

MongoDB

*Máster en Business Intelligence e
Innovación Tecnológica*

Índice de contenidos

1. Introducción
2. Arquitectura
3. Modelado de datos
4. Replicación
5. Operaciones CRUD
6. Índices
7. Sintaxis
8. Ejemplos
9. Agregación
10. MEAN
11. MongoDB y Python

Introducción

- Similar a CouchDB
- Pretende combinar lo mejor de los almacenes clave/valor, bases de datos de documentos y RDBMS
- Hace uso de JSON y tiene su propio lenguaje de consultas
- Implementada en C++
- MongoDB (de la palabra en ingles “humongous” que significa enorme) es un sistema de base de datos NoSQL orientado a documentos
 - MongoDB guarda estructuras de datos en documentos tipo BSON (Binary JSON (JSON Binario) con un esquema dinámico , haciendo que la integración de los datos en ciertas aplicaciones sea mas fácil y rápida.
- Lo que MongoDB no puede hacer:
 - No hay tablas de BBDD
 - No hay “joins”
 - No hay transacciones
 - MongoDB no usa esquemas de datos





Características

- Consultas Ad hoc
 - MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares.
 - Las consultas pueden devolver un campo específico del documento pero también puede ser una función JavaScript definida por el usuario.
- Indexación
 - Cualquier campo en un documento de MongoDB puede ser indexado, al igual que es posible hacer índices secundarios.
 - El concepto de índices en MongoDB es similar a los encontrados en base de datos relacionales. con el maestro actual.

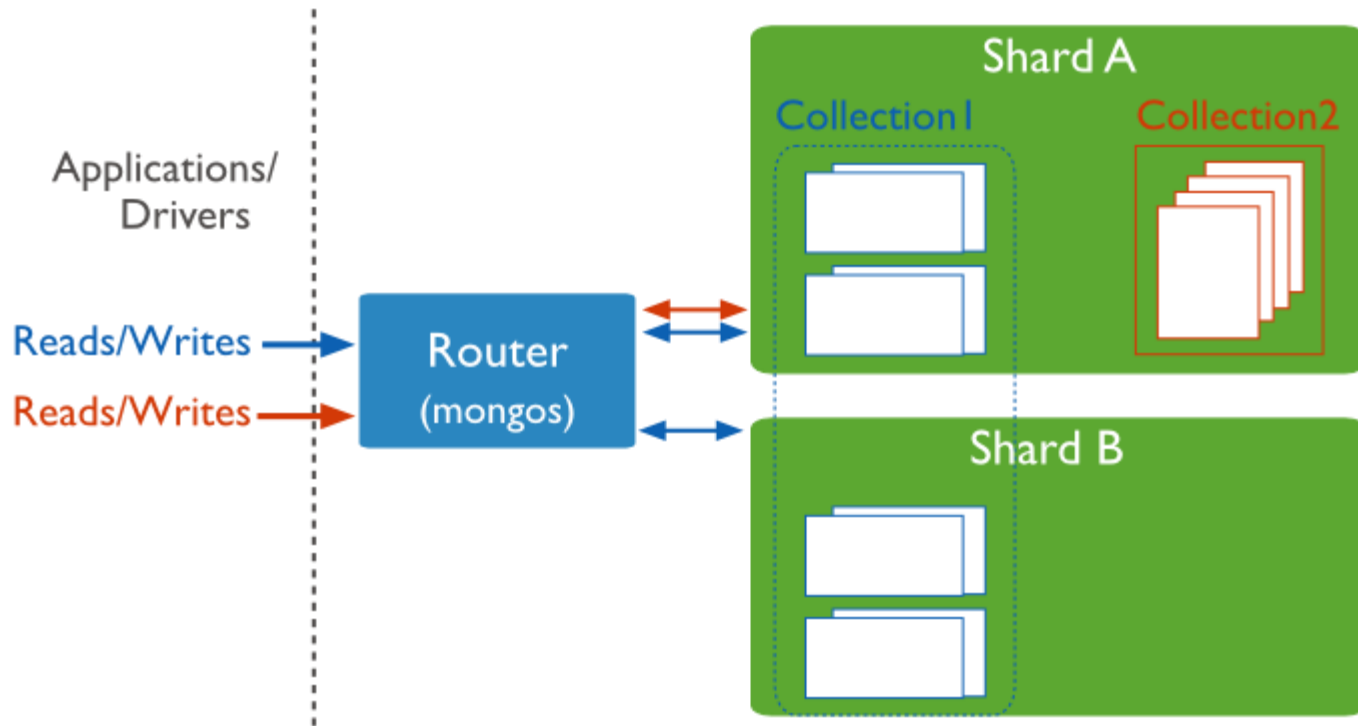
Características

- Replicación
 - MongoDB soporta el tipo de replicación maestro-esclavo.
 - El maestro puede ejecutar comandos de lectura y escritura.
 - El esclavo puede copiar los datos del maestro y sólo se puede usar para lectura o para copia de seguridad, pero no se pueden realizar escrituras.
 - El esclavo tiene la habilidad de poder elegir un nuevo maestro en caso del que se caiga el servicio con el maestro actual.

Características

- Sintaxis: basada en el javascript
 - Se convierte en una de las base de datos favoritas para la comunidad de desarrolladores en ese idioma
 - Presente en el stack MEAN
- Balanceo de carga
 - MongoDB se puede escalar de forma horizontal usando el concepto de “shard”.
 - El desarrollador elige una llave shard, la cual determina cómo serán distribuidos los datos en una colección. Los datos son divididos en rangos (basado en la llave shard) y distribuidos a través de múltiples shard.
 - Un shard es un maestro con uno o más esclavos.
 - MongoDB tiene la capacidad de ejecutarse en múltiple servidores, balanceando la carga y/o duplicando los datos para poder mantener el sistema funcionando en caso que exista un fallo de hardware.

Características



Características

- Almacenamiento de archivos
 - MongoDB puede ser utilizado con un sistema de archivos, tomando la ventaja de la capacidad que tiene MongoDB para el balanceo de carga y la replicación de datos utilizando múltiples servidores para el almacenamiento de archivos.
 - Esta función (que es llamada GridFS) está incluida en los drivers de MongoDB y disponible para los lenguajes de programación que soporta MongoDB.
- Agregación
 - La función MapReduce y el operador aggregate() puede ser utilizada para el procesamiento por lotes de datos y operaciones de agregación.
 - Estos mecanismos permiten que los usuarios puedan obtener el tipo de resultado que se obtiene cuando se utiliza el comando SQL “group-by”.

Teorema CAP

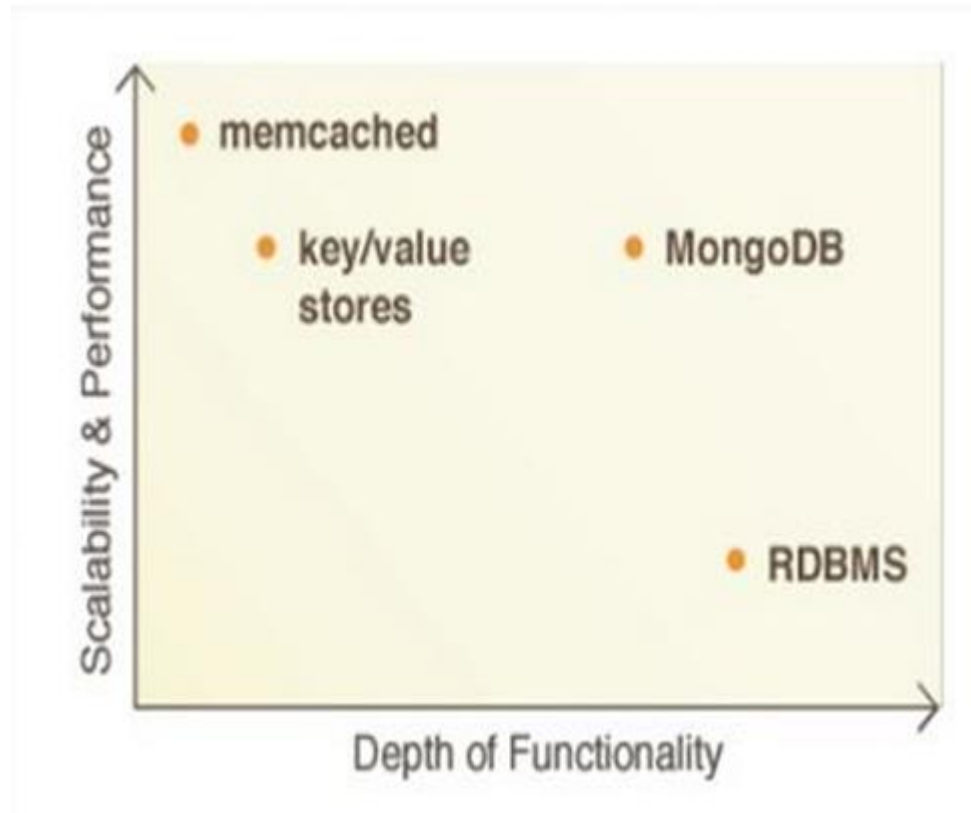
- Por ejemplo MongoDB es CP por defecto.
- También podemos configurar el nivel de consistencia, eligiendo el número de nodos a los que se replicarán los datos.
- Por otro lado podemos configurar si se pueden leer datos de los nodos secundarios (en MongoDB solo hay un servidor principal, que es el único que acepta inserciones o modificaciones). Si permitimos leer de un nodo secundario, sacrificamos consistencia, pero ganamos disponibilidad.

Casos de uso

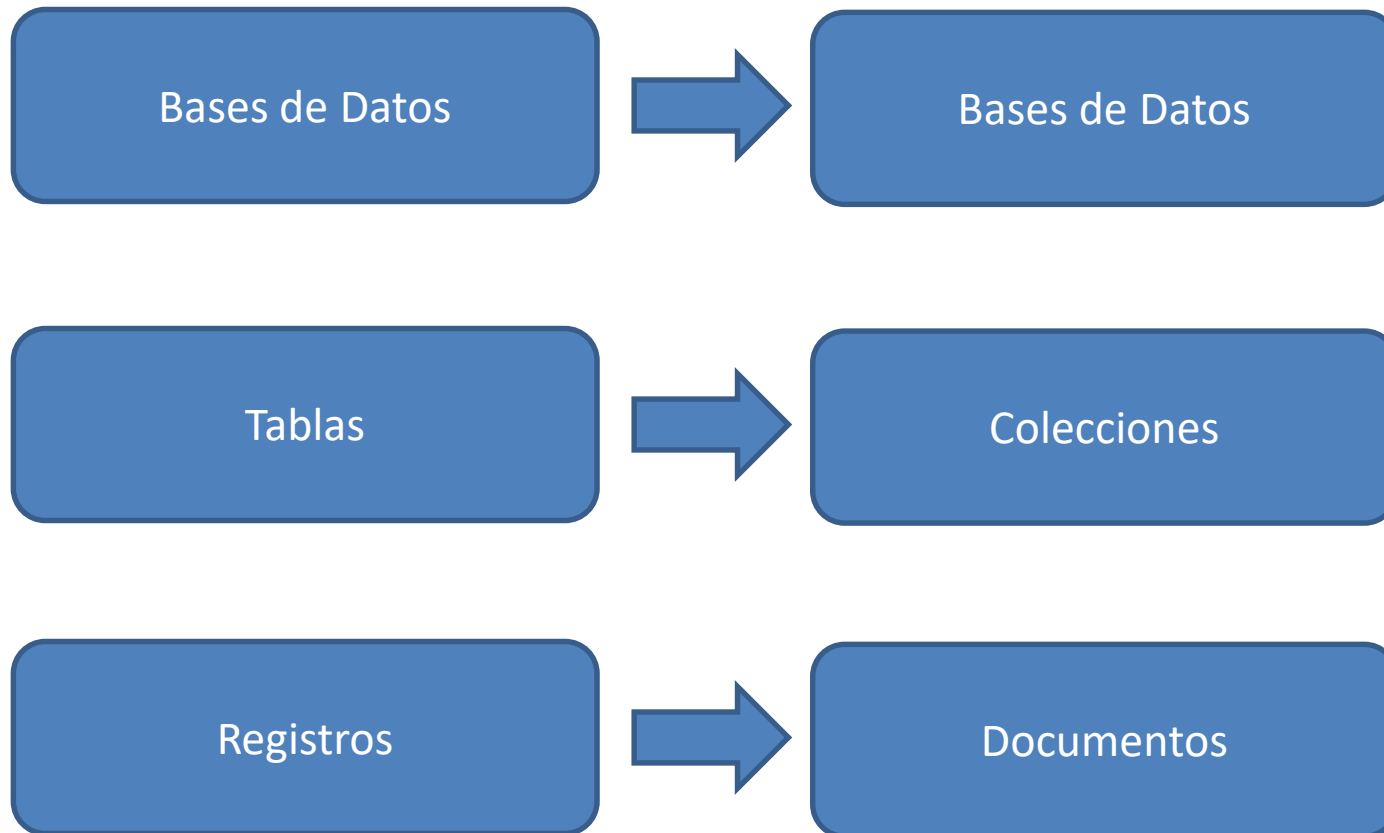
- Almacenamiento y registro de eventos
- Para sistemas de manejo de documentos y contenido
- Comercio Electrónico
- Juegos
- Problemas de alto volumen
- Aplicaciones móviles
- Almacén de datos operacional de una página Web
- Manejo de contenido
- Almacenamiento de comentarios
 - Votaciones
 - Registro de usuarios
 - Perfiles de usuarios
 - Sesiones de datos
- Proyectos que utilizan metodologías de desarrollo iterativo o ágiles
- Manejo de estadísticas en tiempo real

Rendimiento

- Buen equilibrio entre velocidad y funcionalidad

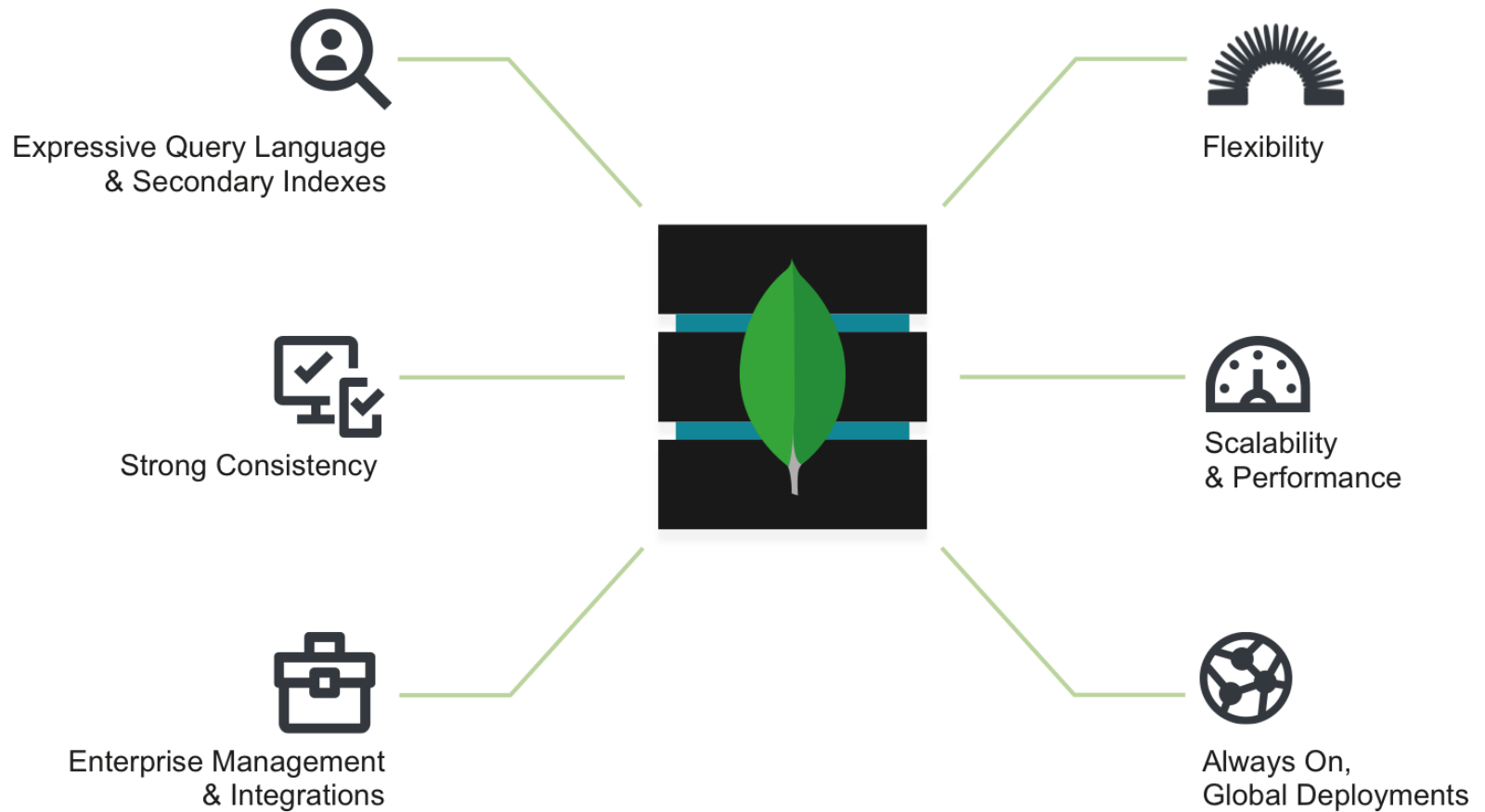


Conceptos



Colecciones y documentos

- MongoDB guarda la estructura de los datos en documentos tipo JSON con un esquema dinámico llamado BSON, lo que implica que no existe un esquema predefinido.
- Los elementos de los datos son llamados documentos y se guardan en colecciones
- Una colección puede tener un número indeterminado de documentos
 - Las colecciones son como tablas y los documentos como filas
 - Cada documento en una colección puede tener diferentes campos.
- La estructura de un documento es simple y compuesta por “key-value pairs” parecido a las matrices asociativas en un lenguaje de programación
 - Como valor se pueden usar números, cadenas o datos binarios como imágenes o cualquier otro “key-value pairs”.



Colecciones y documentos

- MongoDB tiene el concepto de “base de datos” con el que estamos familiarizados (schema en el mundo relacional).
 - Dentro de un servidor MongoDB podemos tener 0 o más BBDD, cada una actuando como un contenedor de todo lo demás.
- Una base de datos puede tener una o más “colecciones”, equivalente en el mundo relacional a una “tabla”.
- Las colecciones están hechas de 0 o más “documentos”, donde un documento puede considerarse equivalente a una fila de una tabla de un RDBMS.
- Un documento está compuesto de uno o varios “campos” que son equivalentes a las columnas de una fila.
- Los “índices” en MongoDB funcionan como los de las RDBMS.
- Los “cursores” son utilizados para acceder progresivamente a los datos recuperados con una consulta
 - Pueden usarse para contar o moverse hacia delante entre los datos

JSON

```
{
  "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
  "Last Name": "PELLERIN",
  "First Name": "Franck",
  "Age": 29,
  "Address": {
    "Street": "1 chemin des Loges",
    "City": "VERSAILLES"
  }
}
```

- Acronimo de JavaScript Object Notation es un tipo de documentos dedicado al intercambio de información
- No es de extrañar su elección dada la fuerte vinculación de MongoDB con el JS.

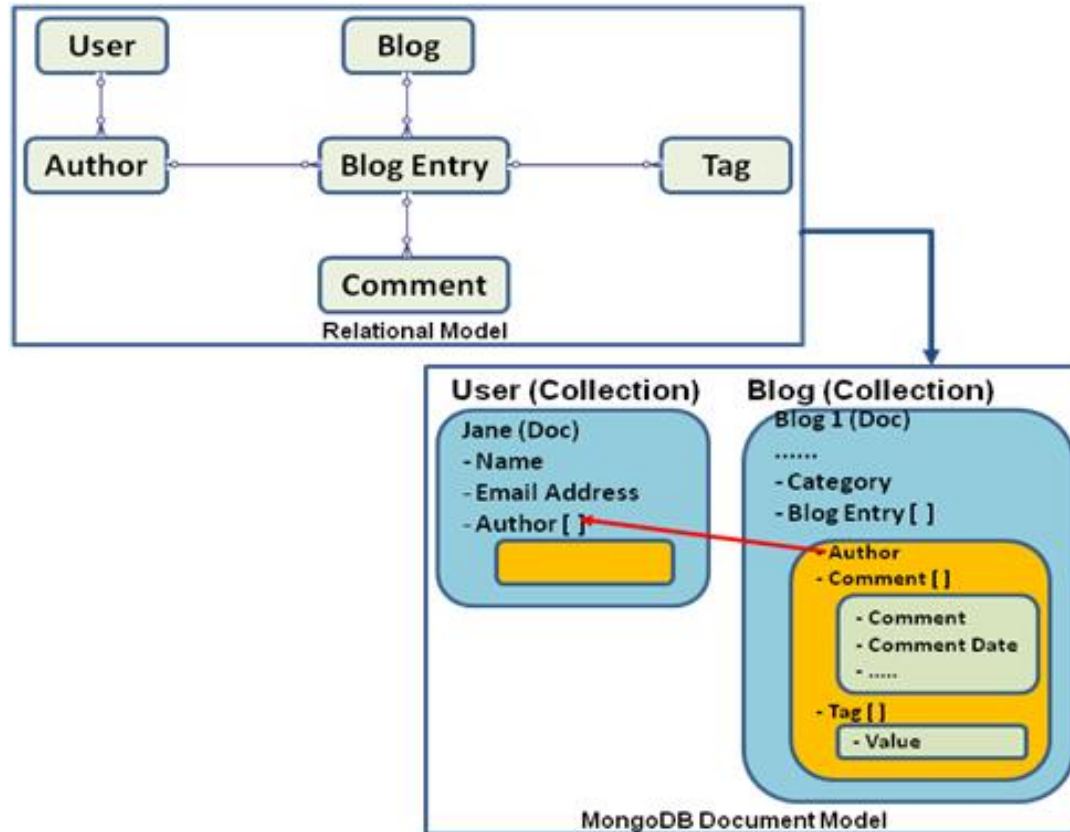
BSON

- BSON versión binaria de JSON, serialización codificada en binario de documentos JSON
 - Soporta colocar documentos dentro de un documento y arrays dentro de otros documentos y arrays
 - Contiene extensiones que permiten representar tipos de datos que no son parte de JSON, más anclado a cadenas String
 - Por ejemplo, BSON tiene los tipos de datos Date y BinData
- BSON soporta los siguientes tipos de datos numéricos:
 - int32 - 4 bytes (enteros con signo de 32-bit)
 - int64 - 8 bytes (enteros con signo de 64-bit)
 - double - 8 bytes (64-bit IEEE 754 de coma flotante)

Claves de los documentos

- Además de los datos de un documento, MongoDB siempre introduce un campo adicional `_id`
 - Todo documento tiene que tener un campo `_id` único
 - Este campo `_id` puede contener un valor de cualquier tipo BSON, excepto un array
- Podemos generar el identificador nosotros:
 - `x = "55674321R"`
 - `y = ObjectId("507f191e810c19729de860ea")`
- O dejarle a MongoDB que lo haga
 - Entonces el tipo de ese campo es `ObjectId`
 - Es un tipo de datos de 12 bytes, donde 4 bytes representan un timestamp, 3 un identificador de máquina, 2 el identificador del proceso y 3 restantes un contador
 - Tiene el atributo `str` y los métodos `getTimeStamp()` y `toString()`
- Es preferible que MongoDB lo genere por nosotros
- El campo `_id` es indexado lo que explica que se guardan sus detalles en la colección del sistema `system.indexes`

Cambio respecto al concepto relacional



Cambio respecto al concepto relacional

- Al ser orientado a documentos, el esquema de las entidades no es fijo como sucede en el modelo relacional.
 - Existe lo denominado esquema orientado a documentos o orientado a aplicación.
- Consideraciones:
 - ¿Tiene sentido introducir toda la información en una única colección?
 - ¿Qué información debe pertenecer a cada documento?
 - ¿Se pueden cambiar los datos en un sitio central (usando referencias) en vez de cambiarse en el conjunto de documentos que contienen esos datos?

Modelado

- La fase de diseño es una de las más importantes y costosas del modelo relacional, dónde se definen todas las entidades y las relaciones entre ellas.
- En MongoDB se utiliza un diseño de datos orientado a aplicación: el contenido y atributos de cada entidad será propio de cada documento.
- Es habitual dar al documento un sentido de entidad desnormalizada: MongoDB no soporta joins de forma nativa, por lo que la información acaba repetida por todos los documentos en los que debe estar.
 - Esto no evita que podamos realizar “joins” de forma programática desde otras aplicaciones.
 - Existe una limitación importante a la hora de “empotrar” información: el tamaño máximo de cada documento es de 16 MB.
 - Tampoco existen restricciones por lo que no es posible validar coherencia ni consistencia entre relaciones.

Modelado

- Relaciones uno a uno (1-1)
 - Los siguientes aspectos deberían tenerse en cuenta cuando se modelan relaciones uno a uno:
 - La frecuencia de acceso a los documentos
 - Tamaño de los elementos, teniendo en cuenta la limitación de 16 MB
 - La atomicidad de los datos y su consistencia
 - Considera el siguiente código:

```
db.captains.insert({_id: 'JamesT.Kirk' , name: 'James T. Kirk', age: 38, ship:'ussenterprise'});
db.starships.insert({_id : 'ussenterprise' , name: 'USS Enterprise', class: 'Galaxy', captain: 'JamesT.Kirk'} );
```

Un modo más eficiente, en este caso, es colocar el documento del capitán dentro del documento nave:

```
db.starships.insert( { _id: 'ussenterprise1', name: 'USS Enterprise', class: 'Galaxy', captain: {name: 'James T. Kirk', age: 38}} );
```

Modelado

- Relaciones uno a muchos (1-N)
 - Si una nave tiene mucho miembros de tripulación o una nave tiene muchos instructores •
 - ¿Tiene sentido empotrar la lista de toda la tripulación o de instructores dentro del documento nave que tiene un límite de tamaño de 16 MB?
 - Por tanto, en caso de relaciones 1-N suele ser a menudo conveniente enlazar documentos entre colecciones y además hacerlo desde la colección que guarda muchos valores a la colección que guarda sólo uno.
 - ¿Estamos hablando de una relación 1-N o de 1-pocos? – En el segundo caso un array dentro del documento podría ser una mejor opción

- Ejemplo:

```
db.starships.insert( { _id: 'ussenterprise1', name: 'USS Enterprise',  
class: 'Galaxy', captain: { name : 'James T. Kirk', age : 38}, instructors:  
[1, 2, 3]}
```

```
db.instructors.insert( { _id: 1, name : 'Tuvok', candidates : [99, 100],  
starship: 'ussenterprise1' } );
```


Modelado

- Relaciones muchos a muchos (N-M)
 - Considera la relación en la que un candidato puede tener varios instructores y viceversa •
 - Varios candidatos serán asignados a un instructor y un instructor será asignado a varios candidatos para que los instruya
 - En esta relación tendremos dos colecciones (candidates e instructors) y un enlace bidireccional, dado que cada candidato tiene una lista de instructores y por cada instructor hay una lista de candidatos
 - Si quisiéramos empotrar los candidatos dentro del documento instructor, sería necesario tener un instructor antes de un candidato, no podrían existir uno sin el otro
 - Ejemplo:

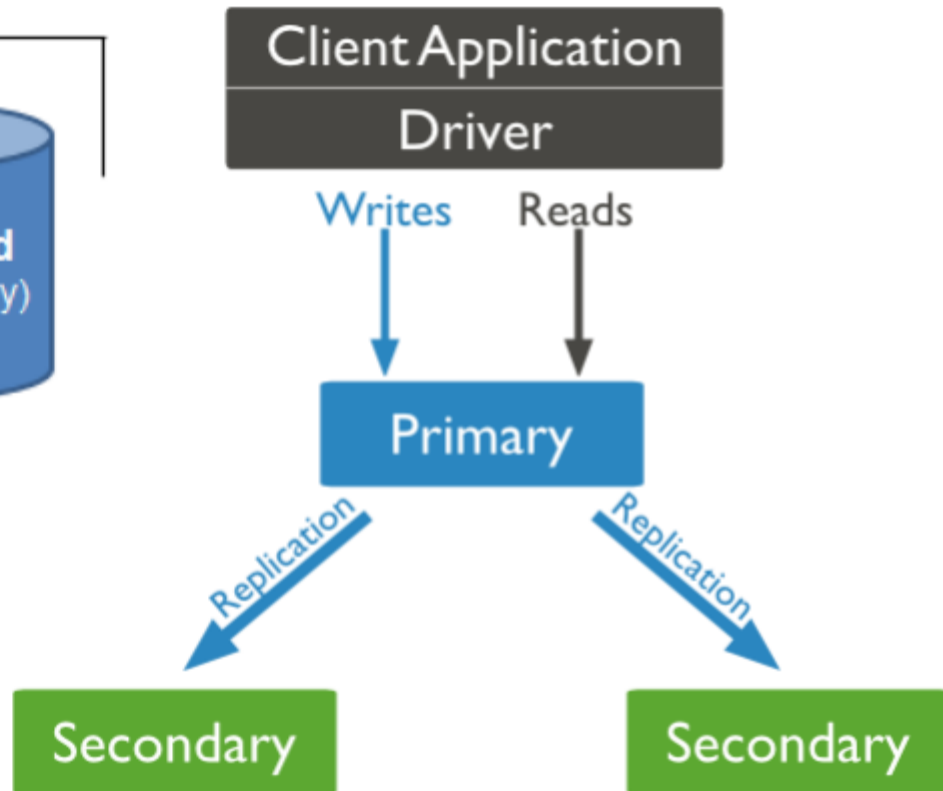
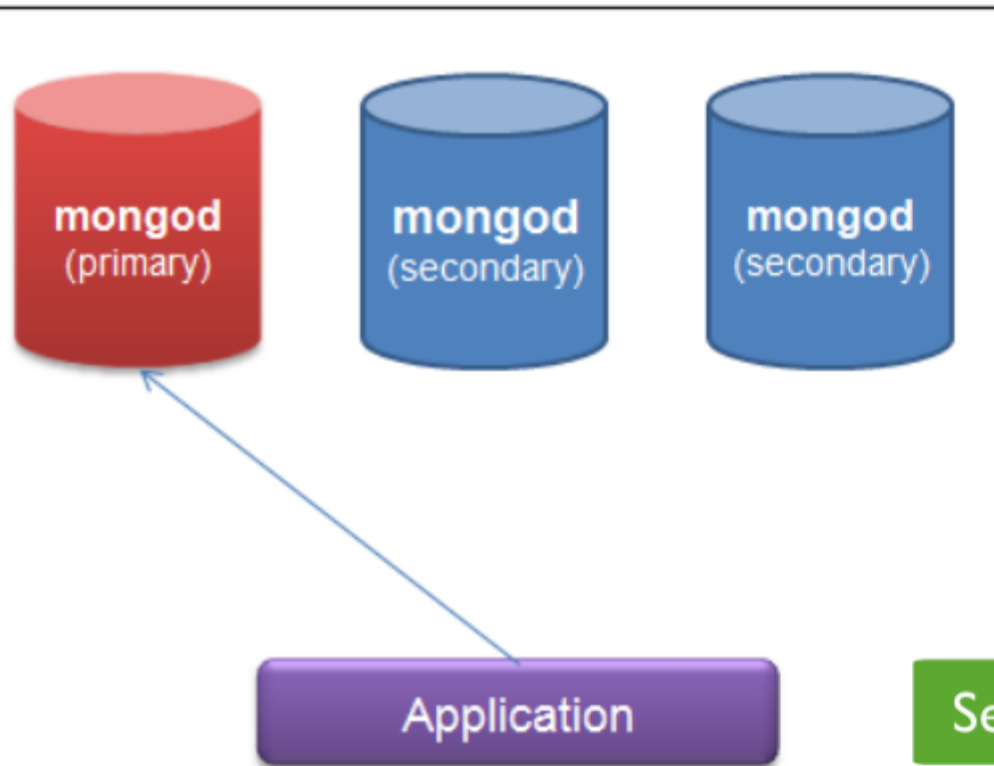
```
db.candidates.insert( { _id : 99, name : 'Harry Kim', instructors : [1, 2] } );
db.instructors.insert( { _id : 1, name : 'Tuvok', candidates : [99, 100],
starship: 'ussenterprise1' } );
```
 - Recuerda que es tu programa el que tendrá que garantizar la consistencia. Por ejemplo, asegurando que existe un candidato cuyo `_id` es 100

Replicación

- La replicación funciona similar a cómo funciona la replicación en una base de datos relacional
 - Replicación Master-Slave. Las escrituras son enviadas a un único servidor, el maestro, que sincroniza su estado a uno o varios esclavos
 - Todos juntos configuran un ReplicaSet
- MongoDB puede ser configurado para soportar lecturas en esclavos o no, lo cual puede redundar en la distribución de la carga a cambio de sufrir en consistencia
- Si el maestro se cae, un esclavo es promocionado como máster automáticamente
- Aunque la replicación puede mejorar el rendimiento (distribución de lecturas) su propósito principal es mejorar en robustez
 - La combinación de replicación y sharding son un enfoque común
 - Cada shard puede consistir de un maestro y un esclavo

Replicación

MongoDB Replica Set

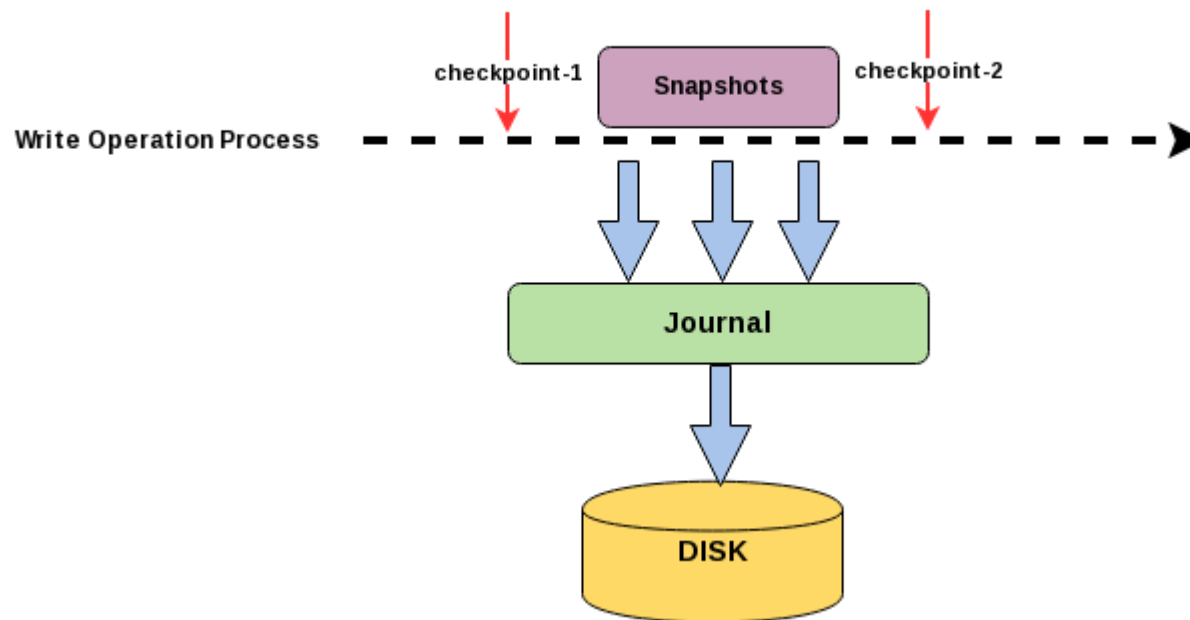


Replica Set

- Todos los writes van a un nodo primario (máster) pero es posible hacer lecturas de los nodos secundarios (slave)
- Cuando se lee de un nodo esclavo no hay garantía que el dato leído esté actualizado
- Sin embargo si sólo se lee y escribe del nodo primario sí hay garantía de consistencia
- En el periodo entre la caída de un nodo primario y un nuevo nodo primario es elegido, la consistencia está comprometida.
 - Todo nodo tiene un nivel de prioridad usado en el proceso de elección
 - Si el nivel de prioridad es puesto a 0 , ese nodo no puede ser nodo primario
- El modo de escritura en un ReplicaSet está configurado por:
 - El número de nodos que van a soportar el write
 - Los nodos sobre cuyos journals se va a escribir

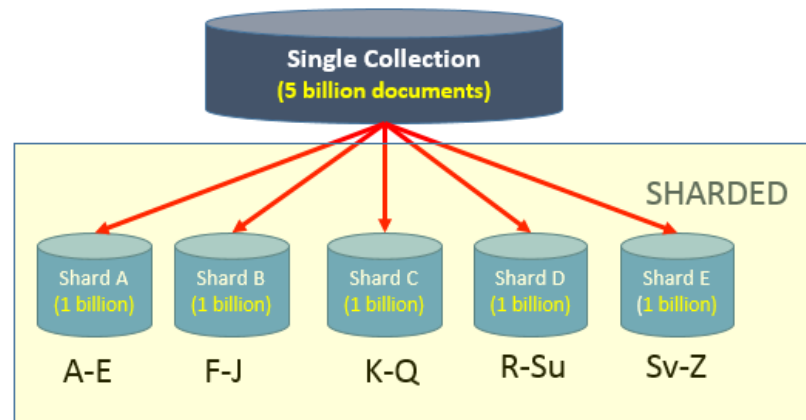
Journals

- El journal es una estructura intermedia que almacena las modificaciones sobre documentos antes de persistirlos al disco.



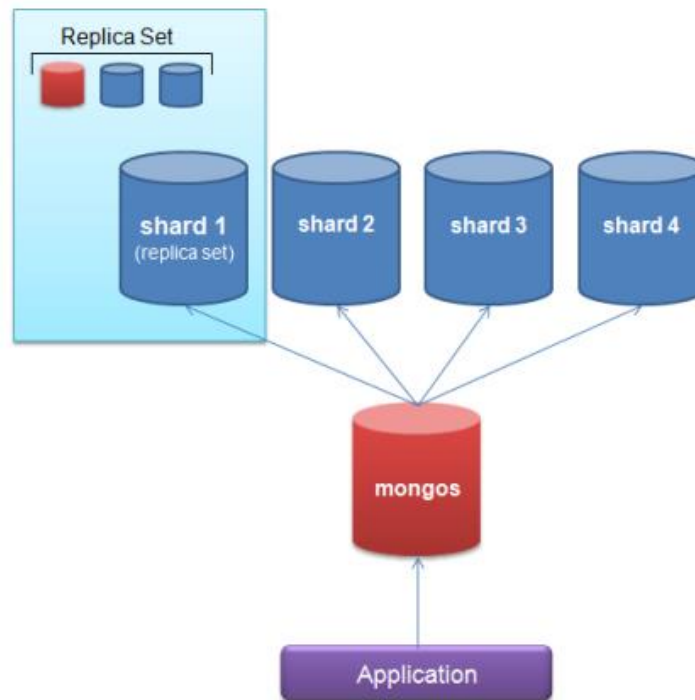
Sharding

- Un sharded cluster es un conjunto de Replica Sets (shards) cuya función es la de repartirse uniformemente la carga de trabajo,
- Esto nos permite escalar horizontalmente nuestras aplicaciones para así poder trabajar con grandes cantidades de datos.
- *MongoDB uses sharding to support deployments with very large data sets*
- Un *shard* es un maestro con uno o varios esclavos y los datos son distribuidos por rangos entre todas las instancias de la base de datos. Los esclavos pueden ser añadidos dinámicamente.



Sharding

- MongoDB soporta auto-sharding, una técnica de escalabilidad horizontal que separa datos a través de varios servidores.
- Una implementación podría poner los datos con nombre que empieza entre A-M en el servidor 1 y el resto en el servidor 2



Sharding

- El modus operandi de sharding consiste en dividir una colección en varios nodos y luego acceder a sus contenidos a través de un nodo especial que actúa como router (mongos)
- Es una decisión que se toma cuando se despliega la base de datos, bien siendo normal o usando sharding
- Para usar sharding es necesario que cada colección declare un shard-key
 - Clave definida para uno o varios campos de un documento
 - Debe ayudar a dividir la colección en fragmentos (chunks)
 - Los chunks son luego distribuidos entre nodos, lo más equitativamente posible
- La instancia mongos usa la clave shard-key para determinar el chunk y así el nodo utilizado

Sharding

- Las condiciones que un shard-key ha de cumplir son:
 - Cada documento tiene un shard-key
 - El valor del shard-key no puede modificarse
 - Debe ser parte de un índice y debe ser el primer campo de un índice compuesto
 - No puede haber un índice único a no ser que esté conformado por el shard-key
 - Si no se usa el shard-key en una operación de lectura esta petición llegará a todos los shards
 - La clave de shard debe ofrecer suficiente cardinalidad para poder utilizar todos los shards

Operaciones

- CRUD vs IFUR

CRUD	IFUR
Create	Insert
Read	Find
Update	Update
Delete	Remove

Insert

- Trabajamos en un documento implícito llamado db que representa la base de datos.
- Si la base de datos no se modifica explícitamente nos conectamos por defecto a la base de datos test.
- Una colección (ships) se crea automáticamente cuando insertamos un documento en ella
`db.ships.insert({'name':'USS Prometheus','operator':'Starfleet','class':'Prometheus'})`

Find

- Los métodos `findOne()` y `find()` usan un documento como primer parámetro y un segundo para indicar los campos sobre los que realizar la selección
- El campo `_id` se muestra por defecto en los resultados:
`db.ships.findOne({'name':'USS Defiant'}, {'class':true,'_id':false})`
- El método `find()` devuelve todos los documentos que cumplen el lo especificado en el documento de selección.
- Necesita los operadores de consulta para realizar consultas más precisas:

Operador	Descripcion	Ejemplo
\$gt	Mayor	<code>db.ships.find({class:{\$gt:'P'}})</code>
\$gte	Mayor o igual	<code>db.ships.find({class:{\$gte:'P'}})</code>
\$lt	Menor	<code>db.ships.find({class:{\$lt:'P'}})</code>
\$lte	Menor o igual	<code>db.ships.find({class:{\$lte:'P'}})</code>

Find

- Los selectores de consulta en MongoDB son como la cláusula where de una sentencia SQL
 - Se usan para encontrar, contar, actualizar y borrar documentos de una colección
 - Un selector es un objeto JSON, el más sencillo es {} que sirve para seleccionar todos los documentos (null también funciona).
- El operador \$exists se utiliza para comprobar la presencia o ausencia de un campo:
 - `db.unicorns.find({vampires: {$exists: false}})`
- Si queremos utilizar el operador booleano OR tenemos que hacer uso del operador \$or y asociarle un array de tuplas clave/valor sobre los que realizar el OR:
 - `db.unicorns.find({gender: 'f', $or: [{loves: 'apple'}, {loves: 'orange'}, {weight: {$lt: 500}}]})`
- Un valor de tipo ObjectId asociado al campo _id puede seleccionarse como:
 - `db.unicorns.find({_id: ObjectId("TheObjectId")})`

Update

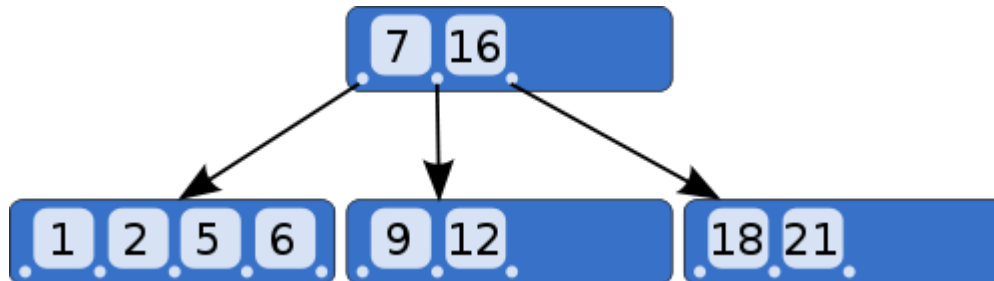
- Cuidado con el uso por defecto de update que por defecto reemplaza el documento seleccionado con los campos del documento pasado como segundo argumento, manteniendo sólo el campo `_id`:
 - `db.ships.update({name : 'USS Prometheus'}, {name : 'USS Something'})`
 - `db.ships.find({name : 'USS Something'}).pretty()`
- Generalmente lo que queremos es reemplazar algunos elementos y mantener el resto intacto con el operador `$set`:
 - `db.ships.update({name : 'USS Something'}, {$set : {operator : 'Starfleet', class : 'Prometheus'}})`
 - `db.ships.find({name : 'USS Something'}).pretty()`
- Si queremos quitar un conjunto de valores de un documento se puede hacer uso del operador `$unset`:
 - `db.ships.update({name : 'USS Prometheus'}, {$unset : {operator : 1}})`

Remove

- Como en todos los comandos de MongoDB se reutiliza la sintaxis de la operación find() para indicar qué documentos se quieren eliminar.
- Si emitimos un remove() sin parámetros sobre una colección eliminará uno a uno todos los elementos de la colección.
- La operación drop() eliminará todos los documentos de la colección más rápido ya que no trabaja a nivel de documento y además elimina los índices
- Para colecciones grandes es mejor usar drop() y luego recrear los índices sobre una nueva colección vacía.
- Ejemplos:
 - `db.ships.drop();`
 - `db.ships.remove({name : 'USS Prometheus'})`

Indexación

- Los índices mejoran el rendimiento de las consultas y operaciones de ordenación en MongoDB, de una forma similar a las RDBMS.
- MongoDB los guarda como un B-Tree
- Los datos se guardan en una estructura de árbol pero de forma balanceada entre nodos
- Permite la recuperación de listas de claves ordenadas en ambas direcciones del árbol, por lo que la misma indexación permite una recuperación tanto ascendente como descendente



Indexación

- Se crean con la sentencia `ensureIndex()`, identificando el sentido de ordenación por campo ascendente (1) o descendente (-1):
 - `db.unicorns.ensureIndex({name: 1});`
- Para conocer los índices de la colección, la sentencia es:
 - `db.unicorns.getIndexes()`
- Se eliminan con `dropIndex()`:
 - `db.unicorns.dropIndex({name: 1});`
- Para asegurarnos que el índice es único, usamos el atributo `unique`:
 - `db.unicorns.ensureIndex({name: 1}, {unique: true});`
- Los índices también pueden ser compuestos:
 - `db.unicorns.ensureIndex({name: 1, vampires: -1});`

Explain

- Para saber si se está usando un índice o no, usamos explain():
 - db.unicorns.find().explain()
- Si la salida indica que se usa un cursor de tipo BasicCursor indica que el campo no es indexado...se escanearán más documentos y tardarán más las búsquedas u ordenaciones:

```
{
  "cursor" : "BasicCursor", // el campo no es indexado
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 12,
  "nscanned" : 12,
  "nscannedObjectsAllPlans" : 12,
  "nscannedAllPlans" : 12,
  "scanAndOrder" : false,
  "indexOnly" : false, // si la consulta se puede resolver mirando sólo el índice
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0, // indica cuánto tardó la consulta
  "indexBounds" : {
  },
  "server" : "dipinaXPS14z:27017"
}
```

Explain

- Si creamos un índice ese campo tendrá asociado un índice de tipo BtreeCursor, optimizándose las operaciones sobre esos documentos

```
{
  "cursor" : "BtreeCursor name_1", // los campos de consulta son indexados
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "name" : [
      [
        "Pilot",
        "Pilot"
      ]
    ]
  },
  "server" : "dipinaXPS14z:27017"
}
```

Comandos

Command Line

mongo | Start the shell from the command line

Command line options:

--hostname localhost | hostname to connect

--port 27017 | connect to port 27017 (default)

-u foo | username foo

-p bar | password bar

--authenticationDatabase arg | database to authenticate

mongoimport | import a data from a file into MongoDB

> mongoimport --db <db> -c <coll> --file <filename> --type <type>

mongodump | Dumps contents of a db to a file

> mongodump --db <db> -c <coll>

mongorestore | restore from a dump to MongoDB

> mongorestore --db <db> -c <coll> <bson file>

Operaciones básicas

Basic Shell commands

help | get help for the context you are in
exit | exit the shell

use <database> | select and use database
db | show selected database
show dbs | show databases on server
show collections (show tables) | show collections in current db
show users | show users in current database

Querys

Query commands (db.collection.)

Finding documents:

`find().pretty()` | Finds all documents using nice formatting
`find({}, {name:true, _id:false})` | retrieve only *name* field
`findOne()` | Finds one arbitrary document
`findOne({name:'abc'})` | Finds one document by attribute

Inserting document:

`insert({name:'a'})` | Insert new document in the collection

Removing document:

`remove()` | Remove all collection documents
`remove({name:'a'})` | Remove by criteria

Updating documents:

`update({name:'a'}, {age:25})` | replaces the whole document
`update({name:'a'}, {$set:{age:25}})` | change certain attribute
`update({name:'b'}, {$unset:{age:1}})` | unset attribute
`findAndModify({query:{...}, sort:{...},update:{...}})` |
Automatically find and update

Queries

Query operators (\$)

Comparison:

Ex: `db.<collection>.find({ qty: { $gt: 20 } })`

\$gt | matches values greater than the value

\$in | matches values supplied in an array

\$gte | matches values greater than or equal the value

\$lte | matches values less than or equal the value

\$ne | matches all values that are not equal to given

\$lt | matches values less than the value

\$nin | matches values that do not exist in an array

Logical:

Ex: `db.<coll>.find({ price: 9, $or: [{ qty: { $lt: 20 } }, { sale: true }] })`

\$or | joins query clauses with a logical OR

\$and | joins query clauses with a logical AND

\$not | returns documents that do not match

\$nor | joins query clauses with a logical NOR

Evaluation:

\$exists | matches documents that have a field

> `db.inventory.find({ qty: { $exists: true, $nin: [5, 15] } })`

\$type | matches a field if it is of a given BSON type

> `db.inventory.find({ price: { $type: 1 } })`

Evaluation:

\$mod | perform a modulo on a field and select if 0

> `db.inventory.find({ qty: { $mod: [4, 0] } })`

\$regex | matches a regex expression on a field

> `db.collection.find({ field: /acme.*corp/i })`

\$where | matches against a JavaScript expression

> `db.myCollection.find({ $where: "this.credits - this.debits < 0" })`

Array:

\$all | matches arrays that contain all elements given

> `db.inventory.find({ tags: { $all: ["appliance", "school", "book"] } })`

\$elemMatch | matches multiple conditions in array

> `db.collection.find({ array: { $elemMatch: { value1: 1, value2: { $gt: 1 } } } })`

\$size | matches if the array is of specified size

> `db.collection.find({ field: { $size: 1 } })`

Update

Update operators (\$)

Field operators:

- \$inc | Increment a value by a specified amount
 - \$rename | rename a field
 - \$setOnInsert | set a value only if inserting
 - \$set | set the value of a field on an existing document
 - \$unset | remove the field from an existing document
-

Array operators:

- \$ | update the first element in an array that matches
- > db.coll.update({ _id: 1, grades: 80 }, { \$set: { "grades.\$" : 82 } })
- > db.coll.update({ _id: 4, "grades.stack": 85 }, { \$set: { "grades.\$.std": 6 } })

- Ex:** db.coll.update({ name: "joe" }, { \$push: { scores: 89 } })
- \$addToSet | add element to array if it doesn't exist
 - \$push | adds an item to an array
 - \$pop | update the first element in an array that matches
 - \$pull | remove items which match a query statement
 - \$pullAll | remove multiple values from an array

Array modifiers:

- \$each | modify \$push and \$addToSet to add many
 - > db.coll.update({ name: "joe" }, { \$push: { scores: { \$each: [90, 85] } } })
 - \$slice | modify \$push to limit size of updated array
 - > db.coll.update({ _id: 2 }, { \$push: { grades: { \$each: [80, 78], \$slice: -5 } } })
 - \$sort | modify \$push to reorder documents in array
 - > db.coll.update({ _id: 2 }, { \$push: { grades: 81, \$sort: { grades: 1 } } })
-

Bitwise:

- \$bit | performs a bitwise update of a field
 - > db.coll.update({ field: NumberInt(1) }, { \$bit: { field: { and: NumberInt(5) } } })
-

Isolation:

- \$isolated | isolates a write operation for multiple documents
- > db.coll.update({field1:1,\$isolated:1}, {\$inc:{field2 : 1}}, {multi: true})

Indexes

Indexes

Indexing - db.collection.

ensureIndex({a:1}) | Creates an ascending index
dropIndex({'a':1}) | Removes an index from a collection
getIndexes() | Get indexes details of a collection
reIndex() | Rebuild all indexes on a collection
compact() | Defragment a collection and rebuild indexes

Properties- db.collection.<IndexOp>({...},{<option>})

expireAfterSeconds:300 | delete docs after set time
unique:true | only unique data
sparse:true | only documents with the index field

Options - db.collection.<IndexOp>({...},{<option>})

background:true | create index in the background
dropDups:true | Drop duplicates on unique index creation

Info - db.collection.

totalIndexSize() | get index size

Puedes ver el cheatsheet completo [aquí](#)

Ejemplos

Arrancar consola

```
user@ubuntu:~$ mongo
MongoDB shell version: 2.4.9
connecting to: test
Server has startup warnings:
Sat Feb  4 12:49:13.472 [initandlisten]
Sat Feb  4 12:49:13.472 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary
.
Sat Feb  4 12:49:13.472 [initandlisten] **          32 bit builds are limited to le
ss than 2GB of data (or less with --journal).
Sat Feb  4 12:49:13.472 [initandlisten] **          See http://dochub.mongodb.org/c
ore/32bit
Sat Feb  4 12:49:13.483 [initandlisten]
> █
```

Crear DB got

```
> use got
switched to db got
>
```

Ejemplos

Insertar registro:

```
> db.characters.insert({'name':'Robb','family':'Stark','alias':'King in the North'});
```

Ver colecciones:

```
> show collections;
characters
system.indexes
> █
```

Buscar registro:

```
> db.characters.find();
{ "_id" : ObjectId("5895ed5e0f2434a13e8f761d"), "name" : "Robb", "family" : "Stark", "alias" : "King in the North" }
```

Insertar y volver a buscar:

```
> db.characters.insert({'name':'Jaime','family':'Lannister','alias':'Kingslayer'});
> db.characters.find();
{ "_id" : ObjectId("5895ed5e0f2434a13e8f761d"), "name" : "Robb", "family" : "Stark", "alias" : "King in the North" }
{ "_id" : ObjectId("5895edad0f2434a13e8f761e"), "name" : "Jaime", "family" : "Lannister", "alias" : "Kingslayer" }
```

Ejemplos

Analizar indexación:

```
> db.characters.find().explain();
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 2,
  "nscanned" : 2,
  "nscannedObjectsAllPlans" : 2,
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {

  },
  "server" : "ubuntu:27017"
}
```

Ejemplos

Analizar indexación:

```
> db.characters.ensureIndex({'name':1});
> db.characters.find().explain();
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 2,
  "nscannedObjects" : 2,
  "nscanned" : 2,
  "nscannedObjectsAllPlans" : 2,
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "name" : [
      [
        "Robb",
        "Robb"
      ]
    ]
  },
  "server" : "ubuntu:27017"
}
```

```
> db.characters.find({'name':'Robb'}).explain();
{
  "cursor" : "BtreeCursor name_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "name" : [
      [
        "Robb",
        "Robb"
      ]
    ]
  },
  "server" : "ubuntu:27017"
}
```

Estructuras

Comportamiento de los documentos en una colección:

```
> db.characters.insert({'name': 'Arya', 'family': 'Stark', 'alias': 'Horseface', 'weapon': 'Needle'});
> db.characters.find();
{ "_id" : ObjectId("5895ed5e0f2434a13e8f761d"), "name" : "Robb", "family" : "Stark", "alias" : "King in the North" }
{ "_id" : ObjectId("5895edad0f2434a13e8f761e"), "name" : "Jaime", "family" : "Lannister", "alias" : "Kingslayer" }
{ "_id" : ObjectId("5895ee650f2434a13e8f7620"), "name" : "Arya", "family" : "Stark", "alias" : "Horseface", "weapon" : "Needle" }
```

Find con exists:

```
> db.characters.find({weapon: {$exists: true}});
{ "_id" : ObjectId("5895ee650f2434a13e8f7620"), "name" : "Arya", "family" : "Stark", "alias" : "Horseface", "weapon" : "Needle" }
```

Find con or:

```
> db.characters.find({$or: [{name: 'Robb'}, {name: 'Jaime'}]});
{ "_id" : ObjectId("5895ed5e0f2434a13e8f761d"), "name" : "Robb", "family" : "Stark", "alias" : "King in the North" }
{ "_id" : ObjectId("5895edad0f2434a13e8f761e"), "name" : "Jaime", "family" : "Lannister", "alias" : "Kingslayer" }
```

Find sobre ObjectId:

```
> db.characters.find({_id: ObjectId("5895edad0f2434a13e8f761e")});
{ "_id" : ObjectId("5895edad0f2434a13e8f761e"), "name" : "Jaime", "family" : "Lannister", "alias" : "Kingslayer" }
```

Update y remove

Update

```
> db.characters.update({name : 'Arya'}, {$set : {alias : 'Arry'}});
> db.characters.find({alias : 'Arry'}).pretty();
{
  "_id" : ObjectId("5895ee650f2434a13e8f7620"),
  "alias" : "Arry",
  "family" : "Stark",
  "name" : "Arya",
  "weapon" : "Needle"
}
```

Remove

```
> db.characters.remove({alias : 'Arry'});
> db.characters.find().pretty();
{
  "_id" : ObjectId("5895ed5e0f2434a13e8f761d"),
  "name" : "Robb",
  "family" : "Stark",
  "alias" : "King in the North"
}
{
  "_id" : ObjectId("5895edad0f2434a13e8f761e"),
  "name" : "Jaime",
  "family" : "Lannister",
  "alias" : "Kingslayer"
}
```

Estado de la base de datos

Stats

```
> db.stats();
{
  "db" : "got",
  "collections" : 3,
  "objects" : 8,
  "avgObjSize" : 59,
  "dataSize" : 472,
  "storageSize" : 16384,
  "numExtents" : 3,
  "indexes" : 2,
  "indexSize" : 16352,
  "fileSize" : 50331648,
  "nsSizeMB" : 16,
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 5
  },
  "ok" : 1
}
```


Estructuras complejas

Podemos crear estructuras dentro de las estructuras de un documento, de forma que se consiga representar una relación dentro de una relación

```
> db.families.insert({'name':'Stark', 'procedence':'Winterfell', 'members':[{'name':'Robb','alias':'King in the North'}]});
> db.families.find().pretty();
{
  "_id" : ObjectId("5895f8d70f2434a13e8f7623"),
  "name" : "Stark",
  "procedence" : "Winterfell",
  "members" : [
    {
      "name" : "Robb",
      "alias" : "King in the North"
    }
  ]
}
```

Estructuras complejas

Dentro de esa estructura, podremos añadir nuevos registros con el operador \$push:

```
> db.families.update({'name': 'Stark'},{$push: {'members': {'name': 'Jon Snow', 'alias': 'White wolf'}}})
> db.families.find().pretty();
{
  "_id" : ObjectId("5895f91a0f2434a13e8f7624"),
  "members" : [
    {
      "name" : "Robb",
      "alias" : "King in the North"
    },
    {
      "name" : "Jon Snow",
      "alias" : "White wolf"
    }
  ],
  "name" : "Stark",
  "procedence" : "Winterfell"
}
```

Estructuras complejas

Dentro de esa estructura, podremos eliminar registros con el operador \$pull:

```
> db.families.update({'name': 'Stark'},{$pull: {'members': {'name': 'Robb'}}});  
> db.families.find().pretty();  
{  
  "_id" : ObjectId("5895fbba0f2434a13e8f7626"),  
  "members" : [  
    {  
      "name" : "Jon Snow",  
      "alias" : "White wolf"  
    }  
  ],  
  "name" : "Stark",  
  "procedence" : "Winterfell"  
}
```

Relaciones

Podemos definir distintas colecciones con datos cruzados entre ellas, referenciándolos mediante identificadores.

```
> db.characters.insert({'id':1,'name':'Robb','family':'Stark','alias':'King in the North'});
> db.characters.insert({'id':2,'name':'Jaime','family':'Lannister','alias':'Kingslayer'});
>
> db.families.insert({'name':'Stark', 'procedence':'Winterfell', 'members':[1,2]});
```

```
> db.characters.find().pretty();
```

```
{
  "_id" : ObjectId("5895fd6c0f2434a13e8f7627"),
  "id" : 1,
  "name" : "Robb",
  "family" : "Stark",
  "alias" : "King in the North"
}
{
  "_id" : ObjectId("5895fd6c0f2434a13e8f7628"),
  "id" : 2,
  "name" : "Jaime",
  "family" : "Lannister",
  "alias" : "Kingslayer"
}
```

```
> db.families.find().pretty();
```

```
{
  "_id" : ObjectId("5895fd800f2434a13e8f7629"),
  "name" : "Stark",
  "procedence" : "Winterfell",
  "members" : [
    1,
    2
  ]
}
```

Funciones de agregación

- Las operaciones de agregación procesan registros de datos y devuelven resultados.
- MongoDB 2.2 introdujo una framework de agregación modelado en torno al concepto de pipelines de procesamiento de datos.
- Ofrece un conjunto de operadores sencillos de agregación como `count()`, `distinct()`, `group()`, `mapReduce()`.
- El operador `aggregate()` añade la posibilidad de pipelines complejos, incluyendo en las últimas versiones la funcionalidad de joins a través de `$lookup`.

Función	Descripción
<code>db.collection.aggregate()</code>	Ejecuta un pipeline de agregación
<code>db.collection.group()</code>	Agrupar documentos de una colección
<code>db.collection.count()</code>	Cuenta documentos de una colección
<code>db.collection.mapReduce()</code>	Ejecuta un pipeline de MapReduce para datasets largos.

Pipeline de agregación

- MongoDB incorpora un mecanismo para poder agregar datos de documentos en diferentes pasos:
 - Cada paso toma como entrada un conjunto de documentos y produce un conjunto de documentos como resultado
 - Hay operaciones que en un paso dado mantendrán el mismo número de documentos de entrada pero hay otras que los puedes reducir (filtrar)

db.usgs.aggregate([

\$project



\$match



\$group



\$sort

]);

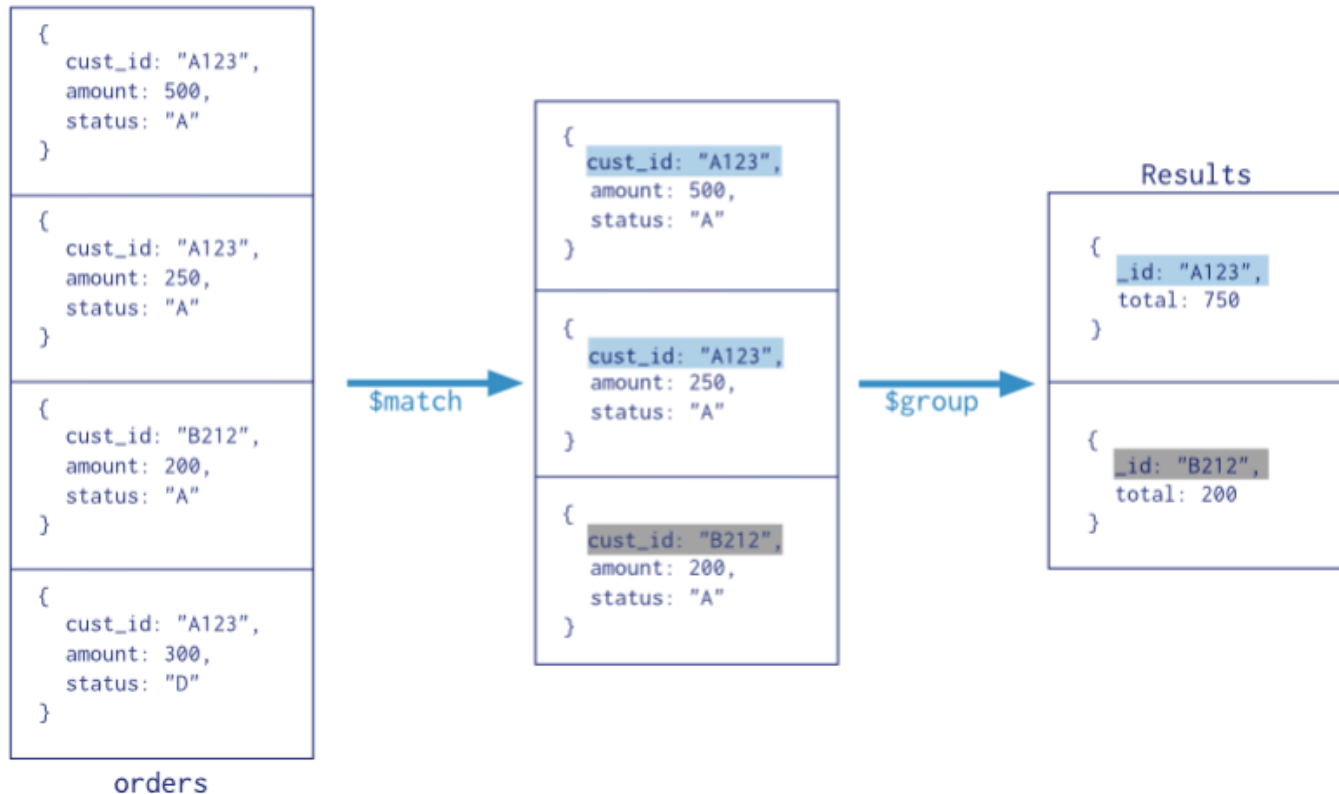
Pipeline de agregación

Operaciones:

- `$project` – cambia el conjunto de documentos modificando claves y valores. Es de tipo 1 a 1
- `$match` – es una operación de filtrado que reduce el conjunto de documentos generando un nuevo conjunto de los mismos que cumple alguna condición, ej. `operator="starfleet"`
- `$group` – hace el agrupamiento en base a claves o campos indexados, reduciendo el número de documentos
- `$sort` – ordenar en ascendente o descendente, dado que es computacionalmente costoso debería ser uno de los últimos pasos de la agregación
- `$skip` – permite saltar entre el conjunto de documentos de entrada, por ejemplo avanzar hasta el décimo documento de entrada. Se suele usar junto con `$limit`
- `$limit` – limita el número de documentos a revisar
- `$unwind` – desagrega los elementos de un array en un conjunto de documentos.

Pipeline de agregación

Collection
↓
`db.orders.aggregate(`
 `$match phase` → `{ $match: { status: "A" } },`
 `$group phase` → `{ $group: { _id: "$cust_id", total: { $sum: "$amount" } } }`
)

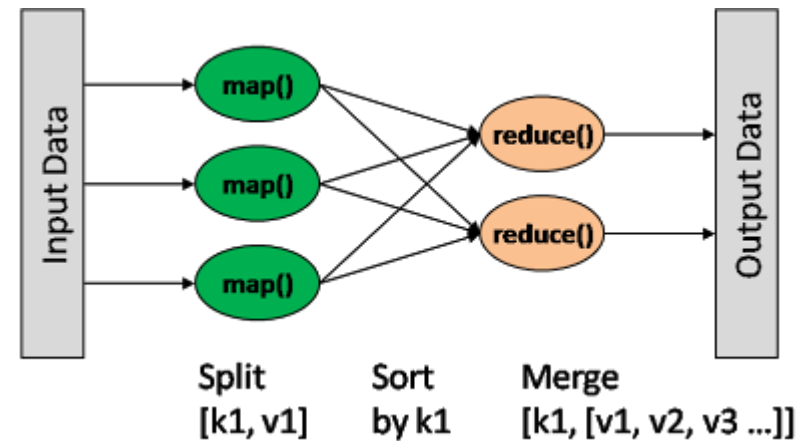


Map Reduce

- MapReduce es un enfoque para procesar grandes volúmenes de datos que tiene dos beneficios:
 - Puede ser paralelizado permitiendo que largos volúmenes de datos sean procesados a través de varios cores/CPU's y máquinas
 - Puedes escribir código real en JavaScript para hacer el procesamiento

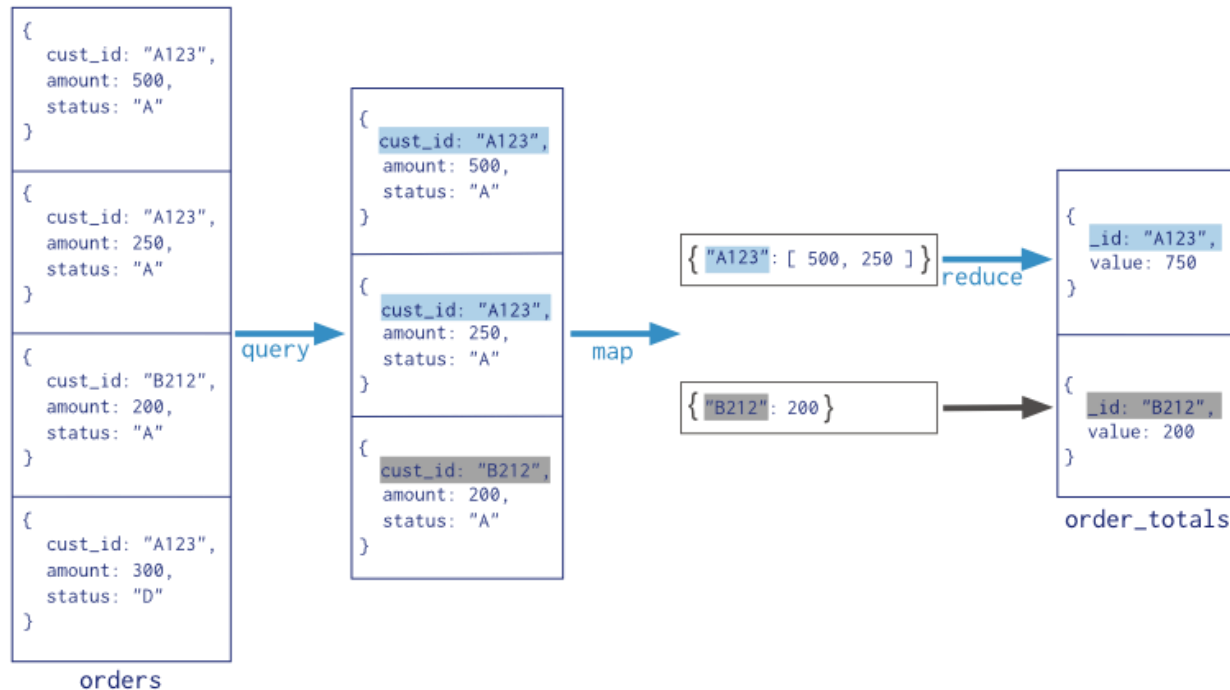
Se realiza a través de dos pasos:

1. Mapear los datos transformando los documentos de entrada en pares clave/valor
2. Reducir las entradas conformadas por pares clave y array de valores asociados a esa clave para producir el resultado final



Map Reduce

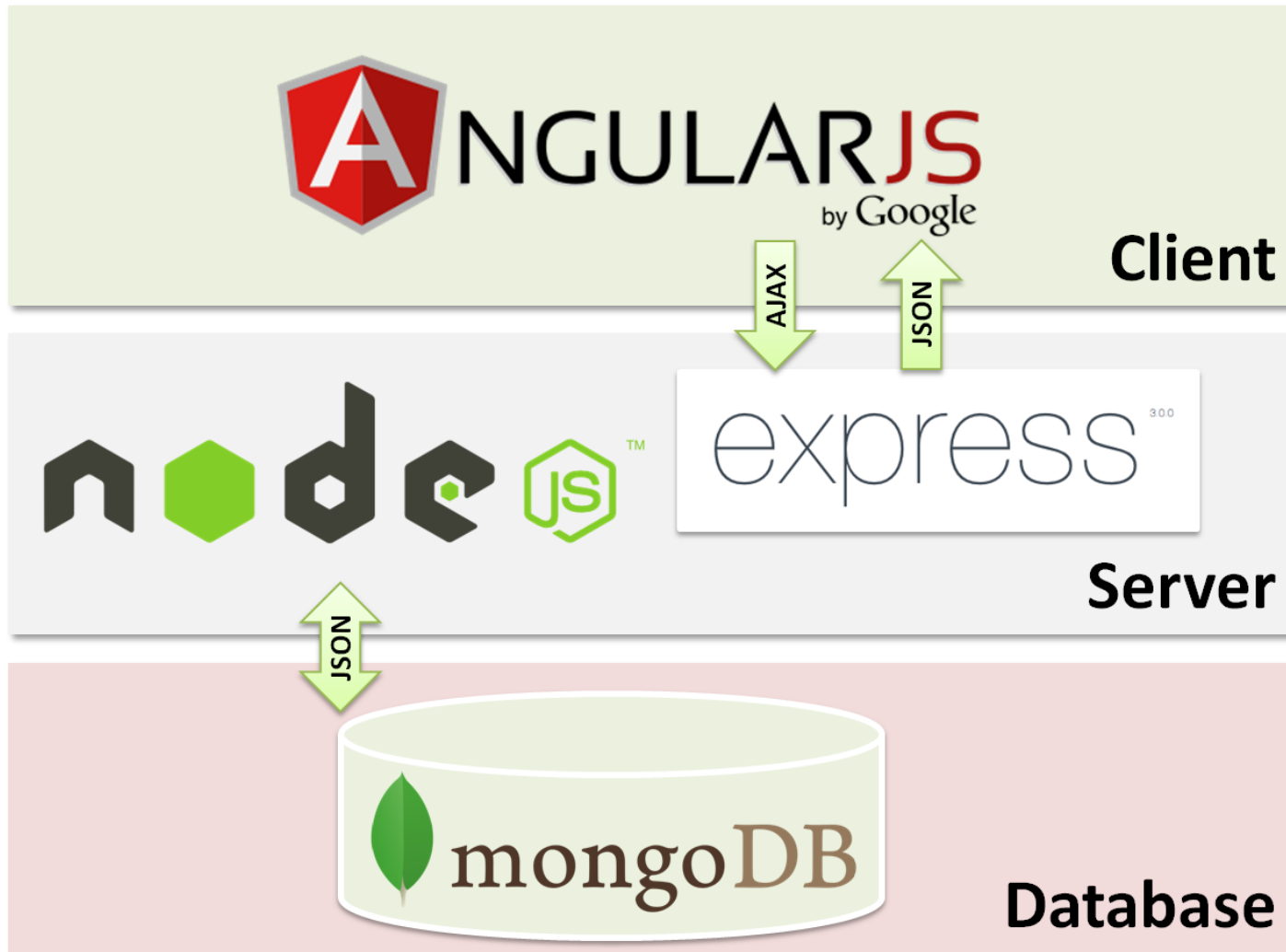
Collection
↓
db.orders.mapReduce(
 map → function() { emit(this.cust_id, this.amount); },
 reduce → function(key, values) { return Array.sum(values) },
 query → {
 query: { status: "A" },
 out: "order_totals"
 }
)



MEAN

- MEAN es un stack de desarrollo web muy novedoso, que ha ayudado a popularizar el uso de MongoDB
- Se caracteriza porque cada una de las piezas del stack tienen a Javascript como elemento central
- Esto hace que sea prácticamente el único stack en el que únicamente es necesario conocer un lenguaje, lo que facilita el aprendizaje y uso (javascript full stack)
- MEAN se compone de:
 - MongoDB como base de datos
 - Express: framework web basado en Node.js
 - AngularJS, orientado a la capa de presentación en forma de html enriquecido
 - Node.js, un servidor de aplicaciones en el que se ejecuta la API definida en Express.

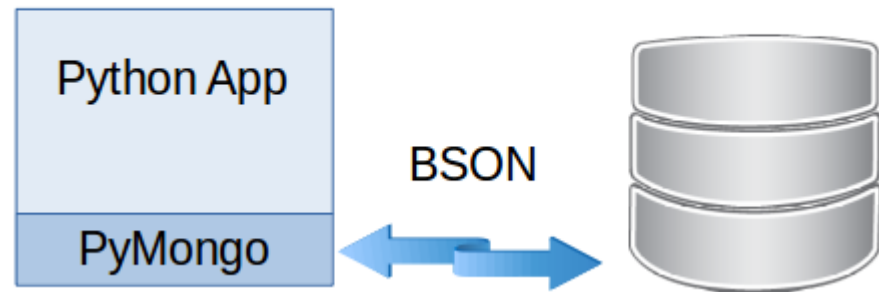
MEAN



Introducción

- El driver más utilizado para conectar Python con MongoDB es **PyMongo**
- Nos proporciona una serie de objetos con los que poder instanciar de forma fácil conexiones, databases, colecciones y trabajar con ellas
- Podemos trabajar directamente con todos los elementos, con una sintaxis similar a la del propio MongoDB

App Architecture



Ejemplo

```
from pymongo import MongoClient  
import pprint
```



Importar librerías

```
#Instanciar Cliente  
client = MongoClient()
```



Crear cliente

```
#Instanciar database  
db = client['got']
```



Instanciar objeto base de datos

```
#Instanciar coleccion  
collection = db['families']
```



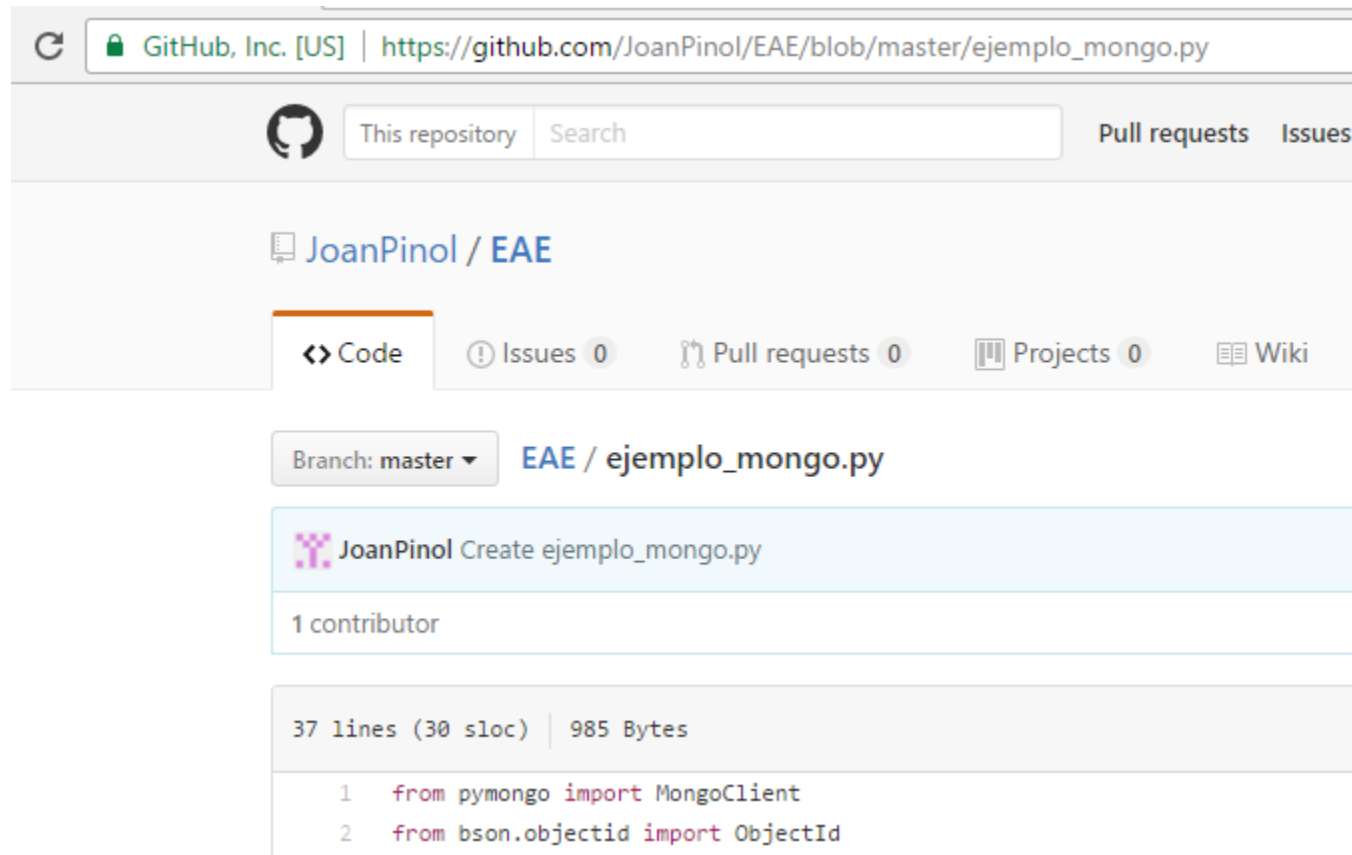
Instanciar familia

```
#Mostrar todo el contenido  
pprint.pprint(collection.find_one())
```



Imprimir el primer resultado

Ejemplo



The screenshot shows a web browser displaying a GitHub repository page. The address bar shows the URL: https://github.com/JoanPinol/EAE/blob/master/ejemplo_mongo.py. The repository name is 'JoanPinol / EAE'. Below the repository name, there are tabs for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), and 'Wiki'. The 'Code' tab is selected. Below the tabs, there is a dropdown menu for 'Branch: master' and the file path 'EAE / ejemplo_mongo.py'. The file was created by 'JoanPinol' and has 1 contributor. The file statistics show '37 lines (30 sloc)' and '985 Bytes'. The code content is as follows:

```
1 from pymongo import MongoClient
2 from bson.objectid import ObjectId
```