()
 - **Collection:** Sirve para realizar operaciones CRUD y gestionar los índices
 - find(), insertOne(), updateMany(), createIndex()

DRIVER PHP

- Conectar con la base de datos
 - \$cliente = new MongoDB\Client("mongodb://localhost:27017");
- Seleccionar una base de datos
 - \$db = \$cliente->selectDatabase('centro');
- Obtener las colecciones
 - \$listaColecciones = \$db->listCollections();

```
<?php
require 'vendor/autoload.php'; // incluir el autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$listaColecciones = $db->listCollections();
foreach ($listaColecciones as $colección){
    var_dump($colección);
    echo "<br/><br/>";
}
?>
```

DRIVER PHP

- Seleccionar una colección
 - MongoDB\Database::selectCollection(nombreColeccion)
- Insertar datos
 - MongoDB\Collection::**insertOne**(\$documento)
 - Devuelve un objeto de la clase MongoDB\InsertOneResult con los siguientes métodos:
 - **isAcknowledged()**: Indica si la inserción ha sido confirmada
 - **getInsertedCount()**: Indica el número de documentos insertados
 - **getInsertedId()**: Indica el _id asignado al documento insertado

DRIVER PHP

- Insertar un profesor en la colección personas

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("personas");
$documento=array(
"Nombre" => "Ángel",
"Tipo" => "Profesor",
"Asignaturas" =>
array("Inglés", "Física"),
"HorasLectivas" => array(
array("dia"=>"lunes", "horas"=>4),
array("dia"=>"martes", "horas"=>2)
));
$resultado=$colección->insertOne($documento);
if($resultado->isAcknowledged() && $resultado->getInsertedCount()==1){
echo "El _id del documento es ".$resultado->getInsertedId();
}else{
echo "No se pudo insertar el documento";
}
?>
```

DRIVER PHP

- Buscar documentos
 - MongoDB\Collection::find(\$filtro=[], \$opciones=[])
 - Las opciones son:
 - **projection**
 - **sort**
 - **skip**
 - **limit**
 - Devuelve un objeto de tipo cursor: MongoDB\Driver\Cursor, que es un array

DRIVER PHP

- Mostrar los nombres de los primeros 2 profesores ordenados por el nombre

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("personas");
$consulta=array("Tipo"=>"Profesor");
$orden=array("sort"=>array("Nombre",1));
$proyección=array("Nombre"=>1);
$resultado=$colección->find($consulta,array($orden,
$proyección,"limit"=>2));
foreach($resultado as $profesor){
print_r($profesor);
echo "<br/><br/>";
}
?>
```

DRIVER PHP

- Aggregation pipeline
 - MongoDB\Collection::aggregate(array \$pipeline)
 - Devuelve un cursor, que podemos convertir en un array con el método toArray()
 - \$cursor->toArray()
 - Cuidado con las comillas dobles y el signo de \$ en PHP
- ¿Cuántas ciudades hay en EEUU?

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("zips");
$etapa1=array('$group'=>array("_id"=>'$city'));
$etapa2=array('$count'=>'numeroCiudades');
$resultado=$colección->aggregate(array($etapa1,$etapa2));
$resultado=$resultado->toArray();
print_r($resultado[0]);
?>
```



DRIVER PHP

- Ejercicio
 - Averiguar cuántas ciudades hay en el estado de Massachusetts - MA

DRIVER PHP

- Averiguar cuántas ciudades hay en el estado de Massachusetts [MA]

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("zips");
$etapa1=array('$match'=>array('state'=>"MA"));
$etapa2=array('$group'=>array("_id"=>'$city'));
$etapa3=array('$count'=>'numeroCiudades');
$resultado=$colección-
>aggregate(array($etapa1,$etapa2,$etapa3));
$resultado=$resultado->toArray();
print_r($resultado[0]);
?>
```



DRIVER PHP

- Ejercicio
 - Mostrar las ciudades de EEUU con más de 1.000.000 habitantes ordenadas por población

DRIVER PHP

- Mostrar las ciudades de EEUU con más de 1.000.000 de habitantes ordenadas por población ascendente

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("zips");
$etapa1=array('$group'=>array('_id'=>'$city',"poblacion"=>array(
'$sum'=>'$pop')));
$etapa2=array('$match'=>array('poblacion'=>array('$gt'=>1000000)))
);
$etapa3=array('$sort'=>array('poblacion'=>1));
$resultado=$colección->aggregate(array($etapa1,$etapa2,$etapa3));
foreach($resultado as $ciudad){
print_r($ciudad);
echo '<br/>';
}
?>
```



DRIVER PHP

- Ejercicio
 - Mostrar todos los códigos postales (zips) de EEUU con más de 50.000 habitantes ordenadas por ciudad y zip

DRIVER PHP

- Mostrar todos los códigos postales (zips) de EEUU con más de 50.000 habitantes ordenadas por ciudad y zip

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("zips");
$proyección=array('city'=>1,'pop'=>1);
$orden=array('city'=>1,'_id'=>1);
$resultado=$colección-
>find(array('pop'=>array('$gt'=>50000)),array('projection'=>$pro
yección,'sort'=>$orden));
foreach($resultado as $ciudad){
print_r($ciudad);
echo '<br/>';
}
?>
```

DRIVER PHP

- Actualización de documentos
 - MongoDB\Collection::updateOne(\$filtro=[], \$actualización=[], \$opciones)
 - En \$actualización se pueden usar los operadores de actualización de MongoDB
 - \$set
 - \$setOnInsert: Realiza la asignación sólo si la operación implica la creación de un documento nuevo (véase upsert)
 - \$inc: Incrementa o decrementa el valor de un campo
 - \$rename: Cambia el nombre del campo
 - \$: Primer elemento de un array
 - Algunas \$opciones son
 - upsert: Si se configura como true crea un documento nuevo si no existe ninguno que cumpla el criterio.
 - arrayFilters: Un array de filtros que deben cumplir los valores de un array para ser modificados por \${<identificador_Array_filter>}]

DRIVER PHP

- Actualización de documentos
 - Cambiar el nombre del profesor "Juan Félix" por "Juanfe"

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("personas");
$criterio=array('Nombre'=>'Juan Félix','Tipo'=>'Profesor');
$resultado=$colección-
>updateOne($criterio,array('$set'=>array('Nombre'=>'Juanfe')));
print_r($resultado);
?>
```

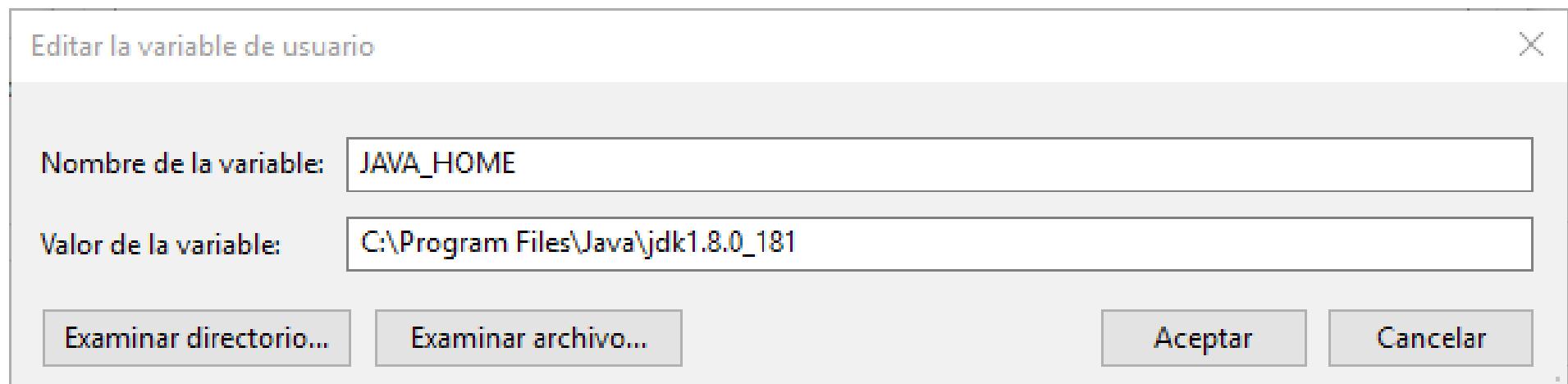
DRIVER PHP

- Actualización de documentos
 - Modificar las nóminas de los administrativos para que su cuantía máxima sea 1.500€
 - Recordar que el identificador (mensualidad) tiene que empezar por minúscula

```
<?php
require 'vendor/autoload.php'; // Incluir autoloader de
Composer
$cliente = new MongoDB\Client("mongodb://localhost:27017");
$db = $cliente->selectDatabase('centro');
$colección=$db->selectCollection("personas");
$criterio=array('Tipo'=>'Administrativo');
$filterArray=array('arrayFilters'=>array(array('mensualidad.Cantidad'=>array('$gt'=>1500)))); 
$resultado=$colección-
>updateMany($criterio,array('$set'=>array('Nominas.$[mensualidad].Cantidad'=>1500)), $filterArray);
print_r($resultado);
?>
```

DRIVER JAVA

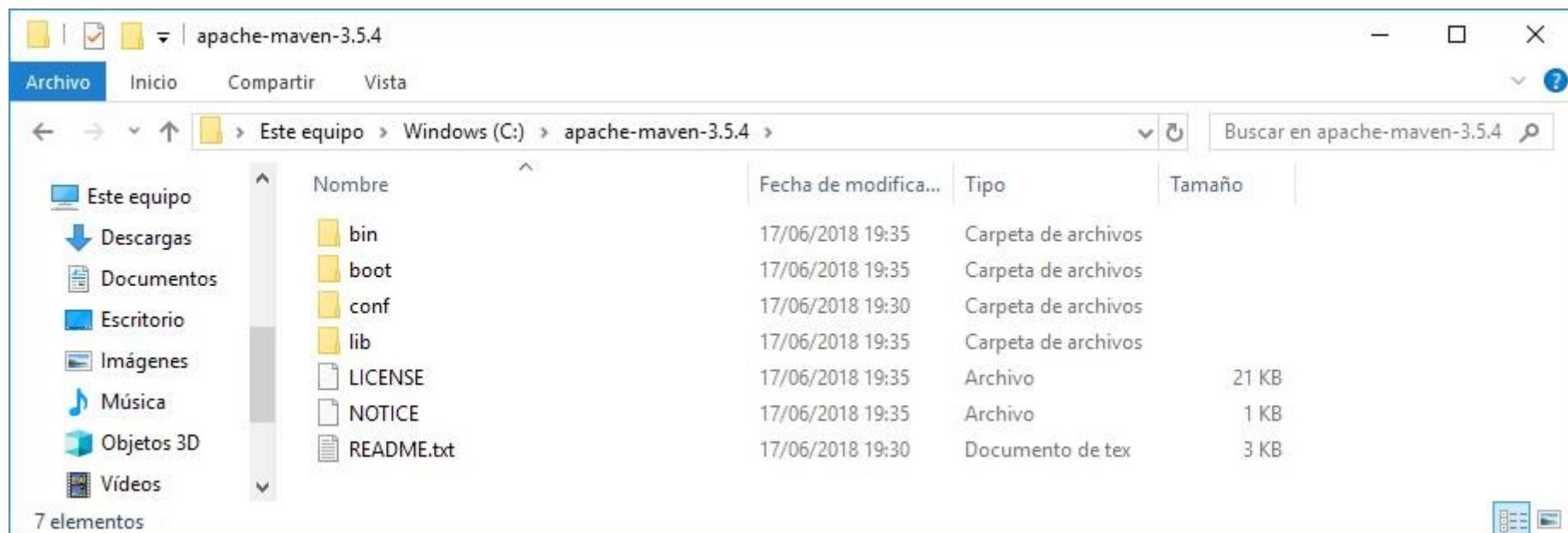
- INSTALAR JAVA
 - La versión actual de Eclipse requiere Java 1.7 o Java 1.8
 - Agregar la variable de usuario JAVA_HOME con la ruta en la que hemos instalado Java
 - Recordar que los cambios no son efectivos hasta que no se pulsa Aceptar en el cuadro de diálogo Variables de entorno



DRIVER JAVA

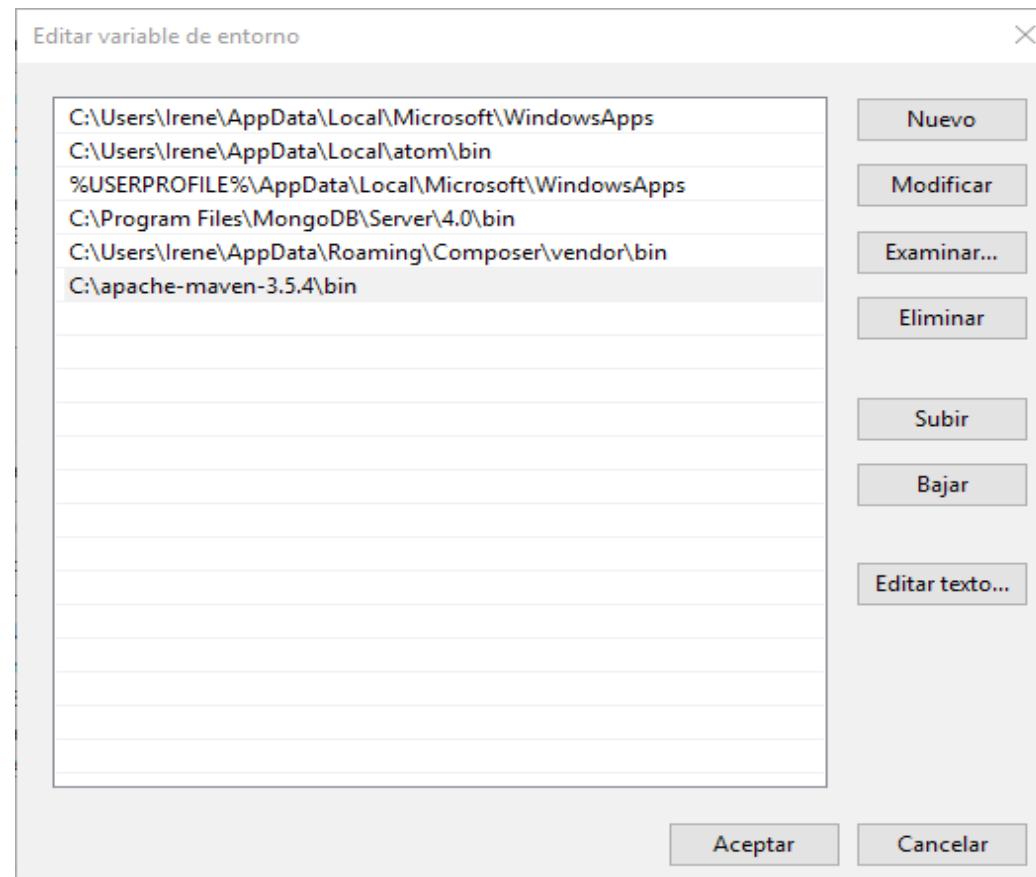
- INSTALAR MAVEN

- <http://apache.rediris.es/maven/maven-3/3.5.4/binaries/apache-maven-3.5.4-bin.zip>
- Descomprimir en cualquier carpeta del sistema; por ejemplo en C:



DRIVER JAVA

- INSTALAR MAVEN
 - Agregar la ruta de la carpeta bin de Maven a la variable del sistema PATH del usuario



DRIVER JAVA

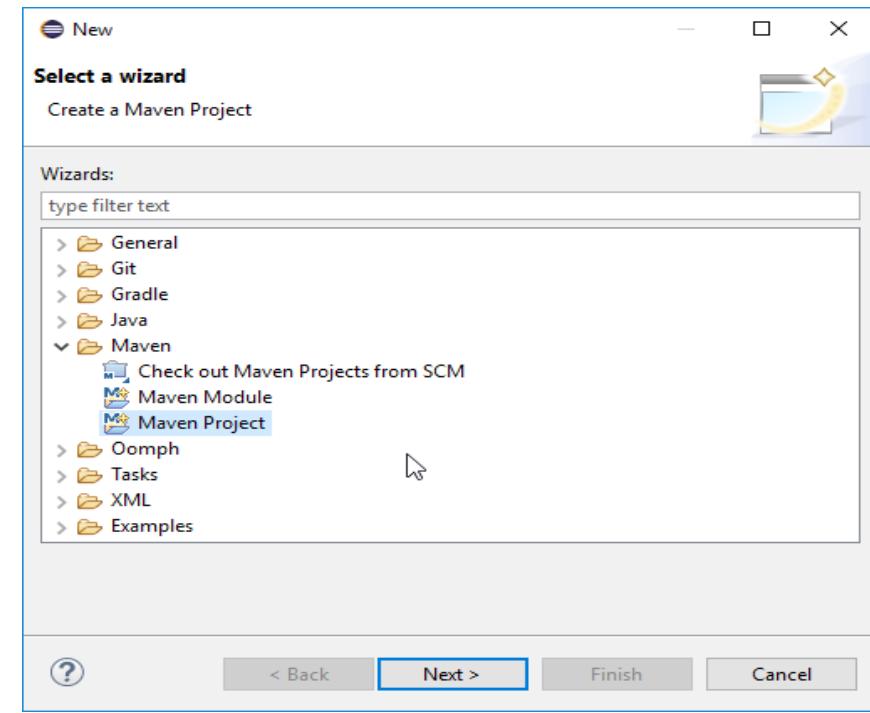
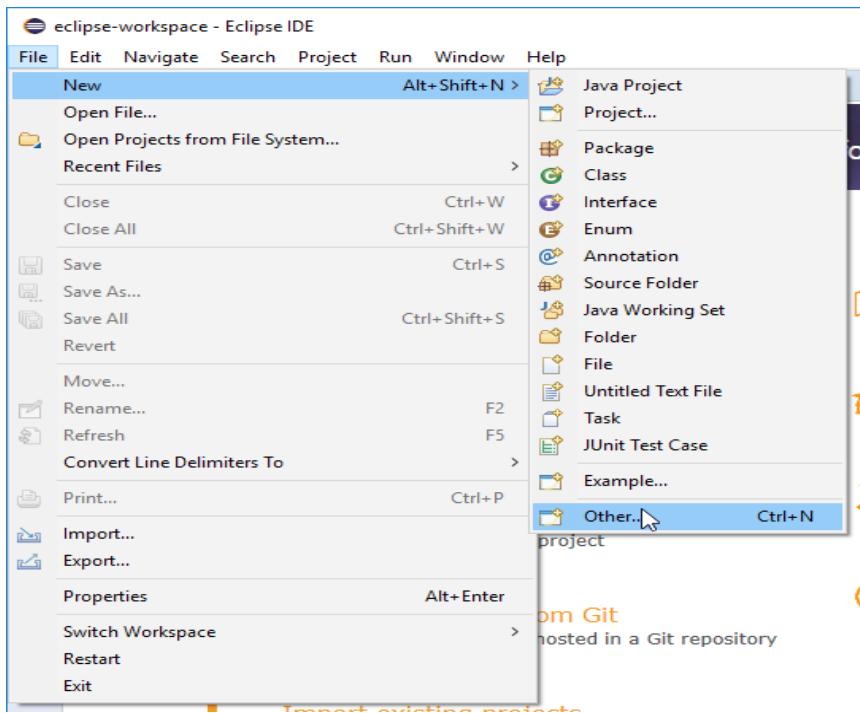
- INSTALAR MAVEN
 - Comprobar que Maven se ha instalado correctamente abriendo una consola del sistema nueva (para que se recarguen las variables del sistema) y ejecutando
 - mvn -v

```
C:\>mvn -v
C:\>
Apache Maven 3.5.4 (1edded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-17T20:33:14+02:00)
Maven home: C:\apache-maven-3.5.4\bin\..
Java version: 1.8.0_181, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_181\jre
Default locale: es_ES, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

C:\>
```

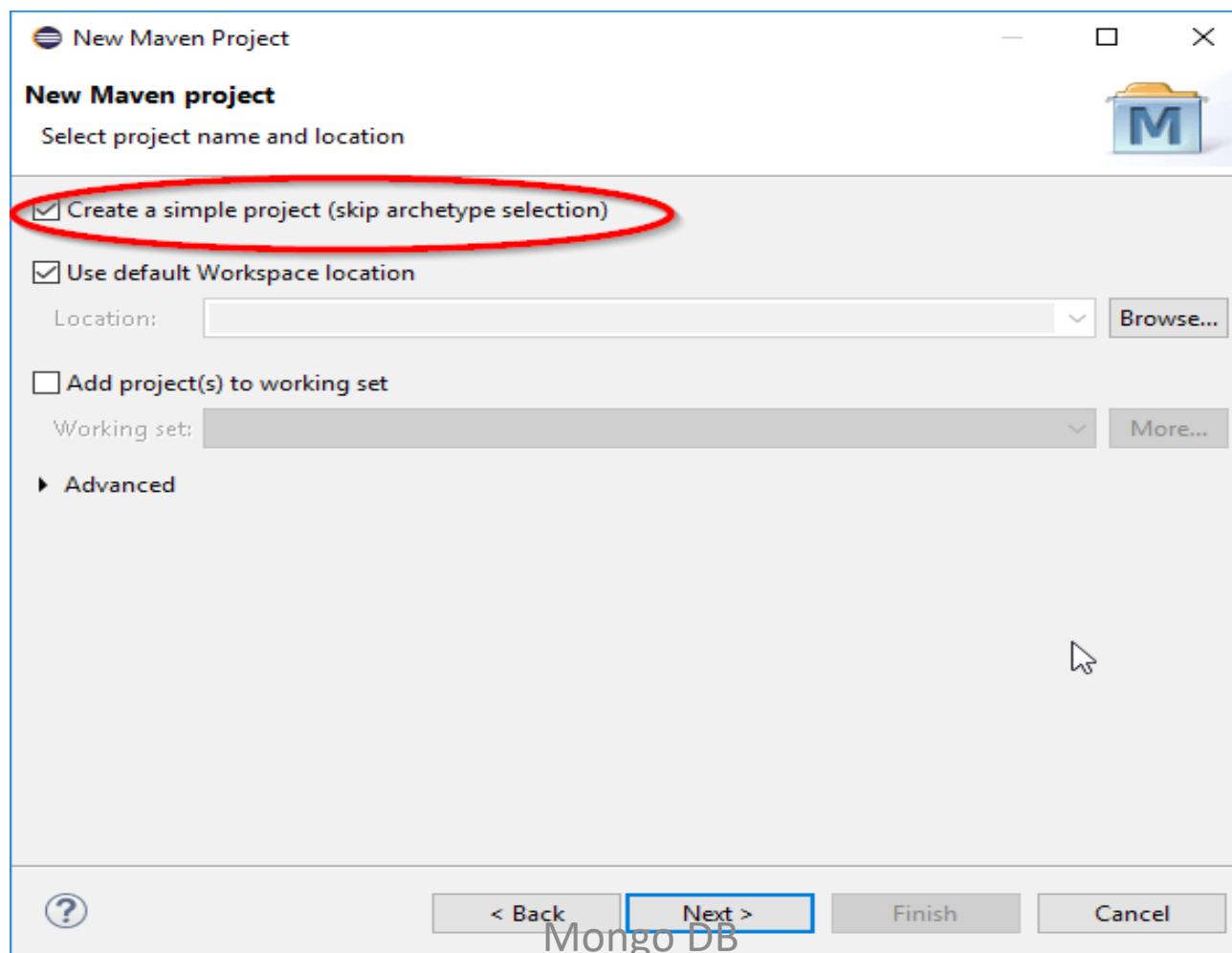
DRIVER JAVA

- INSTALAR ECLIPSE para Java
- Iniciar un proyecto nuevo en Eclipse
 - File > New > Other
 - Maven > Maven project



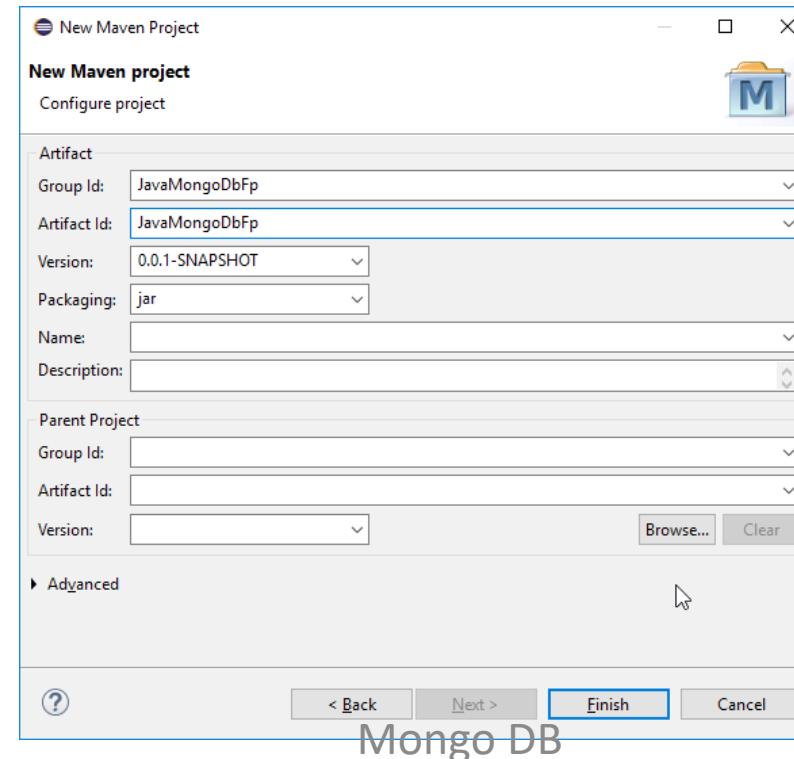
DRIVER JAVA

- Iniciar un proyecto nuevo en Eclipse
 - Activar la casilla Create a simple project



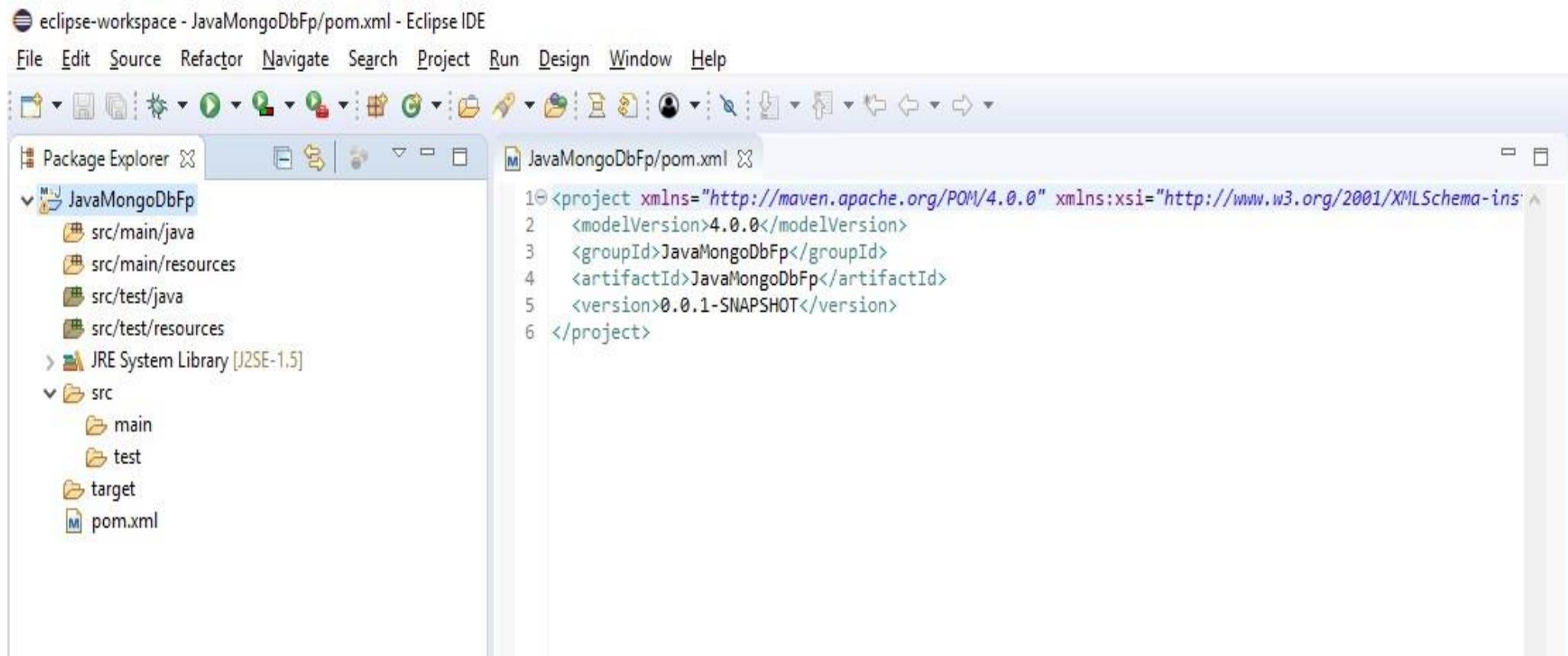
DRIVER JAVA

- Iniciar un proyecto nuevo en Eclipse
 - Configurar el artefacto del proyecto
 - Group id: JavaMongoDbFp
 - Artifact id: JavaMongoDbFp



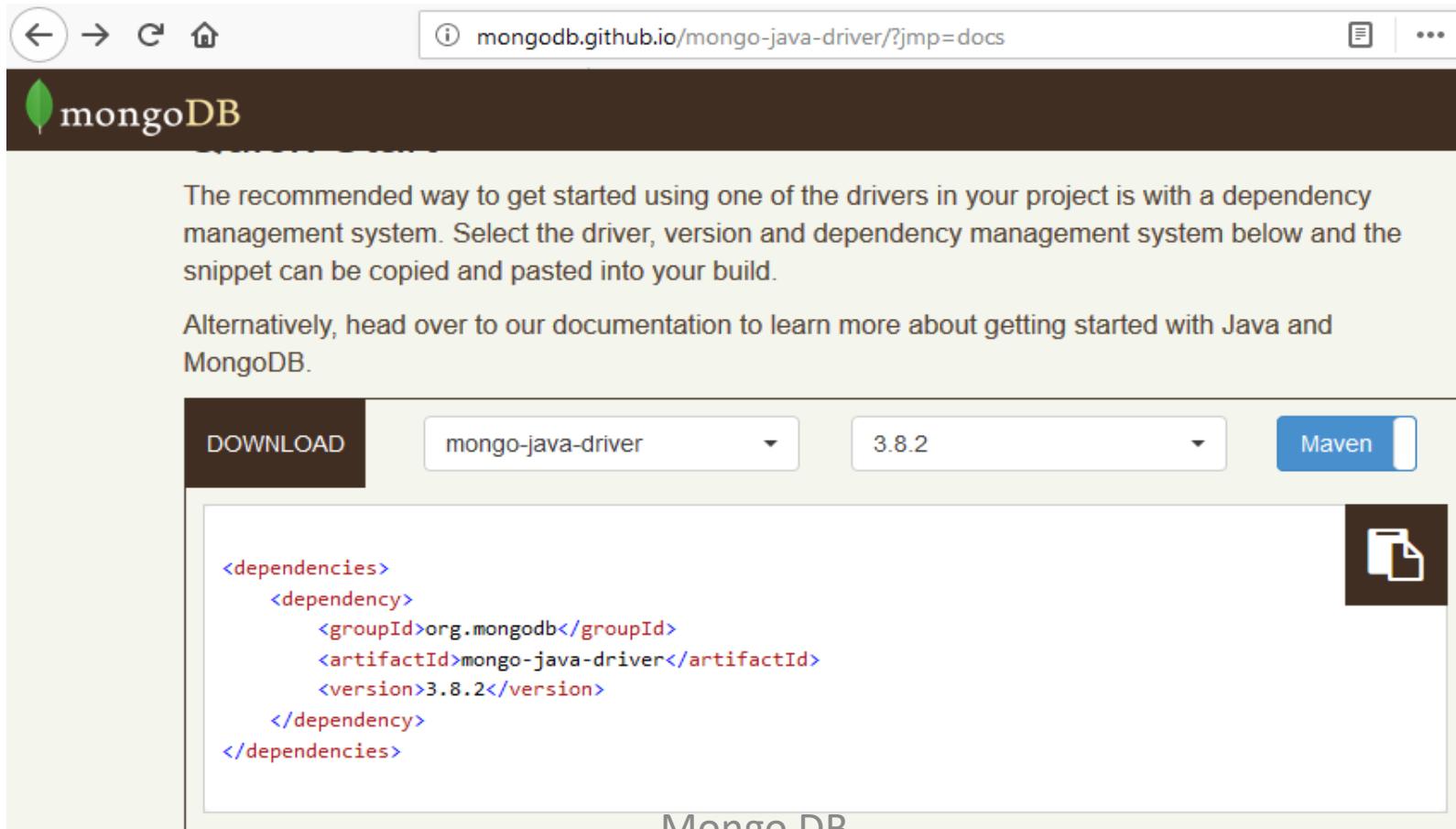
DRIVER JAVA

- Comprobar que se ha creado el archivo pom.xml



DRIVER JAVA

- Completar el archivo pom.xml con las dependencias que nos indica esta página (<http://mongodb.github.io/mongo-java-driver/?jmp=docs>) al elegir mongo-java-driver con Maven



The recommended way to get started using one of the drivers in your project is with a dependency management system. Select the driver, version and dependency management system below and the snippet can be copied and pasted into your build.

Alternatively, head over to our documentation to learn more about getting started with Java and MongoDB.

DOWNLOAD mongo-java-driver 3.8.2 Maven

```
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.8.2</version>
  </dependency>
</dependencies>
```

Mongo DB

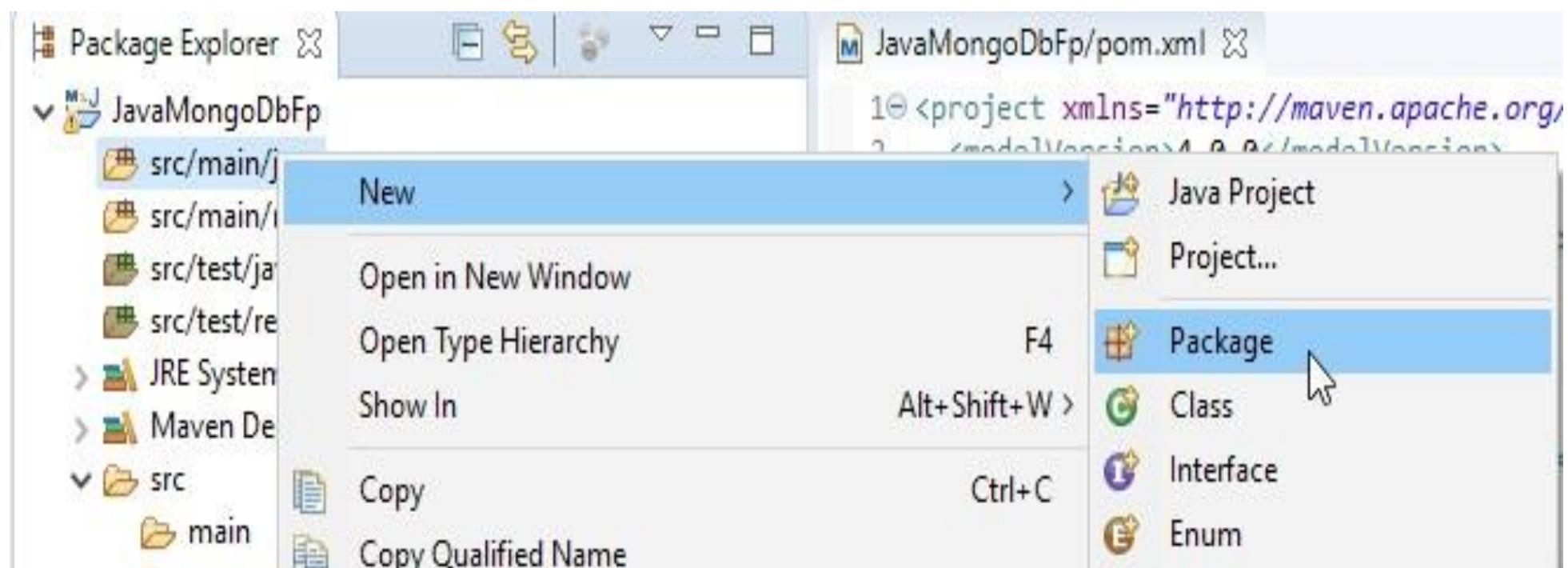
DRIVER JAVA

- El archivo pom.xml quedará del siguiente modo

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>JavaMongoDbFp</groupId>
    <artifactId>JavaMongoDbFp</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongo-java-driver</artifactId>
            <version>3.8.2</version>
        </dependency>
    </dependencies>
</project>
```

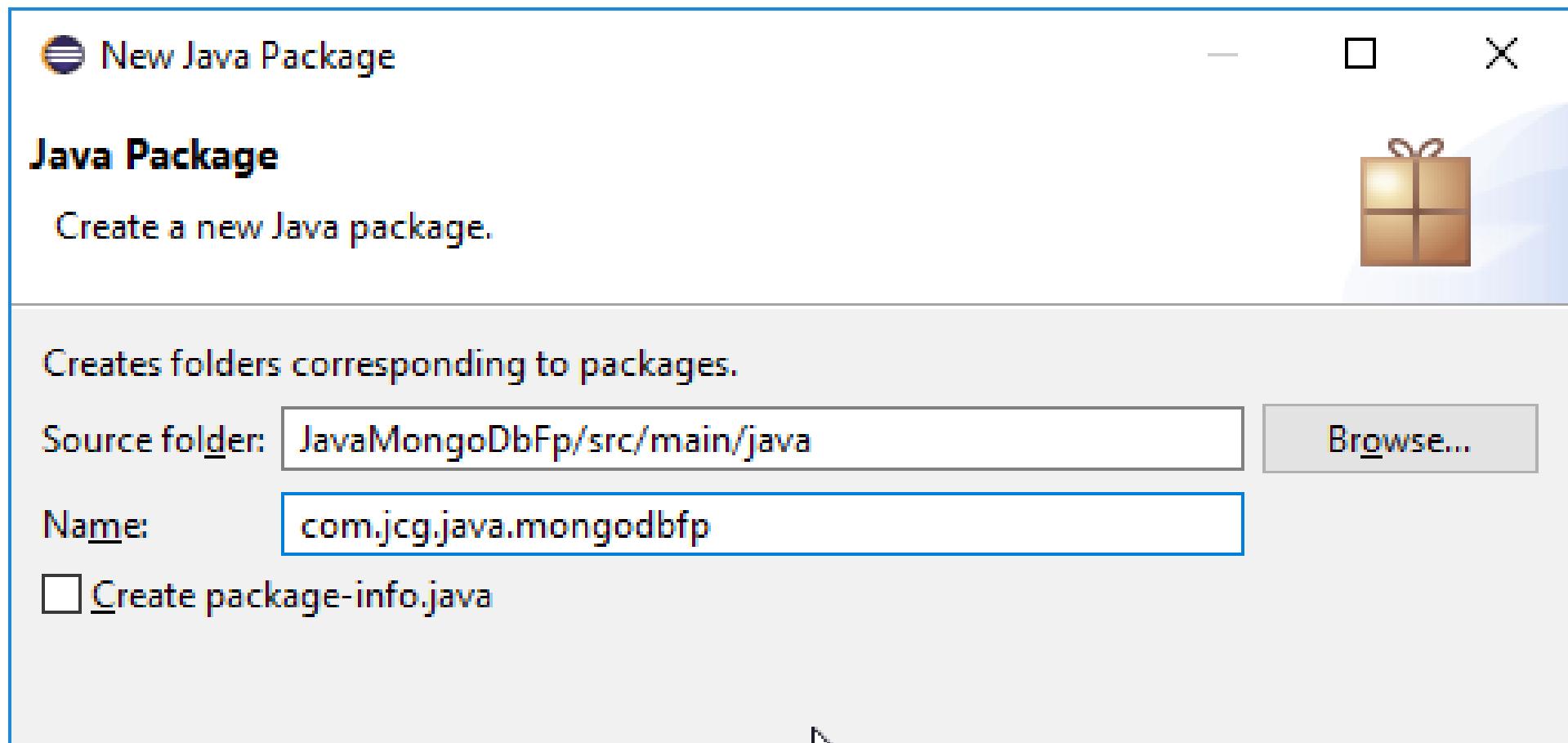
DRIVER JAVA

- Crear un nuevo paquete en src/main/java



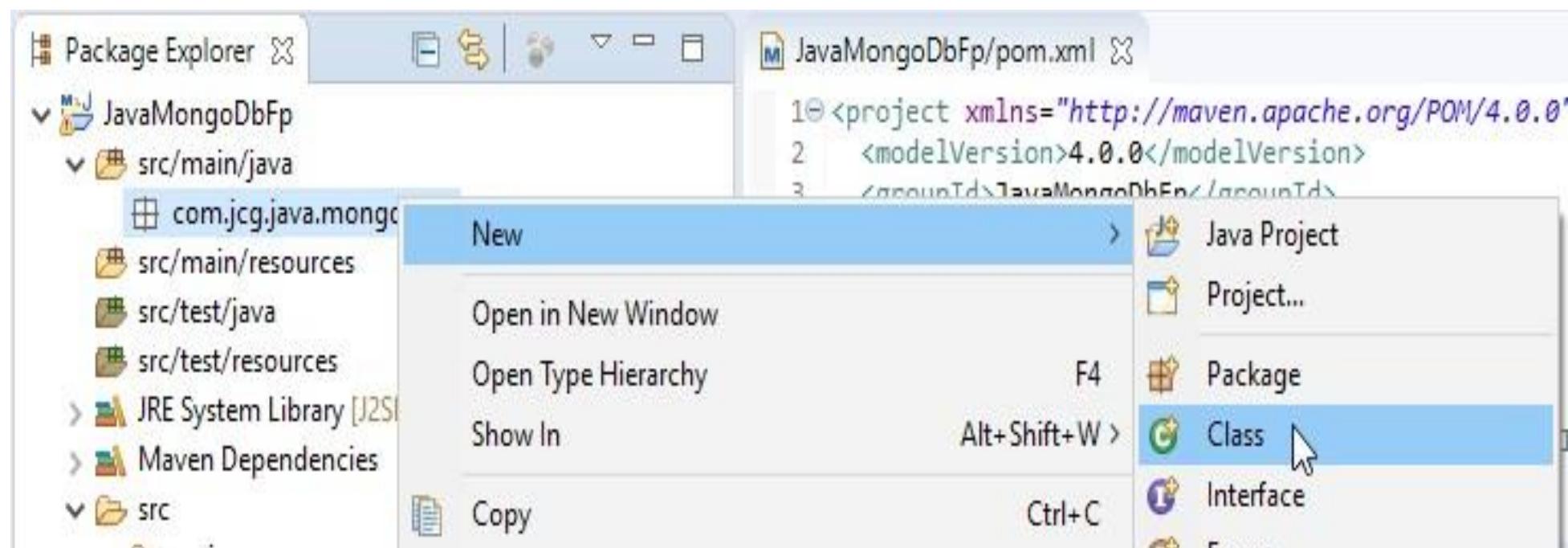
DRIVER JAVA

- Especificar el nombre del paquete
 - com.jcg.java.mongodbfp



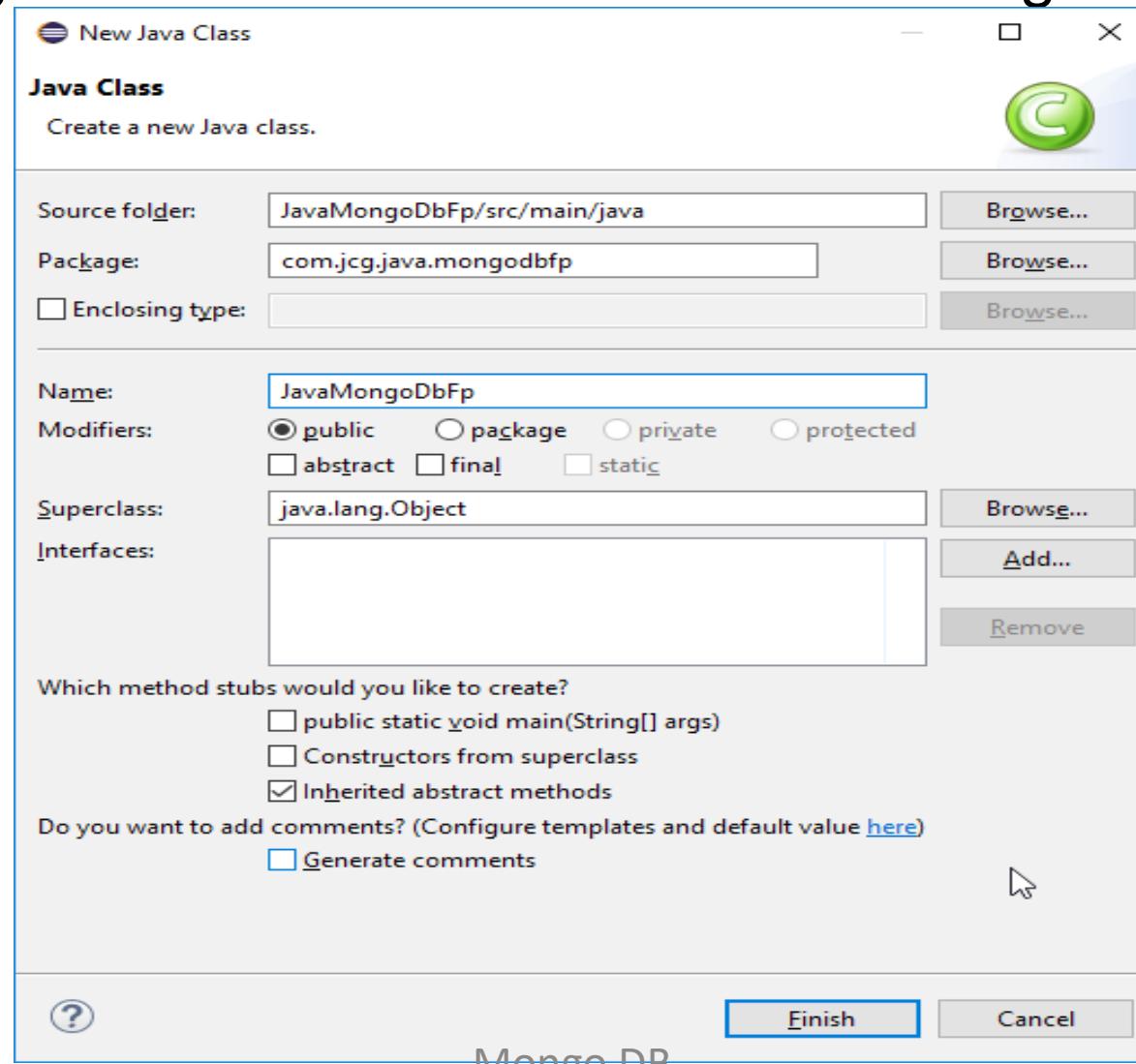
DRIVER JAVA

- Crear una clase dentro del paquete anterior



DRIVER JAVA

- Asignar a la clase el nombre JavaMongoDbFp



DRIVER JAVA

- Conectar con la base de datos

```
MongoClient clienteMongo = MongoClients.create();
```

- Activar una base de datos

```
MongoDatabase baseDatos = clienteMongo.getDatabase("centro");
```

- Obtener un listado de las colecciones

```
baseDatos.listCollectionNames()
```

- Cerrar la conexión

```
clienteMongo.close();
```

DRIVER JAVA

```
package com.jcg.java.mongodbfp;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoDatabase;
public class JavaMongoDbFp {
    public static void main( String args[] ) {
        MongoClient clienteMongo = MongoClients.create();
        MongoDatabase baseDatos = clienteMongo.getDatabase("centro");
        for (String name : baseDatos.listCollectionNames())
            { System.out.println(name);
        }
        clienteMongo.close();
    }
}
```

DRIVER JAVA

- Seleccionar una colección

```
MongoCollection colección =  
database.getCollection("personas");
```

- Buscar el primer documento de una colección

```
Document primero =  
colección.find().first();
```

```
package com.jcg.java.mongodbfp;  
  
import com.mongodb.*;  
import com.mongodb.client.MongoClients;  
import com.mongodb.client.MongoClient; import  
com.mongodb.client.MongoCollection; import  
com.mongodb.client.MongoDatabase; import  
com.mongodb.client.model.Projections; import  
com.mongodb.client.model.Filters;  
import static com.mongodb.client.model.Filters.*;  
import static com.mongodb.client.model.Projections.*;  
import com.mongodb.client.model.Sorts;  
import java.util.Arrays;  
import org.bson.Document;  
  
public class JavaMongoDbFp {  
public static void main( String args[] ) {  
MongoClient clienteMongo = MongoClients.create();  
MongoDatabase baseDatos = clienteMongo.getDatabase("centro");  
MongoCollection<Document> colección =  
baseDatos.getCollection("personas");  
Document primero = colección.find().first();  
System.out.println(primerito.toJson());  
clienteMongo.close();  
}  
}
```

DRIVER JAVA

- Realizar una consulta

```
FindIterable<Document> iterDoc =  
colección.find(new Document("tipo",  
"Alumno"));
```

```
package com.jcg.java.mongodbfp;  
  
import com.mongodb.*;  
import com.mongodb.client.FindIterable; import  
com.mongodb.client.MongoClients; import  
com.mongodb.client.MongoClient; import  
com.mongodb.client.MongoCollection; import  
com.mongodb.client.MongoDatabase; import  
com.mongodb.client.model.Projections; import  
com.mongodb.client.model.Filters;  
import static com.mongodb.client.model.Filters.*;  
import static com.mongodb.client.model.Projections.*;  
import com.mongodb.client.model.Sorts;  
import java.util.Arrays;  
import java.util.Iterator;  
import org.bson.Document;  
  
  
public class JavaMongoDbFp {  
public static void main( String args[] ) {  
MongoClient clienteMongo = MongoClients.create();  
MongoDatabase baseDatos = clienteMongo.getDatabase("centro");  
MongoCollection<Document> colección =  
baseDatos.getCollection("personas");  
//Obtener un objeto iterable de tipo cursor  
FindIterable<Document> iterDoc = colección.find(new  
Document("tipo", "Alumno"));  
int i = 1;  
//Iniciamos el iterador  
Iterator it = iterDoc.iterator();  
while (it.hasNext()) {  
    System.out.println(it.next());  
    i++;}  
clienteMongo.close();  
}
```

DRIVER JAVA

- Insertar un documento

```
Document persona = new Document("nombre", "Sergio")
    .append("tipo",
"Profesor")

.append("datos_personales", new Document("telefono", "123456789")
    .append("email",
"su@correo.es"))

.append("asignaturas",
Arrays.asList("Física",
"Matemáticas"));
colección.insertOne(persona);
```

```
package com.jcg.java.mongodbfp;

import com.mongodb.*;
import com.mongodb.client.FindIterable; import
com.mongodb.client.MongoClients; import
com.mongodb.client.MongoClient; import
com.mongodb.client.MongoCollection; import
com.mongodb.client.MongoDatabase; import
com.mongodb.client.model.Projections; import
com.mongodb.client.model.Filters;
import static com.mongodb.client.model.Filters.*;
import static com.mongodb.client.model.Projections.*;
import com.mongodb.client.model.Sorts;
import java.util.Arrays;
;

import java.util.Iterator
import org.bson.Document;

public class JavaMongoDbFp {
public static void main( String args[] ) { MongoClient
clienteMongo = MongoClients.create();
MongoDatabase baseDatos = clienteMongo.getDatabase("centro");
MongoCollection<Document> colección =
baseDatos.getCollection("personas");
Document persona = new Document("nombre", "Sergio")
    .append("tipo", "Profesor")
    .append("datos_personales", new Document("telefono",
"123456789"))
    .append("email", "su@correo.es"))

.append("asignaturas", Arrays.asList("Física",
"Matemáticas"));
colección.insertOne(persona);
clienteMongo.close();
}
}
```