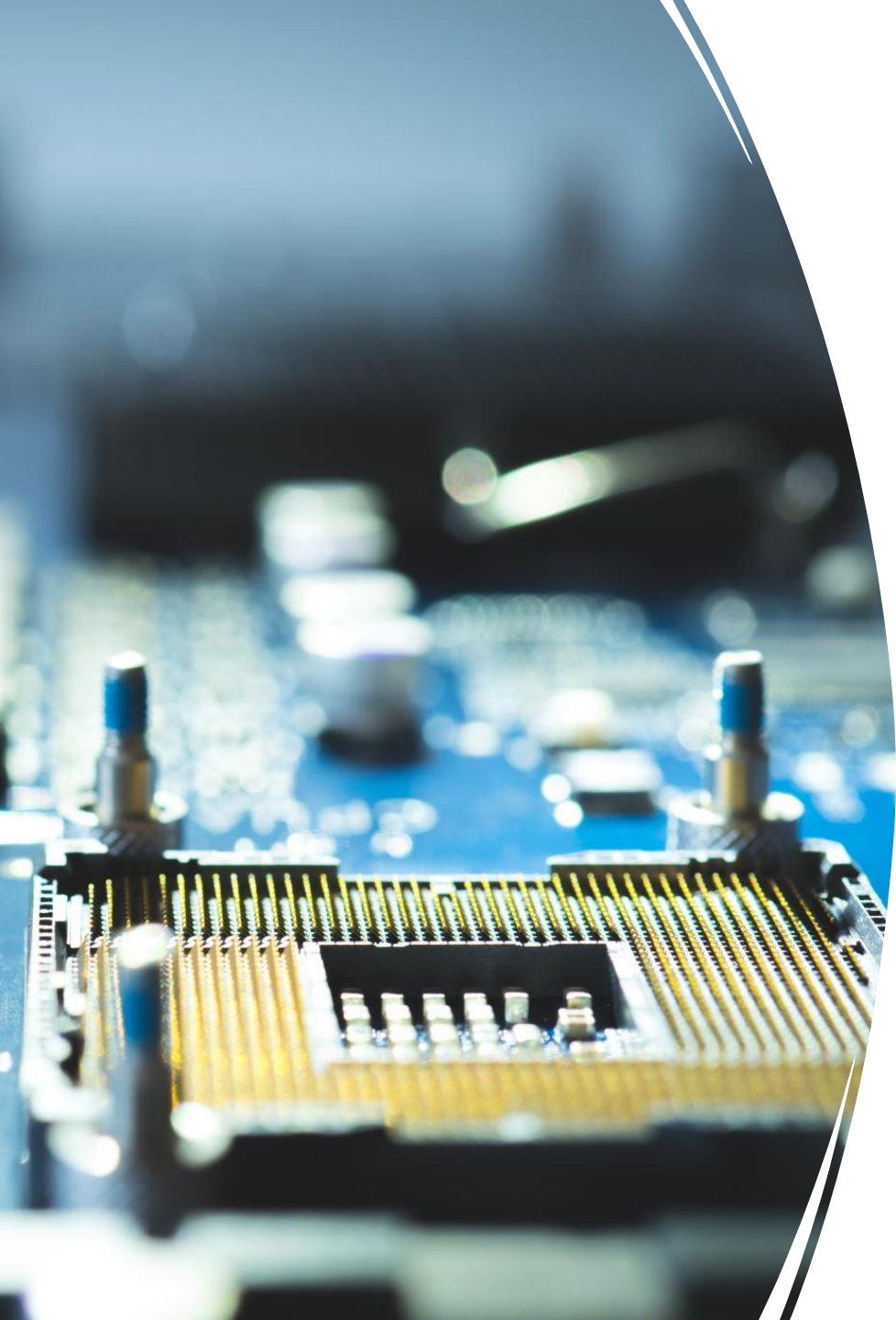


MongoDB



MongoDB

- Lanzado por primera vez públicamente en 2009, MongoDB (a menudo llamado *mongo*) se convirtió rápidamente en una de las bases de datos NoSQL más amplias y utilizadas que existen para el desarrollo de aplicaciones Web.
- MongoDB fue diseñado como una base de datos escalable, el nombre Mongo proviene de "humongous" (extremadamente grande), con rendimiento y fácil acceso a los datos como objetivos principales.
- Es una base de datos de documentos, que le permite almacenar objetos anidados a la profundidad que requerida y consultarlos.
- No se basa en ningún esquema, por lo que los documentos pueden contener campos o tipos que ningún otro documento de la colección contiene.



¿Por qué usar MongoDB?

- Almacenamiento orientado a documentos: los datos se almacenan en forma de documentos de estilo JSON.
- Índice en cualquier atributo
- Replicación y alta disponibilidad
- Fragmentación automática
- Consultas enriquecidas
- Actualizaciones rápidas en el lugar
- Soporte profesional de MongoDB

Ventajas de MongoDB sobre RDBMS

- **Sin esquema.** MongoDB es una base de datos de documentos en la que una colección contiene diferentes documentos. El número de campos, el contenido y el tamaño del documento pueden diferir de un documento a otro.
- **Capacidad de consulta profunda.** MongoDB admite consultas dinámicas en documentos utilizando un lenguaje de consulta basado en documentos que es casi tan potente como SQL.
- **Facilidad de escalabilidad horizontal:** MongoDB es fácil de escalar.

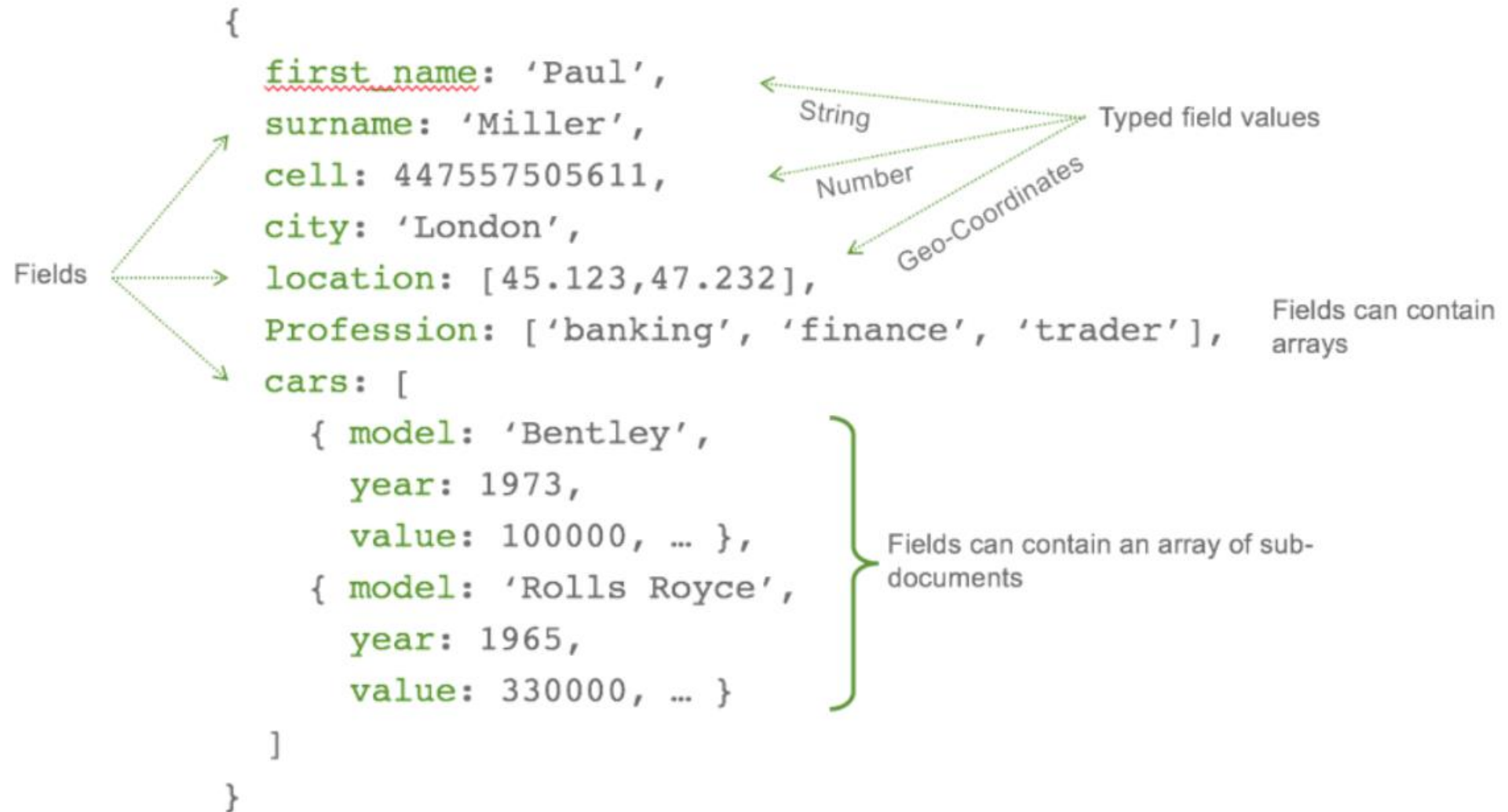
Formato de almacenamiento

- Mongo es una base de datos de documentos JSON (aunque técnicamente los datos se almacenan en una forma binaria de JSON conocida como BSON).
- Un documento de Mongo se puede comparar con una fila de tabla relacional sin un esquema, cuyos valores pueden anidarse a una profundidad arbitraria.

```
{
  "_id" : ObjectId("4d0b6da3bb30773266f39fea"),
  "country" : {
    "$ref" : "countries",
    "$id" : ObjectId("4d0e6074deb8995216a8309e")
  },
  "famous_for" : [
    "beer",
    "food"
  ],
  "last_census" : "Sun Jan 07 2018 00:00:00 GMT -0700 (PDT)",
  "mayor" : {
    "name" : "Ted Wheeler",
    "party" : "D"
  },
  "name" : "Portland",
  "population" : 582000,
  "state" : "OR"
}
```

} JSON document

Estructura de JSON



Reglas de sintaxis JSON

- La sintaxis JSON se deriva de la sintaxis de notación de objetos JavaScript
- Datos JSON: un nombre y un valor
 - Los datos JSON se escriben como pares nombre/valor (también conocidos como pares clave/valor).
 - Un par nombre/valor consiste en un nombre de campo (entre comillas dobles), seguido de dos puntos, seguido de un valor:
`"nombre" : "Juan"`
 - Los nombres JSON requieren comillas dobles.

Valores JSON

- En **JSON**, los *valores* deben ser uno de los siguientes tipos de datos:
 - una cadena
 - un número
 - un objeto
 - una matriz
 - un booleano
 - nulo
- En JSON, *los valores de cadena* deben escribirse entre comillas dobles:

```
{ "nombre" : "Juan" }
```


Cadenas JSON	Las cadenas en JSON deben escribirse entre comillas dobles. <pre>{"nombre":"Juan"}</pre>
Números JSON	Los números en JSON deben ser un entero o un coma flotante. <pre>{"edad":30}</pre>
Objetos JSON	Los valores en JSON pueden ser objetos. <pre>{"empleado":{"nombre":"John", "edad":30, "ciudad":"Nueva York"}}</pre>
Matrices JSON	Los valores en JSON pueden ser matrices. <pre>{"empleados":["Juan", "Ana", "Pedro"]}</pre>
JSON Booleanos	Los valores en JSON pueden ser true/false. <pre>{"venta":true}</pre>
JSON null	Los valores en JSON pueden ser nulos. <pre>{"apellido":null}</pre>

JSON vs XML

- Tanto JSON como XML se pueden utilizar como formato de intercambio de datos.

- Ejemplo JSON

```
{ "empleados": [ {  
  "nombre": "John",   "apellido": "Doe"   }, {  
  "nombre": "Anna",   "apellido": "Smith" }, {  
  "nombre": "Peter",  "apellido": "Jones"  } ] }
```

- Ejemplo XML

```
<empleados>  
  <empleado>  
    <nombre>John</nombre> <apellido>Doe</apellido>  
  </empleado>  
  <empleado>  
    <nombre>Anna</nombre> <apellido>Smith</apellido>  
  </empleado>  
  <empleado>  
    <nombre>Peter</nombre> <apellido>Jones</apellido>  
  </empleado>  
</empleados>
```

JSON es como XML porque

Tanto JSON como XML son "autodescriptibles" (legibles por humanos)

Tanto JSON como XML son jerárquicos (valores dentro de valores)

Tanto JSON como XML pueden ser analizados y utilizados por muchos lenguajes de programación

JSON es diferente a XML porque

JSON no usa la etiqueta final de cierre

JSON es más corto

JSON es más rápido de leer y escribir

JSON puede usar matrices

La mayor diferencia es:

XML debe analizarse con un analizador XML.

JSON puede ser analizado por una función JavaScript estándar (eval).

Porque JSON es mejor que XML

XML es mucho más difícil de analizar que JSON. JSON se analiza en un objeto JavaScript listo para usar.

Para las aplicaciones AJAX, JSON es más rápido y fácil que XML:

Uso de XML

Obtener un documento XML

Utilizar el DOM XML para recorrer el documento en bucle

Extraer valores y almacenar en variables

Uso de JSON

Obtener una cadena JSON

Analizar la cadena JSON

Contenido de la base de datos

- **Base de datos**

- La base de datos es un contenedor físico para colecciones.
- Cada base de datos obtiene su propio conjunto de archivos en el sistema de archivos.
- Un servidor MongoDB normalmente tiene varias bases de datos.

- **Colección**

- La colección es un grupo de documentos de MongoDB.
- Existe una colección dentro de una base de datos.
- Las colecciones no aplican un esquema.
- Los documentos dentro de una colección pueden tener diferentes campos.
- Normalmente, todos los documentos de una colección tienen un propósito similar o relacionado.

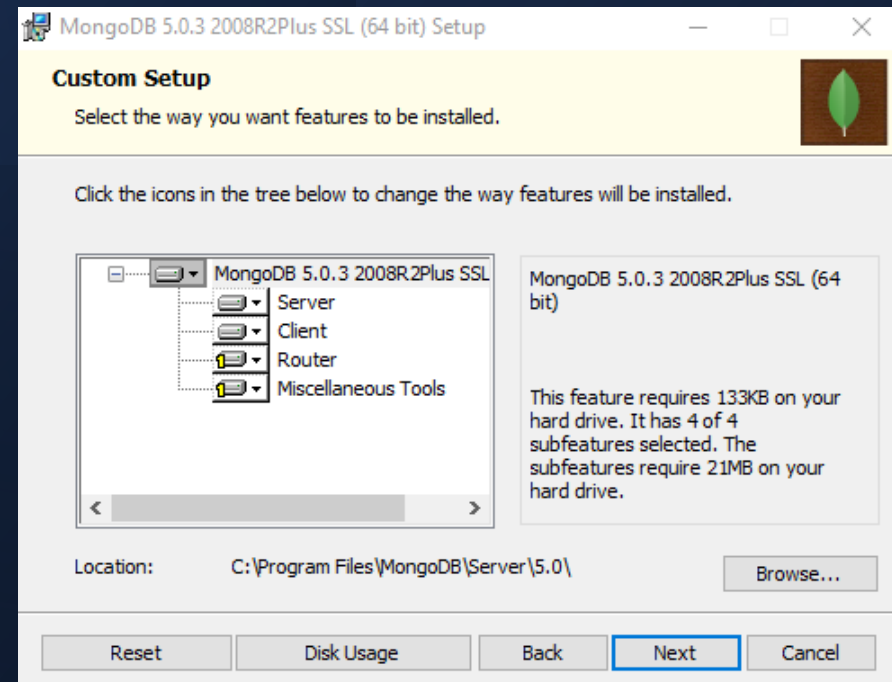
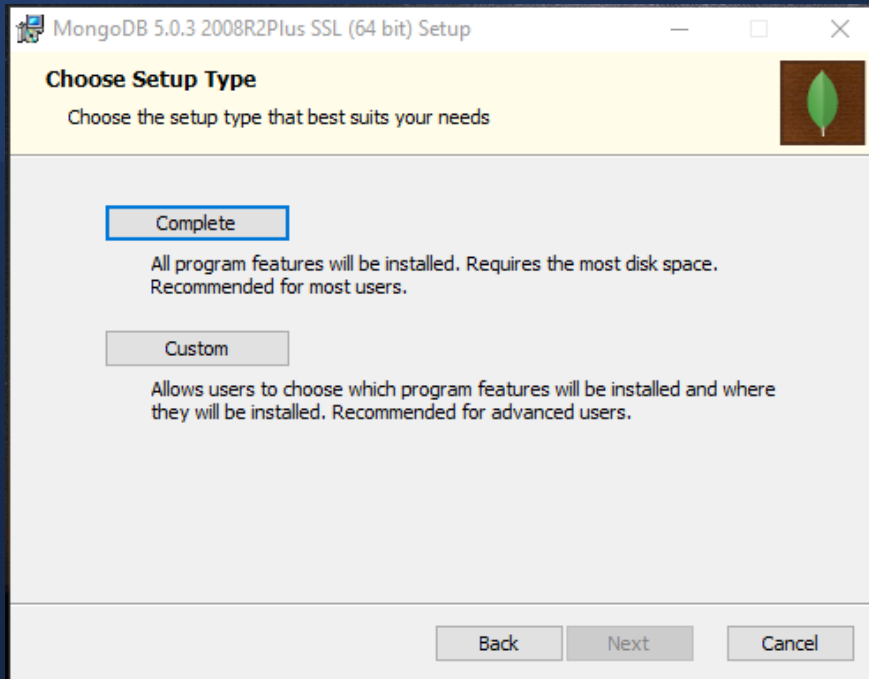
- **Documento**

- Un documento es un conjunto de pares clave-valor.
- Los documentos tienen un esquema dinámico (los documentos de la misma colección no necesitan tener el mismo conjunto de campos o estructura, y los campos comunes en los documentos de una colección pueden contener diferentes tipos de datos).

Relación de concepto con respecto al modelo relacional

RDBMS		MongoDB	
Base de datos		Base de datos	
Tabla		Colección	
Tupla/Fila		Documento	
Columna		Campo	
Reunión de tablas		Documentos incrustados	
Clave principal		Clave principal (clave predeterminada _id proporcionada por mongodb)	

Proceso de instalación



Continuación

MongoDB 5.0.3 2008R2Plus SSL (64 bit) Service Customization

Service Configuration
Specify optional settings to configure MongoDB as a service.

☒ Install MongoDB as a Service

☒ Run service as Network Service user

☐ Run service as a local or domain user:

Account Domain:

Account Name:

Account Password:

Service Name:

Data Directory:

Log Directory:

< Back Next > Cancel

MongoDB Compass

Install MongoDB Compass

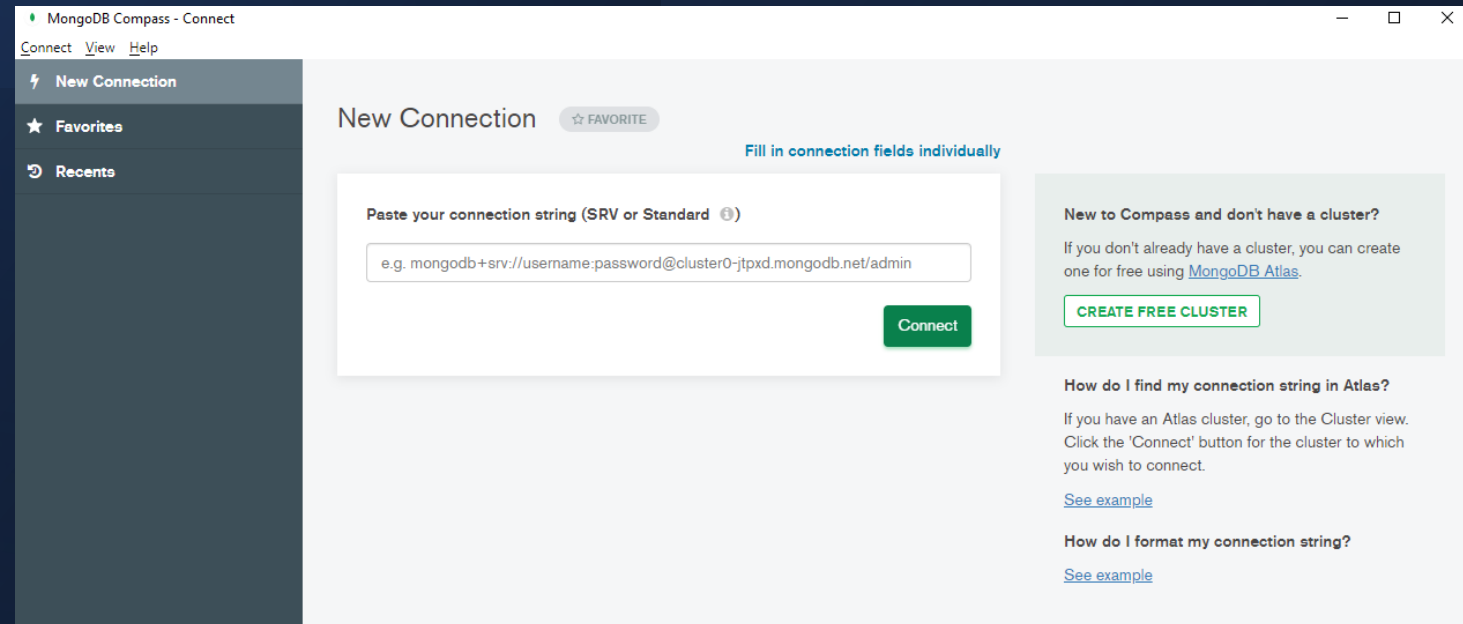
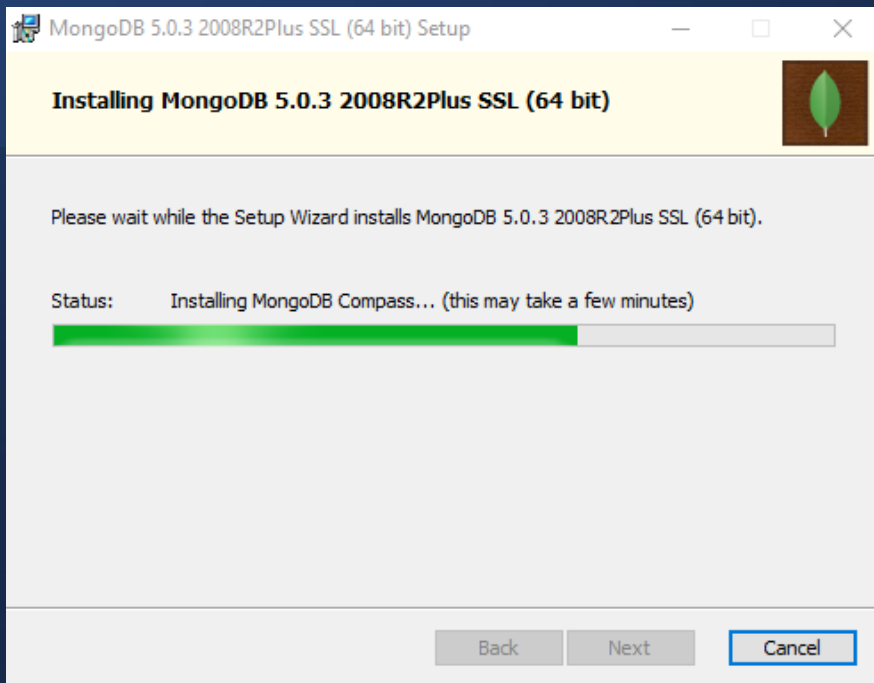
MongoDB Compass is the official graphical user interface for MongoDB.

By checking below this installer will automatically download and install the latest version of MongoDB Compass on this machine. You can learn more about MongoDB Compass here: <https://www.mongodb.com/products/compass>

☒ Install MongoDB Compass

Back Next Cancel

Finalización



Inicialización

- Antes de comenzar, es necesario crear una carpeta (en Windows), en donde se almacenarán los archivos de la base de datos. En una terminal del sistema puede ejecutar los siguientes comandos:

```
C:\>md data
```

```
C:\>md data\db
```

- Este es el valor por omisión y se recomienda mantenerlo.
- Para iniciar el servidor, es necesario ir a la ruta de instalación y ejecutar el archivo mongod.exe

```
C:\>[ruta_instalacion]\cd bin
```

```
C:\>[ruta_instalacion]\bin\mongod.exe
```

- Dejar minimizada esta ventana (no cerrar)
- Abrir otra terminal y ejecutar el cliente

```
C:\>[ruta_instalacion]\bin\mongo.exe
```

```
MongoDB shell version: 2.4.6
```

```
connecting to: test
```

```
>
```

- A partir de aquí comienza la interacción con MongoDB
- El comando `db.help()` regresa un listado de los comandos disponibles en el intérprete de MongoDB

Creación de una base de datos

- Para crear una nueva base de datos, denominada *book*, se ejecuta el comando mongo en la terminal de línea de comandos.

`$ mongo book`

- Actualmente estamos en la base de datos de *book*, pero se pueden ver otras mediante el comando `show dbs` y cambiar la base de datos con el comando `use`. Se puede verificar la base de datos actual con el comando `db`.
- Crear una colección en Mongo es tan fácil como añadir un registro inicial a la colección. Debido a que Mongo no tiene esquema, no hay necesidad de definir nada por adelantado; basta con usarlo. Además, la base de datos *book* no existe realmente hasta que se agreguen valores en ella, mediante la colección llamada `towns`

```
> db.towns.insert({
name: "New York",
population: 22200000,
lastCensus: ISODate("2016-07-01"),
famousFor: [ "the MOMA", "food", "Derek Jeter" ],
mayor : {
  name : "Bill de Blasio",
  party : "D"
}
})
```

Consultas

- Con el comando **show collections**, se puede verificar que la colección ahora existe.

```
> show collections
towns
```

- Se crea la colección `towns` almacenando un objeto en ella. Para enumerar el contenido de una colección se usa la función `find()`.

```
> db.towns.find()

{
  "_id" : ObjectId("59093bc08c87e2ff4157bd9f"),
  "name" : "New York",
  "population" : 22200000,
  "lastCensus" : ISODate("2016-07-01T00:00:00Z"),
  "famousFor" : [ "the MOMA", "food", "Derek Jeter" ],
  "mayor" : {
    "name" : "Bill de Blasio",
    "party" : "I"
  }
}
```

Creación de colecciones

- También es posible crear una colección con el comando `createCollection()` sobre la base de datos en uso.

```
>use test
```

```
switched to db test
```

```
>db.createCollection("mycollection")
```

```
{ "ok" : 1 }
```

```
>show collections
```

```
mycollection
```

- Es preferible usar `createCollections()` cuando se desea establecer los parámetros de configuración de la base de datos, aunque por lo regular se crea automáticamente cuando se inserta el primer documento.

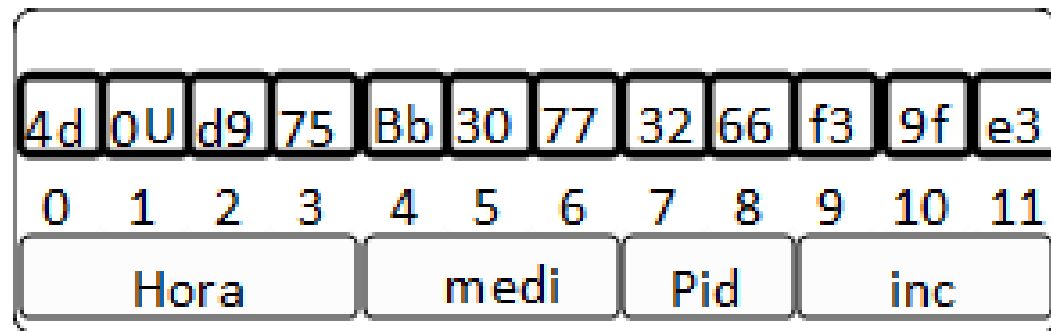
```
>db.createCollection("mycol", { capped  
: true, autoIndexId : true, size  
: 6142800, max : 10000 } )
```

```
{ "ok" : 1 }
```

Campo	Tipo	Descripción
capped	Boolean	(Opcional) Si es true , habilita una colección con límites. La colección de tamaño fijo sobrescribe automáticamente sus entradas más antiguas cuando alcanza su tamaño máximo. Si especifica true , también debe especificar el parámetro de tamaño.
autoIndexId	Boolean	(Opcional) Si es true , crea automáticamente un índice en el campo <code>_id</code> . El valor predeterminado es false .
size	number	(Opcional) Especifica un tamaño máximo en bytes para una colección con límites. Si <i>capped</i> es true , también debe especificar este campo.
max	number	(Opcional) Especifica el número máximo de documentos permitidos en la colección con límite (capped).

Consultas

- La salida JSON contiene un campo `_id` de tipo `ObjectId`. Es similar al tipo SERIAL de una base de datos relacional.
 - El `ObjectId` es siempre de 12 bytes, compuesto por una estampa de tiempo, un ID de equipo cliente, un ID de proceso de cliente y un contador incrementado secuencial de 3 bytes.
- Cada proceso en cada máquina puede manejar su propia generación de ID sin colisionar con otras instancias **mongod**.



JavaScript

- La lengua materna de Mongo es JavaScript.
- **db** es un objeto JavaScript que contiene información sobre la base de datos actual.
- **db.x** es un objeto JavaScript que representa una colección (denominada **x**).
- Los comandos son funciones de JavaScript.

```
> typeof db
```

```
object
```

```
> typeof db.towns
```

```
object
```

```
> typeof db.towns.insert
```

```
function
```

- Es posible crear una función propia que encapsule la inserción de datos en JavaScript:

```
>function insertCity(name, population, lastCensus, famousFor,
mayorInfo) {
>db.towns.insert({
  name: name,
  population: population,
  lastCensus: ISODate(lastCensus),
  famousFor: famousFor,
  mayor : mayorInfo
});
}
```

Llamando la función recién creada desde la terminal se puede invocar a la función `insertCity()` con los valores a insertar:

```
> insertCity("Portland", 582000, '2016-09-20', ["beer",
"food", "Portlandia"], { name : "Ted Wheeler", party : "D" })

> insertCity("Punxsutawney", 6200, '2016-01-
31', ["Punxsutawney Phil"], { name : "Richard Alexander"})
```

- Si se desea inspeccionar el código fuente de una función, es necesario invocarla sin parámetros ni paréntesis

> **db.towns.insert**

```
function (obj, options, _allowDot) {  
  if (!obj)  
    throw Error("no object passed to  
    insert!");  
  var flags = 0;  
  // etc.  
}
```


- Anteriormente se usó la función `find()` sin parámetros para obtener todos los documentos. Para acceder a uno específico, es necesario establecer una consulta en el campo `_id`.
- `_id` es de tipo `ObjectId`, por lo tanto, para realizar consultas, se debe convertir una cadena de texto con la función `ObjectId(str)`.

```
> db.towns.find({
  "_id" : ObjectId("59094288afbc9350ada6b807") })
{
  "_id" : ObjectId("59094288afbc9350ada6b807"),
  "name" : "Punxsutawney",
  "population" : 6200,
  "lastCensus" : ISODate("2016-01-31T00:00:00Z"),
  "famousFor" : [ "Punxsutawney Phil" ],
  "mayor" : { "name" : "Richard Alexander" }
}
```

- La función `find()` también acepta un segundo parámetro opcional: un objeto *fields* para filtrar los campos a recuperar.
 - Para tener solo el nombre de la ciudad (junto con `_id`), hay pasar el nombre deseado con el valor a `1` (o `true`).

```
> db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, { name: 1 })
{
  "_id" : ObjectId("59093e9eafbc9350ada6b803"),
  "name" : "Punxsutawney"
}
```

- Para recuperar todos los campos excepto *name*, hay que establecer *name* en `0` (o `false`).

```
> db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, { name : 0 })
{
  "_id" : ObjectId("59093e9eafbc9350ada6b803"),
  "población" : 6200,
  "lastCensus" : ISODate("2016-01-31T00:00:00Z"),
  "famousFor" : [ "Punxsutawney Phil" ]
}
```

- En Mongo se pueden construir consultas sobre los campos, rangos o una combinación de criterios.
- Para encontrar todas las ciudades que comienzan con la letra *P* y tienen una población inferior a 10,000, se puede usar una expresión regular y un operador de rango.
 - La consulta devuelve el objeto JSON para Punxsutawney, pero incluyendo sólo los campos de nombre y población:

```
> db.towns.find({ nombre : /^P/, población : { $lt : 10000 } },  
{ _id: 0, nombre : 1, población : 1 })  
{ "nombre" : "Punxsutawney", "población" : 6200 }
```

Notación de Regex

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the <i>beginning</i> of the string
abc\$	abc at the <i>end</i> of the string
a b	either of a and b
^abc abc\$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does <i>not</i> contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\b	perl when <i>not</i> followed by a word boundary (e.g. in perlert but not in perl stuff)

- Los operadores condicionales en Mongo siguen el formato de `field : { $op : valor }`, donde `$op` es una operación como `$ne` (no igual a) o `$gt` (mayor que).
- Es posible construir operador como objetos de JavaScript, como en los siguientes criterios donde la población debe estar entre 10,000 y 1 millón de personas.

```
> var population_range = {  
  $lt: 1000000,  
  $gt: 10000  
}
```

```
> db.towns.find(  
  { name : /^P/, population : population_range },  
  { name: 1 }  
)
```

```
{ "_id" : ObjectId("59094292afbc9350ada6b808"), "name" :  
  "Portland" }
```

Operadores de comparación

OPERADOR	DESCRIPCIÓN
\$regex	Coincidencia por cualquier cadena de expresión regular compatible con PCRE (o usar los delimitadores //)
\$ne	No es igual a
\$lt	Menor que
\$lte	Menor o igual que
\$gt	Mayor que
\$gte	Mayor o igual que
\$exists	Comprobar la existencia de un campo
\$all	Hacer coincidir todos los elementos de una matriz
\$in	Hacer coincidir cualquier elemento de una matriz
\$nin	No coincide con ningún elemento de una matriz
\$elemMatch	Coincidencia de todos los campos en un arreglo de documentos anidados
\$or	o
\$nor	No o
\$size	Coincidencia de arreglo de tamaño dado
\$mod	Operador módulo
\$type	Coincidencia si el campo es de una tipo de dato dado
\$not	Negación de la comparación

Operación	Sintaxis	Ejemplo	Equivalente
Igualdad	{<key>:<value>}	db.mycol.find({"by":"Fred"})	where by = 'Fred'
Menos que	{<key>:{\$lt:<value>}}	db.mycol.find({"likes":{\$lt:50}})	where likes < 50
Menos que o igual	{<key>:{\$lte:<value>}}	db.mycol.find({"likes":{\$lte:50}})	where likes < = 50
Mayor que	{<key>:{\$gt:<value>}}	db.mycol.find({"likes":{\$gt:50}})	where likes > 50
Mayor que o igual	{<key>:{\$gte:<value>}}	db.mycol.find({"likes":{\$gte:50}})	where likes > = 50
No es igual	{<key>:{\$ne:<value>}}	db.mycol.find({"likes": {\$ne:50}})	where likes != 50

Comparativa de operadores RDBMS con MongoDB

- Para consultas con rangos de datos también es posible utilizar los operadores de comparación, como en el siguiente ejemplo, que obtiene los nombres de ciudades con censos posteriores al 1 de junio de 2016:

```
> db.towns.find(  
  { lastCensus : { $gte : ISODate('2016-06-01') } }, { _id : 0,  
  name: 1 }  
)  
{"name" : "New York" }  
{"name" : "Portland" }
```


AND en MongoDB

- En el método **find()**, si se pasan varias claves separándolas por ',' (coma), entonces MongoDB lo trata como condición **AND**.

```
>db.mycol.find(  
  {  
    $and: [  
      {key1: value1}, {key2:value2}  
    ]  
  }  
)
```

- Ejemplo

```
>db.mycol.find({$and:[{"by":"Fred"}, {"title": "MongoDB  
Overview"}]}).pretty()  
{  
  "_id": ObjectId(7df78ad8902c),  
  "title": "MongoDB Overview",  
  "description": "MongoDB is no sql database",  
  "by": "Fred",  
  "tags": ["mongodb", "database", "NoSQL"]  
}
```

OR en MongoDB

- Para consultar documentos basados en la condición OR, debe usar la palabra clave **\$or**.

```
>db.mycol.find(  
  {  
    $or: [  
      {key1: value1}, {key2:value2}  
    ]  
  }  
)
```

- Ejemplo

```
>db.mycol.find({$or:[{"by":"Fred"}, {"title": "MongoDB  
Overview"}]}).pretty()  
  
{  
  "_id": ObjectId(7df78ad8902c),  
  "title": "MongoDB Overview",  
  "description": "MongoDB is no sql database",  
  "by": "Fred",  
  "tags": ["mongodb", "database", "NoSQL"]  
}
```

- Es posible realizar consultas para valores exactos:

```
> db.towns.find(
{ famousFor : 'food' },
{ _id : 0, name : 1, famousFor : 1 }
)
{"name" : "New York", "famousFor" : [ "the MOMA", "food",
"Derek Jeter" ] }
{ "name" : "Portland", "famousFor" : [ "beer", "food",
"Portlandia" ] }
```

- Asi como para valores parciales:

```
> db.towns.find(
{ famousFor : /MOMA/ },
{ _id : 0, name : 1, famousFor : 1 }
)
{"name" : "New York", "famousFor" : [ "the MOMA", "food" ]
}
```

- Obtener todos los valores coincidentes:

```
> db.towns.find(
{ famousFor : { $all : ['food', 'beer'] } },
{ _id : 0, name:1, famousFor:1 }
)

{"name" : "Portland", "famousFor" : [ "beer", "food",
"Portlandia" ] }
```

- O la ausencia de coincidencia:

```
> db.towns.find(
{ famousFor : { $nin : ['food', 'beer'] } },
{ _id : 0, name : 1, famousFor : 1 }
)

{"name" : "Punxsutawney", "famousFor" : [ "Punxsutawney
Phil" ] }
```

- La búsqueda de datos dentro de subdocumentos anidados se realiza mediante el operador . (punto)

```
> db.towns.find(  
  { 'mayor.party' : 'D' },  
  { _id : 0, name : 1, mayor : 1 }  
)
```

```
{ "name" : "New York", "mayor" : { "name" : "Bill de Blasio",  
  "party" : "D" } }  
{ "name" : "Portland", "mayor" : { "name" : "Ted Wheeler", "party"  
  : "D" } }
```

- O aplicando operadores de comparación

```
> db.towns.find(  
  { 'mayor.party' : { $exists : false } },  
  { _id : 0, name : 1, mayor : 1 }  
)
```

```
{ "name" : "Punxsutawney", "mayor" : { "name" : "Richard Alexander"  
  }}  
}}
```

```

> db.countries.insert({
  id : "us",
  name : "United States",
  exports : {
    foods : [
      { name : "bacon", tasty :
true },
      { name : "burgers" }
    ]
  }
})
> db.countries.insert({
  id : "ca",
  name : "Canada",
  exports : {
    foods : [
      { name : "bacon", tasty :
false },
      { name : "syrup", tasty :
true }
    ]
  }
})

```

```

> db.countries.insert({
  _id : "mx",
  name : "Mexico",
  exports : {
    foods : [{
      name : "salsa",
      tasty : true,
      condiment : true
    }]
  }
})

```

- Se pueden establecer valores de _id que no sean generados por el sistema y de tipo texto.

- Para mejorar la búsqueda de criterios en campos del documento, es posible usar el operador `$elemMatch`:

```
>db.countries.find(  
  {  
    'exports.foods' : {  
      $elemMatch : {name : 'bacon',tasty : true}  
    }  
  }, { _id : 0, name : 1 }  
)  
{"name" : "United States" }
```

- También es posible incluir operadores en `$elemMatch`:

```
>db.countries.find(
{
  'exports.foods' : {
    $elemMatch : {tasty : true, condiment :{$exists : true}}
  }, { _id : 0, name : 1 })
{"name" : "Mexico" }
```

- Para realizar comparaciones o (or), es necesario establecer los valores como un arreglo:

```
>db.countries.find(
{
  $or : [{ _id : "mx" }, { name : "United States" }]
},{ _id:1})
{"_id" : "us" }
{"_id" : "mx" }
```


Actualización

- La función `update(criteria, operation)` requiere dos parámetros.
 - El primero es una consulta de criterios, el mismo tipo de objeto que pasaría a `find()`.
 - El segundo parámetro es un objeto cuyos campos reemplazarán a los documentos coincidentes o una operación modificadora. En este caso, el modificador `$set` del campo `state` con la cadena `OR`.

```
db.towns.update(  
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },  
  { $set : { "state" : "OR" } }  
);
```

- NOTA. El operador `$set` es necesario ya que Mongo trabaja a nivel de documentos, y no necesariamente un solo valor, por lo que omitir este comando actualizaría todo el documento coincidente con `{ "state" : "OR" }`

Comando usados en actualización

Comando	Descripción
\$set	Establece el campo dado con el valor dado
\$unset	Quita el campo
\$inc	Incrementa el campo dado por el número dado
\$pop	Quita el último (o el primer) elemento de un arreglo
\$push	Agrega el valor a un arreglo
\$pushAll	Agrega todos los valores a un arreglo
\$addToSet	Similar a push, pero no duplica valores
\$pull	Quita los valores coincidentes de un arreglo
\$pullAll	Quita todos los valores coincidentes de un arreglo

-
- Para incrementar la población de Portland en 1,000.

```
db.towns.update(  
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },  
  { $inc : { población : 1000 } }  
)
```

El método Save()

- El método **save()** reemplaza el documento existente por el nuevo documento pasado como parámetro.

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

- Ejemplo

```
>db.mycol.save(  
    {  
        "_id" : ObjectId(5983548781331adf45ec5),  
        "title":"MongoDB",  
        "by":"Fred"  
    }  
)
```

Búsqueda de un solo registro

```
db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
{"_id" : ObjectId("4d0ada87bb30773266f39fe5"),
"famousFor" : [
"beer",
"food",
"Portlandia"
],
"lastCensus" : "Thu Sep 20 2017 00:00:00 GMT-0700 (PDT)",
"mayor" : {
"name" : "Sam Adams",
"party" : "D"
},
"name" : "Portland",
"population" : 582000,
"state" : "OR"
}
```

Limitar registros

- Para limitar los registros en MongoDB, se debe utilizar el método **limit()**. El método acepta un argumento de tipo numérico, que es el número de documentos que desea que se muestren.

```
>db.mycol.find({}, {"title":1, _id:0}).limit(2)
```

```
{"title":"MongoDB Overview"}
```

```
{"title":"NoSQL Overview"}
```

```
>
```

Saltar registros

- Además del método `limit()`, hay un método más `skip()` que también acepta el argumento de tipo de número y se utiliza para omitir el número de documentos. Se emplea junto con `limit()`

```
>db.mycol.find({}, {"title":1, _id:0}).limit(1).skip(1)
```

```
{"title":"NoSQL Overview"}
```

```
>
```

Ordenamiento de registros

- Para ordenar documentos en MongoDB, debe usar el método `sort()`. El método acepta un documento que contiene una lista de campos junto con su orden de clasificación.
- Para especificar el orden de clasificación se utilizan 1 y -1.
 - 1 se usa para el orden ascendente, mientras que -1 se usa para el orden descendente.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```


Eliminación de documentos

- La llamada a `remove()` eliminará todos los documentos que coincidan con los criterios.
 - Es importante tener en cuenta que se eliminará todo el documento coincidente, no solo un elemento o un subdocumento coincidente.
 - Es mejor ejecutar `find()` para verificar los criterios antes de ejecutar `remove()`.

```
> var badBacon = { 'exports.foods'
: { $elemMatch : { name : 'bacon', tasty :
false }
}
}

> db.countries.find(badBacon)
{ "_id" :
ObjectId("4d0b7b84bb30773266f39fef") ,
"name" : "Canada", "exports" : {
"foods" : [{ "name" : "bacon", "tasty" :
false }, { "name" : "syrup", "tasty" :
true } ]
}
}

> db.countries.remove(badBacon)
> db.countries.count()
2
```

Eliminación de colecciones

- MongoDB utiliza el comando `db.collection.drop()` para eliminar una colección de la base de datos.
- Primero se recomienda revisar la colección existente y posteriormente eliminarla

```
>use mydb
```

```
switched to db mydb
```

```
>show collections
```

```
mycollection
```

```
system.indexes
```

```
>db.mycollection.drop()
```

```
true
```

Eliminación de la base de datos

- El método `dropDatabase()` permite la eliminación completa de una base de datos. Debe tener en uso la base de datos a eliminar, en caso contrario eliminará la base de datos test.

```
>use book
```

```
switched to db mydb
```

```
>db.dropDatabase()
```

```
>{ "dropped" : "mydb", "ok" : 1 }
```

```
>show databases
```

```
local          0.78125GB
```

```
test           0.23012GB
```

Indexación

- MongoDB debe escanear todos los documentos de una colección para seleccionar aquellos documentos que coincidan con la instrucción de consulta. Este escaneo es altamente ineficiente y requiere que MongoDB procese un gran volumen de datos.
- Los índices son estructuras de datos especiales, que almacenan una pequeña parte del conjunto de datos en una forma fácil de recorrer. El índice almacena el valor de un campo específico o conjunto de campos, ordenado por el valor del campo como se especifica en el índice.
- MongoDB proporciona varias de las mejores estructuras de datos para la indexación, como el clásico árbol B, así como otras adiciones, como los índices geoespaciales bidimensionales y esféricos.

El método ensureIndex()

- Para crear un índice, debe utilizar el método **createIndex()** de MongoDB.

```
>db.mycol.createIndex({"title":1})
```

- El nombre del campo en el que desea crear el índice y 1 es para el orden ascendente. Para crear un índice en orden descendente, debe usar -1.

```
db.mycol.createIndex({"title":1,  
"description":-1})
```

```
>
```

Ejemplo

- Se generarán como ejemplo 100,000 números de teléfonos entre *1-800-555-0000* y *1-800-565-0000*, con el siguiente código:

```
populatePhones = function(area, start, stop) {
  for(var i = start; i < stop; i++) {
    var country = 1 + (Math.random() * 8) << 0);
    var num = (country * 1e10) + (area * 1e7) + i;
    var fullNumber = "+" + country + " " + area + "-" + i;
    db.phones.insert({_id: num, components: {country:
country, area: area, prefix: (i * 1e-4) << 0, number: i},
    display: fullNumber}

  );
  print("Inserted number " + fullNumber);
}
print("Done!");
}
```

- Ejecutando la función anterior, con un rango de 5,550,000 a 5,650,000 para los números telefónicos:

```
> populatePhones(800, 5550000, 5650000) // This could take a minute
> db.phones.find().limit(2)
{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
"prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
"prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001" }
```

- Cada vez que se crea una nueva colección, Mongo crea automáticamente un índice para el `_id`. Estos índices se pueden encontrar en la colección `system.indexes`. La siguiente consulta muestra todos los índices de la base de datos:

```
> db.getCollectionNames().forEach(function(collection) {
print("Indexes for the " + collection + " collection:");
printjson(db[collection].getIndexes());
});
```

Creación de un índice

- Se va a crear un índice de árbol B en el campo *display*.
- Primero se verificará la velocidad de ejecución. El método `explain()` se utiliza para generar detalles de una operación determinada.

```
> db.phones.find({display: "+1 800-5650001"}).explain("executionStats").executionStats
{
  "executionTimeMillis": 52,
  "executionStages": {"executionTimeMillisEstimate": 58,}
}
```

- Se crea el índice en la colección.

```
> db.phones.createIndex({ display : 1 }, { unique : true, dropDups : true })
```

- El parámetro `fields` es un objeto que contiene los campos con los que se va a indexar. El parámetro `options` describe el tipo de índice que se debe realizar. En este caso, se está creando un índice único en `display` que debería eliminar entradas duplicadas.

- Ahora se repite la ejecución de la búsqueda de datos basado en el índice creado anteriormente y se comparan los tiempos de respuesta

```
> db.phones.find({ display: "+1 800-5650001"
}) .explain("executionStats") .executionStats
{
  "executionTimeMillis" : 0,
  "executionStages": {
    "executionTimeMillisEstimate": 0,
  }
}
```

- Se observa una mejora sustancial en los resultados.
- Si desea indexar en un campo anidado, se debe usar con anotación de puntos: `components.area`. En producción, siempre debe crear índices en segundo plano utilizando la opción `{ background : 1 }`.

```
> db.phones.createIndex({ "components.area": 1
}, { background : 1 })
```

count() y distinct()

- Por ejemplo, para obtener el conteo de números telefónicos mayores a 5599999:

```
> db.phones.count({'components.number': {  
$gt : 5599999 }})  
50000
```

- El método `distinct()` devuelve cada valor coincidente (no documentos completos) donde exista uno o más.
- Para obtener los distintos números que son menores de 5,550,005:

```
> db.phones.distinct('components.number',  
{'components.number': { $lt : 5550005 } })  
[ 5550000, 5550001, 5550002, 5550003,  
5550004 ]
```

Agregación

- Las operaciones de agregación procesan registros de datos y devuelven resultados calculados.
- Las operaciones de agregación agrupan los valores de varios documentos juntos y pueden realizar una variedad de operaciones en los datos agrupados para devolver un único resultado.
- Para la agregación en MongoDB, debe utilizar el método **aggregate()**
 - Permite especificar una lógica en estilo tubería que consta de etapas tales como:
 - **\$match** - filtros que devuelven conjuntos específicos de documentos;
 - **\$group** - funciones que agrupan con base de algún atributo;
 - **\$sort()** - ordena los documentos por una clave de ordenación; y otros más.

Ejemplo

- Suponer un documento con la siguiente composición:

```
{"_id" : ObjectId("5913ec4c059c950f9b799895"),  
"name" : "Sant Julià de Lòria",  
"country" : "AD",  
"timezone" : "Europe/Andorra",  
"population" : 8022,  
"location" : {"longitude" : 42.46372, "latitude" : 1.49129}  
}
```

- Para encontrar la población promedio de las ciudades en determinada zona (Europe/London):

```
> db.cities.aggregate([  
  {$match: {'timezone': {$eq: 'Europe/London'}}},  
  {$group: {_id: 'averagePopulation', avgPop:  
    {$avg: '$population'}}}  
])  
{ "_id" : "averagePopulation", "avgPop" :  
  23226.22149712092 }
```

- Observe la definición del documento con el resultado en los campos `_id` y `avgPop`

- Es posible agregar más operaciones, como ordenamiento y proyección:

```
> db.cities.aggregate([
  {$match: {'timezone': {$eq: 'Europe/London'}}},
  {$sort: {population: -1}},
  {$project: {_id: 0, name: 1, population: 1}}
])

{ "name" : "City of London", "population" :
7556900 }
{ "name" : "London", "population" : 7556900 }
{ "name" : "Birmingham", "population" : 984333 }
```

Expresión	Descripción	Ejemplo
\$sum	Resume el valor definido de todos los documentos de la colección.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}]))
\$avg	Calcula el promedio de todos los valores dados de todos los documentos de la colección.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}]))
\$min	Obtiene el mínimo de los valores correspondientes de todos los documentos de la colección.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}]))
\$max	Obtiene el máximo de los valores correspondientes de todos los documentos de la colección.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}]))
\$push	Inserta el valor en una matriz del documento resultante.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push: "\$url"}}}]))
\$addToSet	Inserta el valor en una matriz del documento resultante, pero no crea duplicados.	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}]))
\$first	Obtiene el primer documento de los documentos de origen según la agrupación. Por lo general, esto solo tiene sentido junto con alguna etapa de "\$sort" aplicada anteriormente.	db.mycol.aggregate([{\$group : {_id : "\$by_user", first_url : {\$first : "\$url"}}}]))
\$last	Obtiene el último documento de los documentos de origen según la agrupación. Por lo general, esto solo tiene sentido junto con alguna etapa de "\$sort" aplicada anteriormente.	db.mycol.aggregate([{\$group : {_id : "\$by_user", last_url : {\$last : "\$url"}}}]))

MapReduce en MongoDB

- En MongoDB, el paso de **map** implica la creación de una función de mapeo que llama a una función **emit()**. El beneficio de este enfoque es que puede emitir más de una vez por documento.
- La función **reduce()** acepta una sola clave y una lista de valores que se emitieron para esa clave.
- Finalmente, MongoDB proporciona un tercer paso opcional llamado **finalize()**, que se ejecuta solo una vez por valor mapeado después de ejecutar los reductores. Esto le permite realizar cualquier cálculo final o limpiar lo necesario.

- Como ejemplo, se generará un informe que cuente todos los números de teléfono que contienen los mismos dígitos para cada país.
- Primero, se almacenará una dirección auxiliar que extrae una matriz de todos los números distintos.

```
distinctDigits = function(phone){  
  var number = phone.components.number + "",  
  seen = [],  
  result = [],  
  i = number.length;  
  while(i--){seen[+number[i]] = 1;}  
  for (var i = 0; i < 10; i++) {  
    if (seen[i]) {result[result.length] = i;}  
  }  
  return result;  
}  
> db.system.js.save({_id: 'distinctDigits', value: distinctDigits})
```


- Debido a que el informe está encontrando números distintos, la matriz de valores distintos es un campo. Pero debido a que también es necesario consultar por país, se considera como otro campo, por lo que se añaden ambos valores como clave compuesta: {digits: X, country: Y}.
- El trabajo del reducer es sumar todos los 1 encontrados.

```
map = function() {  
  var digits = distinctDigits(this);  
  emit({digits: digits, country: this.components.country}, {count : 1});  
}
```

```
reduce = function(key, values) {  
  var total = 0;  
  for (var i = 0; i < values.length; i++) {total += values[i].count;}  
  return { count : total };  
}
```

```
results = db.runCommand({  
  mapReduce: 'phones',  
  map: map,  
  reduce: reduce,  
  out: 'phones.report'  
})
```

-
- Debido a que se establece el nombre de la colección a través del parámetro out (out: 'phones.report'), se puede consultar el resultado como cualquier colección.

```
> db.phones.report.find({'_id.country' : 8})
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },
  "value" : { "count" : 19 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },
  "value" : { "count" : 3 }
}
```

Replicación

La replicación es el proceso de sincronización de datos entre varios servidores.

La replicación proporciona redundancia y aumenta la disponibilidad de los datos con varias copias de datos en diferentes servidores de bases de datos.

La replicación protege una base de datos de la pérdida de un único servidor.

La replicación también le permite recuperarse de fallas de hardware e interrupciones del servicio.

Con copias adicionales de los datos, puede dedicar una a la recuperación ante desastres, la generación de informes o la copia de seguridad.

Client Application

Driver

Writes

Reads

Primary

Replication

Replication

dary

Sec

Conjuntos de réplicas

- Mongo fue diseñado para escalar horizontalmente, no para ejecutarse en modo independiente.
- Fue construido para la consistencia de los datos y la tolerancia de particionamiento, pero la fragmentación de datos tiene un costo: si se pierde una parte de una colección, todo se ve comprometido.
- Rara vez se debe ejecutar una sola instancia de Mongo en producción; en su lugar, se deben replicar los datos almacenados en varios servicios.
- Un conjunto de réplicas es un grupo de instancias **mongod** que alojan el mismo conjunto de datos.
- En una réplica, un nodo es el nodo principal que recibe todas las operaciones de escritura.
- Todas las demás instancias, como las secundarias, aplican operaciones desde el primario para que tengan el mismo conjunto de datos.
 - El conjunto de réplicas solo puede tener un nodo principal.

Consideraciones de la replicación

- El conjunto de réplicas es un grupo de dos o más nodos (generalmente se requieren un mínimo de 3 nodos).
- En un conjunto de réplicas, un nodo es el nodo primario y los restantes son secundarios.
- Todos los datos se replican desde el nodo primario al secundario.
- En el momento de la conmutación automática por error o mantenimiento, se elige un nuevo nodo primario.
- Después de la recuperación del nodo fallido, se une nuevamente al conjunto de réplicas y funciona como un nodo secundario.

- Se simula un ambiente distribuido de varios servidores, ejecutando en terminales distintas.
 - El puerto predeterminado de Mongo es 27017, por lo que se inicia cada servidor en otros puertos.
 - Se recomienda crear un directorio individual para almacenamiento por cada uno de los servidores (p.e. c:\data\srv1, c:\data\srv2, etc.)

- Para levantar el servicio servidor en cada terminal se usa el siguiente comando:

```
mongod --port "PORT" --dbpath "DB_DATA_PATH" --replSet  
"REPLICA_SET_INSTANCE_NAME"
```

- En donde se tienen que establecer los valores de PORT (número de puerto de escucha), DB_DATA_PATH (ruta del directorio para los datos) y REPLICA_SET_INSTANCE_NAME (nombre que identifica a la replicación)

- Para iniciar los servidores con replicación "empresa" (hay que asegurarse que los puertos indicados no estén asignados a otro proceso)

```
$ mongod --replSet empresa --dbpath c:\data\rep1 --port 27001
```

```
$ mongod --replSet empresa --dbpath c:\data\rep2 --port 27002
```

```
$ mongod --replSet empresa --dbpath c:\data\rep3 --port 27003
```

- A continuación hay que abrir una terminal para la ejecución del servidor primario, mediante el comando

```
$ mongo localhost:27001
> rs.initiate({
  id: "empresa",
  members: [
    { _id: 1, host: "localhost:27001" },
    { _id: 2, host: "localhost:27002" },
    { _id: 3, host: "localhost:27003" }
  ]
})
> rs.status().ok
```

- Observe que en el arreglo **members** se establecen los servidores que van a participar en la replicación, con su identificador y su dirección IP con su puerto.
- El objeto `rs` se refiere al conjunto de replicas en uso (replica set)
- el comando `status()` nos permitirá saber cuándo se está ejecutando nuestro conjunto de réplicas

- Cada uno de las consolas mostrará un mensaje similar a

```
Member ... is now in state PRIMARY
```

```
Member ... is now in state SECONDARY
```

- dependiendo de cómo haya sido elegido de entre los nodos participantes (lo más probable es que sea el primero en la lista, con puerto 27001).
- Para comprobar la replicación, en la terminal del nodo designado como primario, se ejecuta una instrucción de mensaje:

```
> db.echo.insert({ say : 'HELLO!' })
```

y posteriormente se termina la consola (CTRL + C), y se observa que los mensajes de los otros nodos indican que alguno de ellos se promovió a primario.

- En esa consola, ejecutar

```
db.echo.find() ()
```

- y se debe observar el valor enviado

- Para saber el estado y la configuración del nodo principal, se ejecutan:

```
> db.status()
```

```
> db.conf()
```

- Si se trata de insertar un valor en un nodo que no es el primario, se lanzará un error como el siguiente:

```
WriteResult({ "writeError" : { "code" : 10107, "errmsg" :  
"not master" } })
```

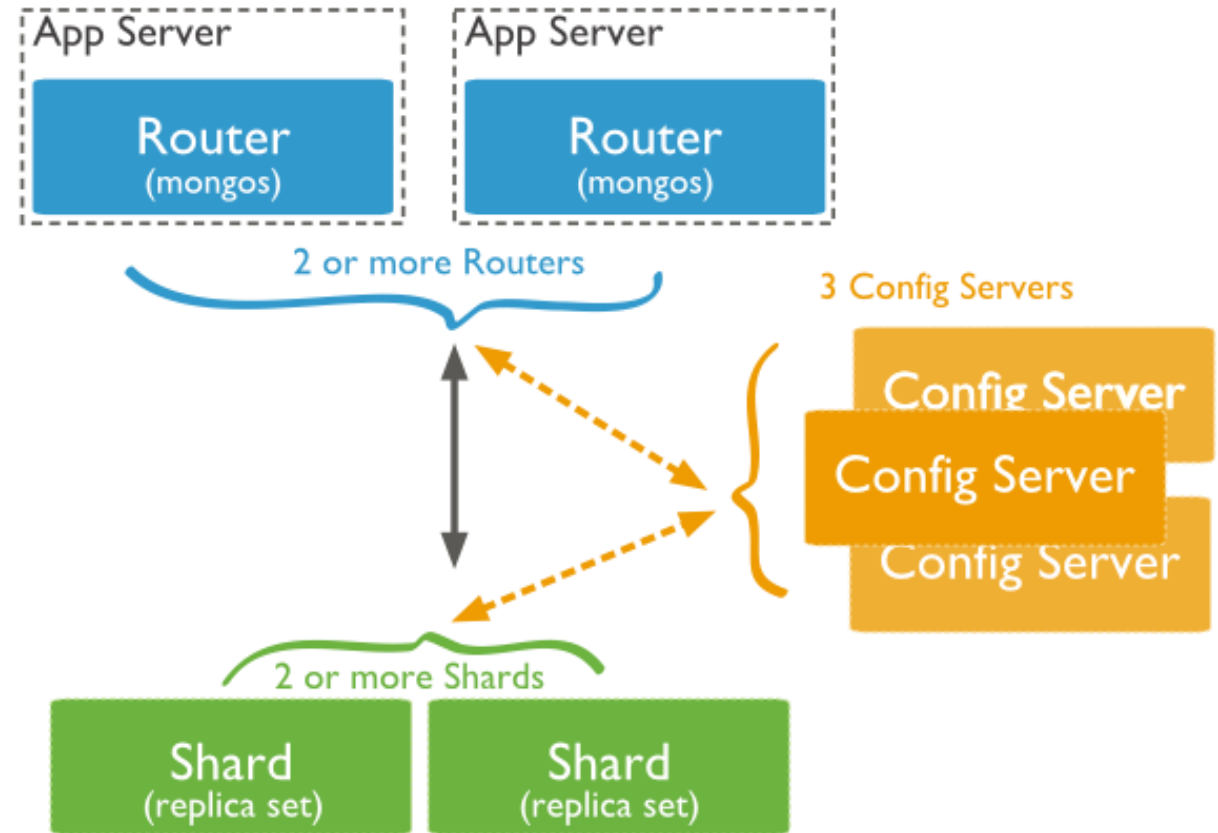
Fragmentación

Uno de los objetivos principales de Mongo es proporcionar un manejo seguro y rápido de conjuntos de datos muy grandes.

El método más claro para lograr esto es a través de la fragmentación horizontal por rangos de valor.

- En lugar de un único servidor que aloja todos los valores de una colección, algún rango de valores se divide o *fragmenta* en otros servidores.
- Por ejemplo, en la colección de números telefónicos, podemos poner todos los números de teléfono menores que 1-500-000-0000 en el servidor A y los mayores o iguales a 1-500-000-0001 en el servidor B.
- Mongo hace esto más fácil mediante el *autosharding*, que puede ser configurado.

Esquema de fragmentación



En el esquema anterior participan tres componentes principales:

Fragmentos: los fragmentos se utilizan para almacenar datos.

- Proporcionan alta disponibilidad y consistencia de datos.
- En el entorno de producción, cada fragmento es un conjunto de réplicas independiente.

Servidores de configuración: los servidores de configuración almacenan los metadatos del clúster. Estos datos contienen una asignación del conjunto de datos del clúster a los fragmentos.

- El enrutador de consultas utiliza estos metadatos para dirigir las operaciones a fragmentos específicos.
- En el entorno de producción, los clústeres fragmentados tienen exactamente 3 servidores de configuración.

Enrutadores de consulta: los enrutadores de consulta son básicamente instancias Mongo, la interfaz con aplicaciones cliente y las operaciones directas al fragmento apropiado.

- El enrutador de consultas procesa y dirige las operaciones a particiones y, a continuación, devuelve los resultados a los clientes.
- Un clúster fragmentado puede contener más de un enrutador de consultas para dividir la carga de solicitudes del cliente.
- Un cliente envía solicitudes a un enrutador de consultas. Generalmente, un clúster fragmentado tiene muchos enrutadores de consulta.

Configuración de fragmentación

- Se necesita un servidor para realizar un seguimiento de las claves para saber las ciudades que van al servidor frag1, frag2 o frag3.
- En Mongo, se crea un *servidor de configuración* (que es un servicio *mongod* normal) que realiza un seguimiento de qué servidor posee qué valores.
- Se deberá crear e inicializar un conjunto de réplicas para la configuración del clúster.
- `c:\... \mongod --configsvr --replSet empresa --dbpath c:\data\config --port 27020`
- Ahora, para el servidor de configuración *localhost:27020* se inicia el clúster de servidores de configuración (con solo un miembro para este ejemplo, pueden ser varios):
- `mongo --host localhost --port 27020`
- ```
> rs.initiate({
 _id: "empresa",
 configsvr: true,
 members: [{_id: 0, host: "localhost:27020"}]})
{ "ok" : 1 }
> rs.status().ok
1
```

- Para la creación de las réplicas, se debe cambiar el parámetro ***shardsvr*** necesario para ser un servidor de fragmentos (solo es capaz de fragmentar).

- ```
c:\... \mongod --shardsvr --replSet empresa --  
dbpath c:\data\frag1 --port 27011  
c:\... \mongod --shardsvr --replSet empresa --  
dbpath c:\data\frag2 --port 27012
```

- ```
c:\... \mongod --shardsvr --replSet empresa --
dbpath c:\data\frag3 --port 27013
```

- Estos nodos servirán para los fragmentos.
- Finalmente, se debe ejecutar otro servidor como único punto de entrada para los clientes. Este servidor se conectará al servidor de configuración para realizar un seguimiento de la información de fragmentación almacenada allí.

```
$ mongos --configdb empresa/localhost:27020 --port 27030
```

- *mongos* es un clon ligero de un servidor *mongod*, lo que lo convierte en el intermediario perfecto para que los clientes se conecten a múltiples servidores fragmentados.

- Para establecer los criterios de fragmentación, se debe establecer una conexión en la terminal del servidor de configuración, en la base de datos de administración:

```
mongo --host localhost --port 27030
```

- Y creando los fragmentos:

```
> sh.addShard('localhost:27011')
{ "shardAdded" : "shard0000", "ok" : 1 }
> sh.addShard('localhost:27012')
{ "shardAdded" : "shard0001", "ok" : 1 }
```

- Para establecer el atributo de fragmentación:

```
> db.runCommand({ enablesharding : "test" })
{ "ok" : 1 }
> db.runCommand({ shardcollection : "test.cities", key :
{name : 1} })
{ "collectionsharded" : "test.cities", "ok" : 1 }
```

# Consideraciones

En la configuración de MongoDB, un problema es decidir quién es promovido cuando un nodo maestro se cae.

- MongoDB se ocupa de esto dando un voto a cada servicio ***mongod***, y el que tiene los datos más recientes es elegido el nuevo maestro.

Cuando los nodos vuelven a aparecer, entran en un estado de recuperación e intentan resincronizar sus datos con el nuevo nodo maestro.

- ¿qué pasaría si el maestro original tuviera datos que aún no se propagan? Esas operaciones se abandonan.
- Una escritura en un conjunto de réplicas de Mongo no se considera correcta hasta que la mayoría de los nodos tienen una copia de los datos.



MongoDB espera un número impar de nodos totales en el conjunto de réplicas. Si los problemas de conexión dividen en dos fragmentos, el fragmento más grande tiene una mayoría y puede elegir un maestro y continuar atendiendo las solicitudes. Sin una mayoría clara, no se pudo alcanzar el quórum.

Que podría suceder con un conjunto de réplicas de cuatro nodos. Un conjunto tendrá el maestro original, pero debido a que no puede ver una *clara mayoría* de la red, el maestro se retira. El otro conjunto tampoco podrá elegir un maestro porque tampoco puede comunicarse con una clara mayoría de nodos. Ambos conjuntos ahora no pueden procesar solicitudes y el sistema está efectivamente caído.

Algunas bases de datos (como CouchDB) están diseñadas para permitir múltiples maestros, pero Mongo no lo está, por lo que no está preparado para resolver las actualizaciones de datos entre ellos. MongoDB se ocupa de los conflictos entre múltiples maestros simplemente no permitiéndolos.

Debido a que es un sistema CP, Mongo siempre conoce el valor más reciente; el cliente no necesita decidir. La preocupación de Mongo es una fuerte consistencia en las escrituras, y evitar un escenario multimaestro no es un mal método para lograrlo.

# GridFS

- Mongo incorpora un sistema de archivos distribuido llamado GridFS.
- Mongo viene incluido con una herramienta de línea de comandos para interactuar con GridFS llamada `mongofiles`.
- Ejemplo de listado de archivos:

```
$ mongofiles -h localhost:27020 list
```

- Para subir cualquier archivo:

```
$ mongofiles -h localhost:27020 put archivo.txt
```

```
$ mongofiles -h localhost:27020 list
```

```
2017-05-11T20:04:39.019-0700 connected to:
localhost:27020
```

```
archivo.txt 2032
```

GridFS divide un archivo en trozos y almacena cada fragmento de datos en un documento separado, cada uno de tamaño máximo 255k.

De forma predeterminada, GridFS utiliza dos colecciones **fs.files** y **fs.chunks** para almacenar los metadatos del archivo y los fragmentos.

Cada fragmento se identifica por su campo objectId único **\_id**. El archivo **fs.files** sirve como documento primario.

El campo **files\_id** del documento **fs.chunks** vincula el fragmento a su elemento primario.

# Comandos útiles en MongoDB

- **mongodump** Exporta datos de Mongo a archivos .bson.
- **mongofiles** Manipula archivos de datos GridFS grandes (GridFS es una especificación para archivos BSON superiores a 16 MB).
- **mongooplog** Sensa registros de bitácora de las operaciones de replicación de MongoDB.
- **mongorestore** Restaura las bases de datos y colecciones de MongoDB a partir de copias de seguridad creadas con `mongodump`.
- **mongostat** Muestra las estadísticas básicas del servidor MongoDB.
- **mongoexport** Exporta datos de Mongo a archivos CSV (valor separado por comas) y JSON.
- **mongoimport** Importa datos en Mongo desde archivos JSON, CSV o TSV (valores separados por términos).
- **mongoperf** Realiza pruebas de rendimiento definidas por el usuario en un servidor MongoDB.
- **mongos** Abreviatura de "MongoDB shard", esta herramienta proporciona un servicio para enrutar correctamente los datos a un clúster de MongoDB fragmentado
- **mongotop** Muestra los datos de uso de cada colección almacenada en una base de datos.
- **bsondump** Convierte archivos BSON a otros formatos, como JSON.

# Fortalezas de Mongo

La principal fortaleza de MongoDB radica en su capacidad para manejar grandes cantidades de datos (y grandes cantidades de solicitudes) mediante replicación y escalado horizontal.

- Pero también tiene un beneficio adicional de un modelo de datos muy flexible. Nunca es necesario ajustarse a un esquema y puede implicar anidar cualquier valor que generalmente se uniría usando SQL en un RDBMS.

Finalmente, MongoDB fue construido para ser fácil de usar.

- Esto no es por accidente y es una de las razones por las que Mongo tiene tanta participación mental entre las personas que han desertado del campo de la base de datos relacionales.

# Debilidades de Mongo

Mongo fomenta la desnormalización de los esquemas (al no tener ninguno) y eso puede ser demasiado para algunos.

- Puede ser peligroso insertar cualquier valor de cualquier tipo en una colección. Un solo error tipográfico puede causar horas de dolor de cabeza si no piensa en mirar los nombres de campo y los nombres de colección como un posible culpable. La flexibilidad de Mongo generalmente no es importante si su modelo de datos ya está bastante maduro.

Debido a que Mongo se centra en grandes conjuntos de datos, funciona mejor en clústeres grandes, lo que puede requerir cierto esfuerzo para diseñar y administrar.

- A diferencia de algunas bases de datos cluster donde agregar nuevos nodos es un proceso transparente y relativamente indoloro, la configuración de un clúster de Mongo requiere un poco más de previsión.

# Referencias

---

- A veces es útil que los documentos se referencien entre sí. Para ello se utiliza una construcción `{ $ref : "collection_name", $id : "reference_id" }`.
  - Por ejemplo, para actualizar la colección de ciudades para que contenga una referencia a un documento en los países.

```
> db.towns.update(
 { _id : ObjectId("59094292afbc9350ada6b808") },
 { $set : { country: { $ref: "countries", $id: "us" } } }
)
```

Ahora la variable `portland` podrá contener el resultado de la búsqueda empleando referencia:

```
> var portland = db.towns.findOne(
 { _id : ObjectId("59094292afbc9350ada6b808") }
)
```

y ser empleada para obtener el resultado

```
> db.countries.findOne({ _id: portland.country.$id })
```