

# MongoDB

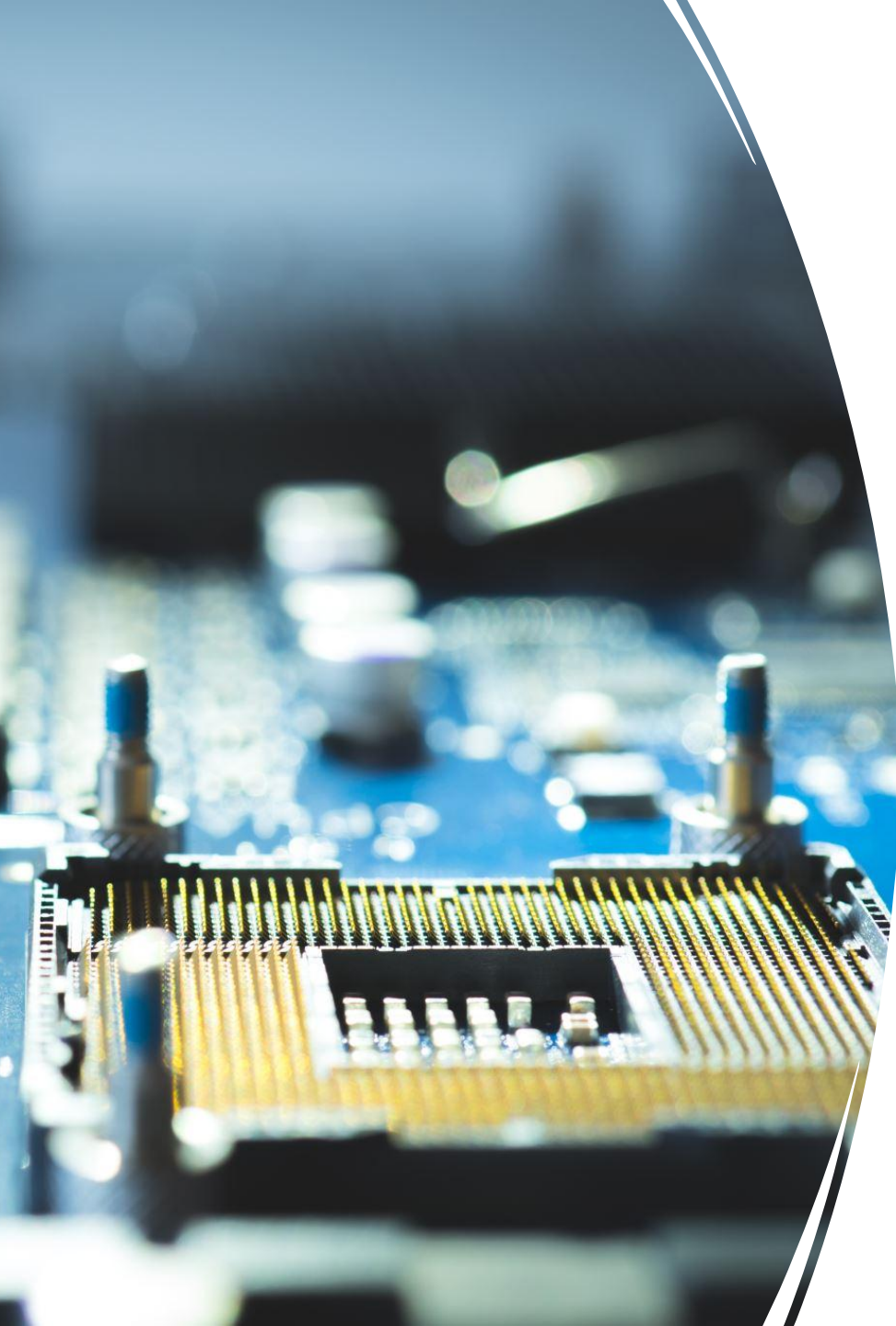
---



# 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.

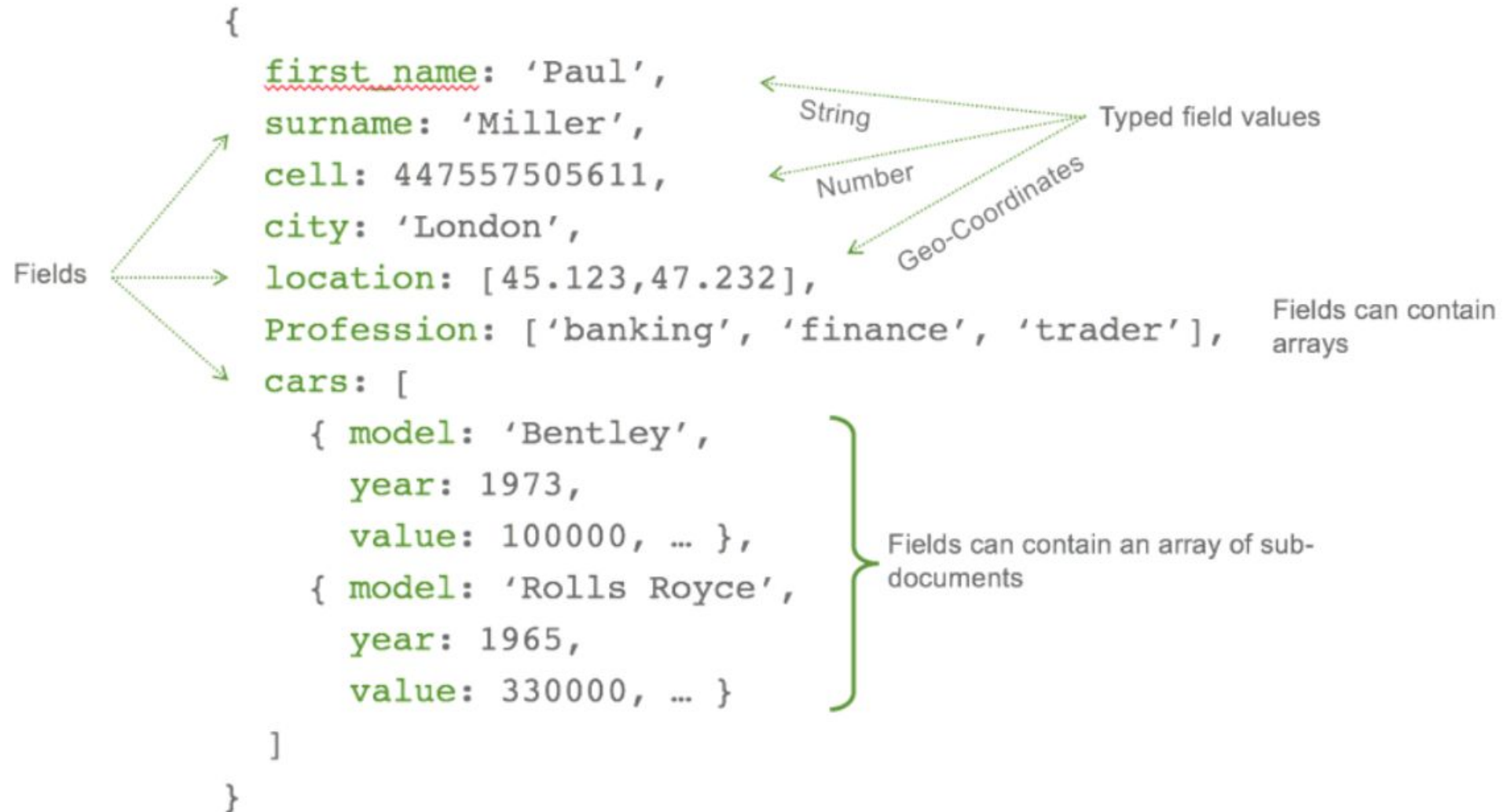
# Formato de almacenamiento

- 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
  - un arreglo
  - un booleano
  - nulo
- En JSON, *los valores de cadena* deben escribirse entre comillas dobles:  
`{ "nombre" : "Juan" }`



Cadenas	Las cadenas en JSON deben escribirse entre comillas dobles. <pre>{ "nombre": "Juan" }</pre>
Números	Los números en JSON deben ser un entero o un coma flotante, sin comillas. <pre>{ "edad": 30 }</pre>
Objetos	Los valores en JSON pueden ser objetos. <pre>{ "empleado": { "nombre": "Juan", "edad": 30, "ciudad": "CDMX" } }</pre>
Arreglos	Los valores en JSON pueden ser arreglos. <pre>{ "empleados": [ "Juan", "Ana", "Pedro" ] }</pre>
Booleanos	Los valores lógicos pueden ser true/false. <pre>{ "venta": true }</pre>
Nulos	Los valores de atributos 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": "Juan",    "apellido": "López"  }, {  
  "nombre": "Ana",    "apellido": "Ramírez"  }, {  
  "nombre": "Pedro",  "apellido": "Jiménez"  } ] }
```

- Ejemplo XML

```
<empleados>  
  <empleado>  
    <nombre>Juan</nombre> <apellido>López</apellido>  
  </empleado>  
  <empleado>  
    <nombre>Ana</nombre> <apellido>Ramírez</apellido>  
  </empleado>  
  <empleado>  
    <nombre>Pedro</nombre> <apellido>Jiménez</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 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.

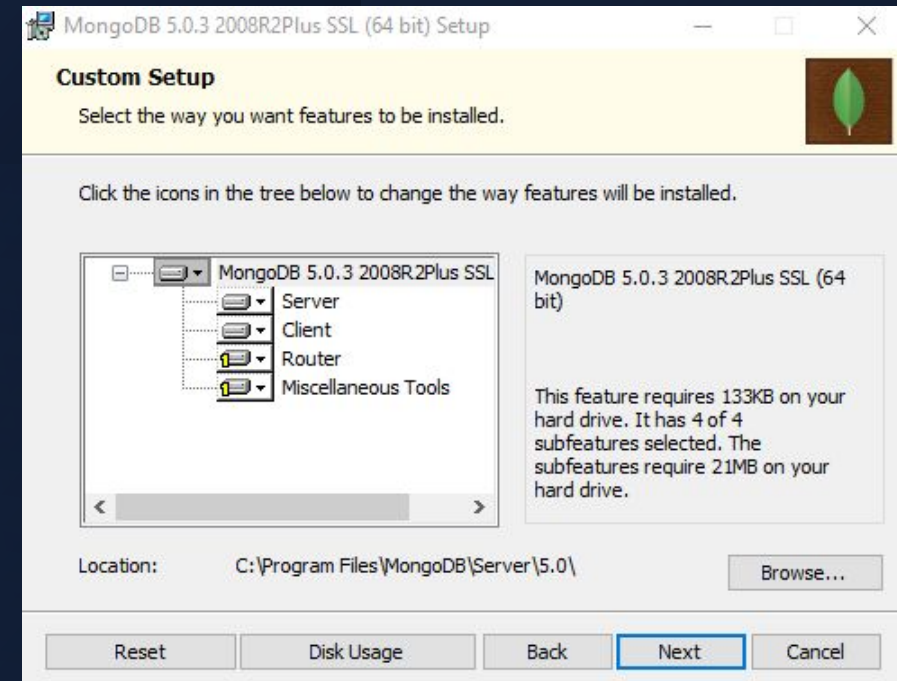
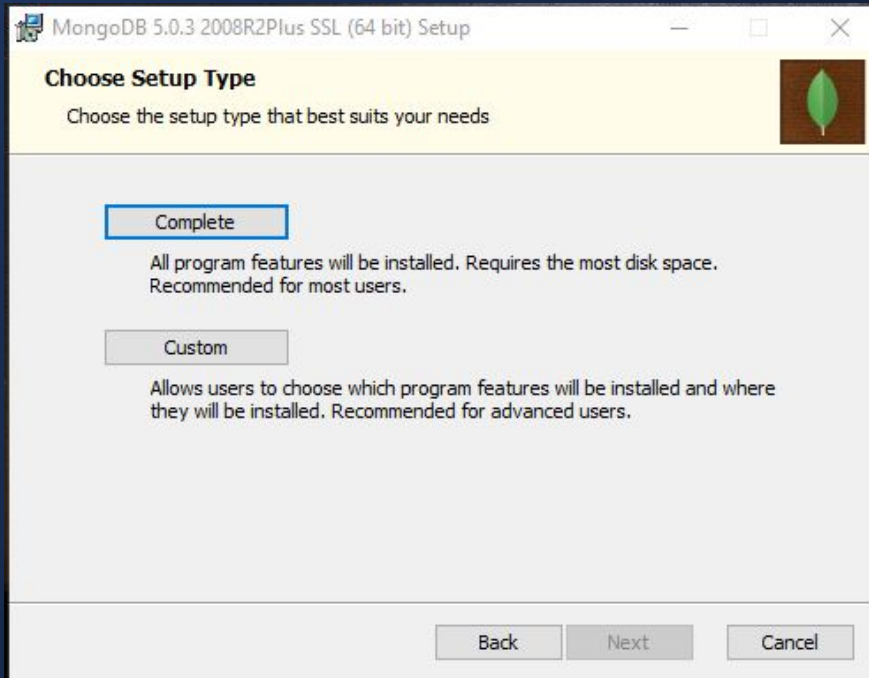
- **Documento**

- 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 atributos o estructura, y los atributos 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		Atributo	
Reunión de tablas		Documentos incrustados	
Clave principal		Clave principal (clave predeterminada _id proporcionada por Mongo)	

# 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.

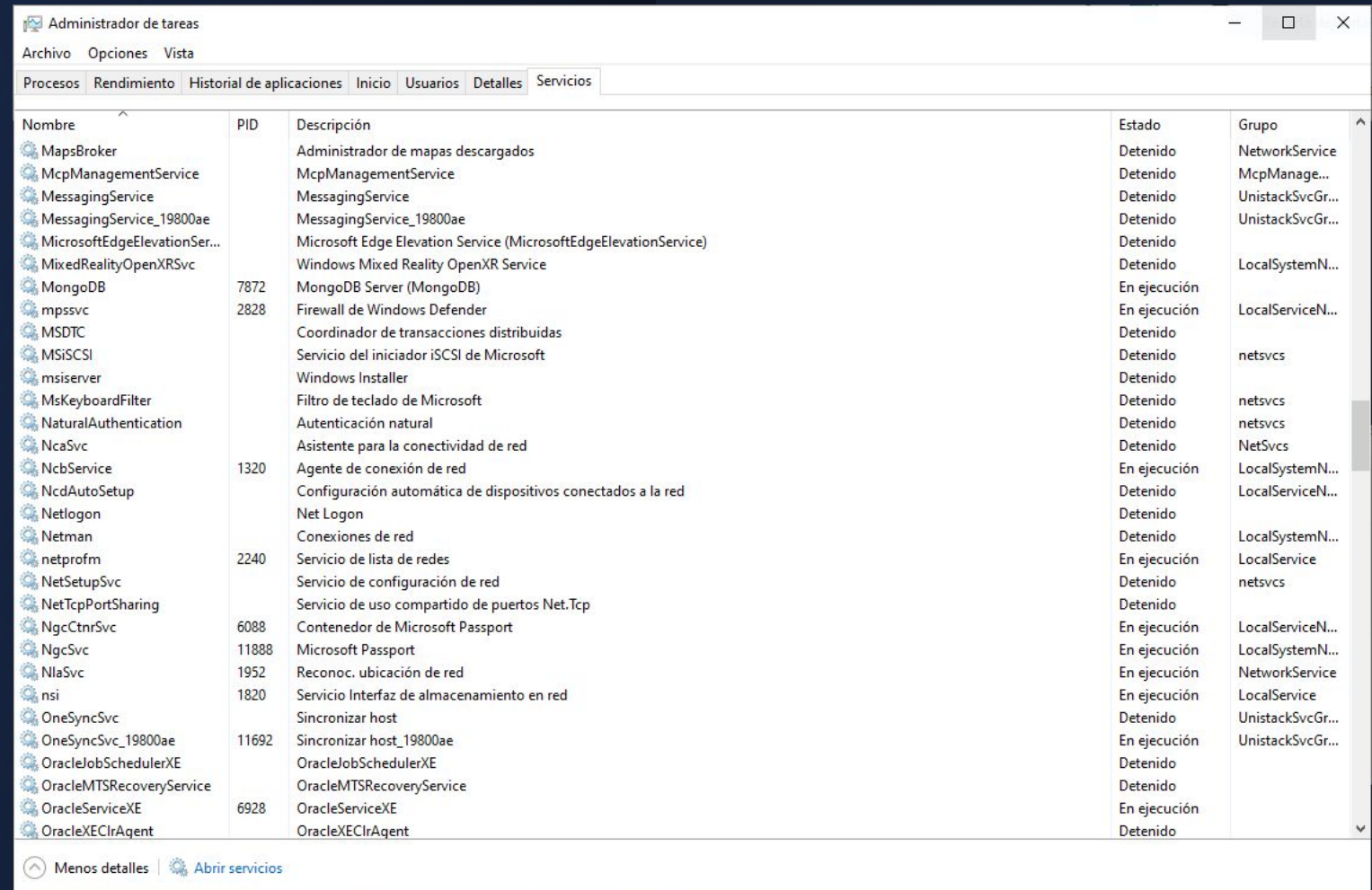
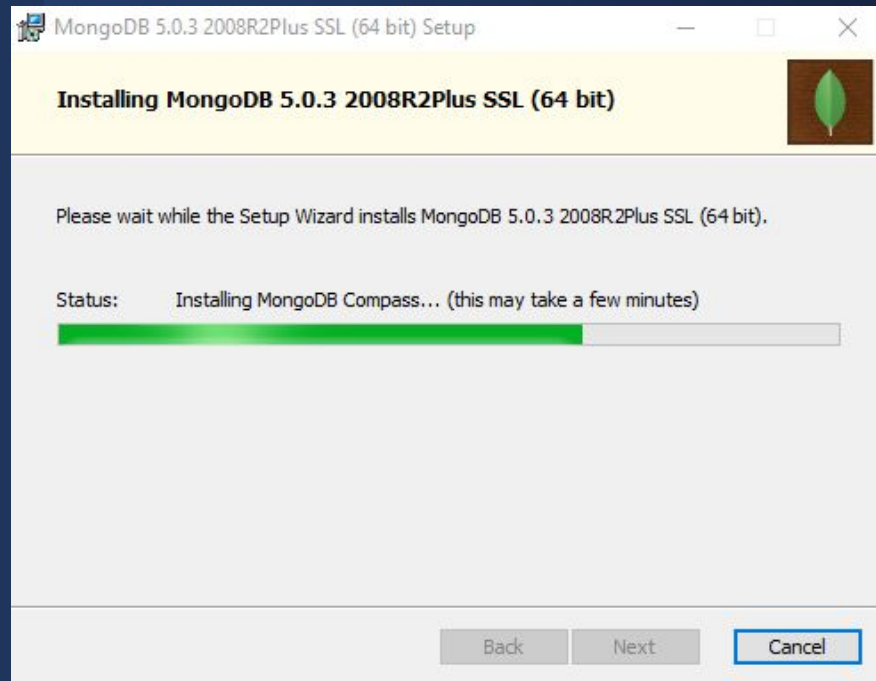
By checking below this installer will automatically download and install the latest version of MongoDB Compass on this machine. You can learn more about MongoDB Compass here: <https://www.mongodb.com/products/compass>

☒ Install MongoDB Compass

Back   Next   Cancel



# Finalización y ejecución





# 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]\bin\mongod.exe
```

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

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

```
MongoDB shell version v5.0.8
```

```
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

```
Implicit session: session { "id" : UUID("cf4e7331-61cc-4e01-9ee1-2f292c52cf97") }
```

```
MongoDB server version: 5.0.8
```

```
=====
```

- 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

- El comando `show dbs` muestra todas las bases de datos del sistema. El comando `use` permite utilizar (cambiarse a ) otra base de datos. Se puede verificar la base de datos actual con el comando `db`.
- Crear una colección en Mongo es posible con solo añadir un registro inicial a dicha colección. Debido a que Mongo no maneja esquemas, no es necesario definir algo previamente. Además, la base de datos *empresa* no existe realmente hasta que se agreguen valores en ella, mediante la colección llamada `empleados` con la función `insertOne()`

```
>db.empleados.insertOne({
nombre: "Juan",
apellidos: "López Pérez",
fecha nacimiento: ISODate("2000-07-01"),
genero: "M",
domicilio : {
  calle : "Juan Escutia #26",
  colonia : "Independencia",
  ciudad: "Cuernavaca",
  CP: 34672
},
idiomas: [ "español", "inglés", "francés" ]
})
{ acknowledged: true,
  insertedId: ObjectId("634df5f3c0c93b307a64da70") }
```

# Inserción de documentos

- La función `insertOne()` recibe como parámetro un documento JSON con la estructura de datos definida por el usuario. Debe de estar conforme a la definición de un documento JSON para que pueda ser insertado, aunque no es necesario (a menos que se defina un validador) que el contenido sea consistente en cada documento insertado.
- La función `insertMany()` permite la inserción de varios documentos en una sola operación. En este caso se debe vigilar que el parámetro sea un arreglo de documentos:

```
>db.empleados.insertMany([{"nombre": "Ana",apellidos: "Pérez Gallegos",fecha_nacimiento:
ISODate("2001-09-20"), genero: "F",domicilio : {calle : "Norte 122 #98-C", colonia : "Agrícola", ciudad:
"Juchitán", CP: 25673}, idiomas: [ "español" ]} , {"nombre: "José Juan",apellidos: "Monr
Escobedo",fecha nacimiento: ISODate("1998-10-10"), genero: "M",domicilio : {calle : "Zaragoza #101", colonia
: "Nueva Industrial", ciudad: "Coatzacoalcos", CP: 45326}, idiomas: [ "español", "italiano" ]}, {"nombr
"Paula",apellidos: "Martínez López",fecha nacimiento: ISODate("2000-11-21"), genero: "F",domicilio : {cal
: "República Dominicana #268", colonia : "Mundial", ciudad: "Torreón", CP: 34672}, idiomas: [null]} ])
{ acknowledged: true,
  insertedIds:
    { '0': ObjectId("634e13cac0c93b307a64da72"),
      '1': ObjectId("634e13cac0c93b307a64da73"),
      '2': ObjectId("634e13cac0c93b307a64da74") } }
```

# Consultas

- Para listar el contenido de una colección, se usa la función `find()`.

```
> db.empleados.find()
{ _id: ObjectId("634e1321c0c93b307a64da71"),
  nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ] }
{ _id: ObjectId("634e13cac0c93b307a64da72"),
  nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ] }
{ _id: ObjectId("634e13cac0c93b307a64da73"),
  nombre: 'José Juan',
  apellidos: 'Monroy Escobedo',
  fecha_nacimiento: 1998-10-10T00:00:00.000Z,
  genero: 'M',
  domicilio:
    { calle: 'Zaragoza #101',
      colonia: 'Nueva Industrial',
      ciudad: 'Coatzacoalcos',
      CP: 45326 },
  idiomas: [ 'español', 'italiano' ] }
{ _id: ObjectId("634e13cac0c93b307a64da74"),
  nombre: 'Paula',
  apellidos: 'Martínez López',
  fecha_nacimiento: 2000-11-21T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'República Dominicana #268',
      colonia: 'Mundial',
      ciudad: 'Torreón',
      CP: 34672 },
  idiomas: [ null ] }
```

# Creación de colecciones

- El comando `show collections` muestra las colecciones creadas hasta el momento en el sistema.

```
> show collections
empleados
```

- 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("prueba")
{ ok: 1 }
> show collections
prueba
```

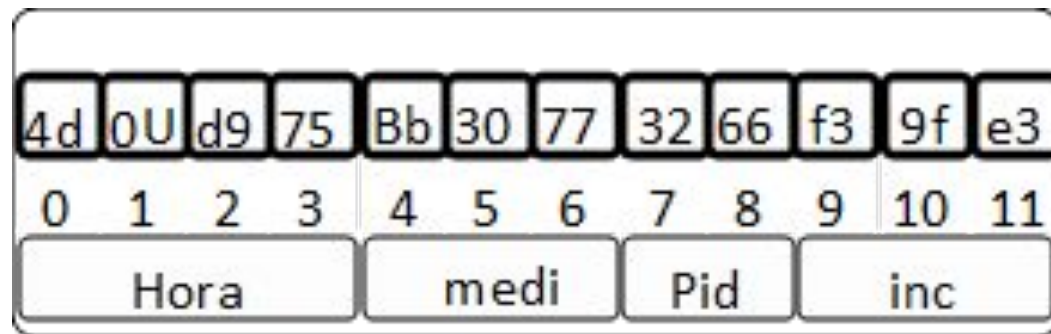
- Es preferible usar `createCollections()` cuando se desea establecer los parámetros de configuración de la base de datos, aunque por lo regular se crea automáticamente cuando se inserta el primer documento.

```
> db.createCollection("demo",
{capped:true, autoIndexId:true,
size:6142800, max:10000})
{ ok: 1 }
> show collections
demo
prueba
```

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

# 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.empleados
'object'
> typeof db.empleados.insertOne
'function'
```

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

```
> inserta_empleado = function(nomb, apell, fn, gen, dom, ids)
{db.empleados.insertOne({nombre:nomb, apellidos:apell,
fecha_nacimiento:ISODate(fn), genero:gen, domicilio:dom, idiomas:ids
})}
```

```
[Function: inserta empleado]
```

La función recién creada se puede invocar con los valores a insertar:

```
> inserta_empleado("María", "López Martínez", "1998-05-28",
{calle:"Buenaventura #10", colonia:"Benito
Juárez", ciudad:"Torreón", CP:45329 }, ["español", "inglés",
"portugués"])
```



- La función `find()` sin parámetros se usa para obtener todos los documentos, sin condiciones. Para acceder a un documento específico y único, se emplea la función `findOne()` y como parámetro, un objeto con la búsqueda de un criterio que garantice un solo documento de retorno. Regularmente se emplea el atributo `_id`. El atributo `_id` se debe convertir a una cadena de texto con la función `ObjectId("_id")`. Si la expresión regresa un conjunto de documentos, solo se muestra el primero.

```
> db.empleados.findOne({_id: ObjectId("634e1321c0c93b307a64da71")})
```

```
{ _id: ObjectId("634e1321c0c93b307a64da71"),  
  nombre: 'Ana',  
  apellidos: 'Pérez Gallegos',  
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,  
  genero: 'F',  
  domicilio:  
    { calle: 'Norte 122 #98-C',  
      colonia: 'Agrícola',  
      ciudad: 'Juchitán',  
      CP: 25673 },  
  idiomas: [ 'español' ] }
```

- El valor del atributo *id* es asignado por el sistema, pero también es posible asignarlo por parte del usuario, como una cadena de texto.

```
> db.empleados.insertOne({
  id:"empleado10",
  nombre: "Guadalupe",
  apellidos: "García Barrera",
  fecha nacimiento: ISODate("1999-01-18"),
  genero: "F",
  domicilio : {
    calle : "calle #129",
    colonia : "Progreso",
    ciudad: "Cuernavaca",
    CP: 34672
  }
})
```

# Búsqueda de un solo registro

---

- De igual manera, la búsqueda de un solo documento puede hacerse con la función *findOne()* y el atributo *\_id* con el valor de texto asignado

```
db.empleados.findOne({ id:"empleado10"})
```

```
{ _id: 'empleado10',  
  nombre: 'Guadalupe',  
  apellidos: 'García Barrera',  
  fecha_nacimiento: 1999-01-18T00:00:00.000Z,  
  genero: 'F',  
  domicilio:  
    { calle: 'calle #129',  
      colonia: 'Progreso',  
      ciudad: 'Cuernavaca',  
      CP: 34672 } }
```

- Las funciones `find()` y `findOne()` también aceptan un segundo parámetro opcional: un objeto *fields* para filtrar los atributos a recuperar.
  - Para obtener solo el nombre (junto con el `_id`), hay que pasar como objeto el atributo deseado con el valor a 1 (o **true**).

```
> db.empleados.find({_id: ObjectId("634e1321c0c93b307a64da71"), {nombre:1}})
{ _id: ObjectId("634e1321c0c93b307a64da71"),
  nombre: 'Ana' }
```

- Para recuperar todos los atributos, excepto *nombre*, hay que establecerlo a 0 (o **false**).

```
> db.empleados.find({_id: ObjectId("634e1321c0c93b307a64da71"), {nombre:0}})
{ _id: ObjectId("634e1321c0c93b307a64da71"),
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ] }
```

- En Mongo, se pueden construir consultas sobre los atributos, rangos, o una combinación de criterios.
- Para encontrar todos los empleados cuyo nombre comience con la letra *P* (mayúscula) y sean de género femenino (caracter 'F'), se puede usar la siguiente expresión:

```
db.empleados.find({nombre:/^P/, genero: 'F'}, { id:0,nombre:1,apellidos:1}  
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

- La expresión */^P/* es una expresión regular. En la siguiente diapositiva se muestran algunos ejemplos de construcción y uso de las expresiones regulares.

# Notación de Regex

A continuación se muestran los símbolos empleados en las expresiones regulares, para escribir consultas de búsqueda de patrones:

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 unarios en Mongo siguen el formato de `{atributo : { $operador : valor }}`, donde `$op` es un operador de comparación, como `$ne` (no igual a) o `$gt` (mayor que). En la siguiente diapositiva se muestran algunos de los operadores empleados en JavaScript.
- Los operadores binarios siguen el formato `{ $operador : [ {atributo : valor}, {atributo : valor} ] }` en donde el operador binario tiene como valor un arreglo de dos objetos como condiciones. Es necesario que sean dos condiciones para que no sea un error.
- Es posible construir el operador de comparación como un objeto de JavaScript, y posteriormente invocarlo dentro de la especificación de la consulta, como en los siguientes criterios:

```
var condicion = {$eq: 'F'}
db.empleados.find({nombre:/^P/, genero:condicion},{ id:0,nombre:1,apellidos:1})
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# 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 atributo
\$all	Hacer coincidir todos los elementos de un arreglo
\$in	Hacer coincidir cualquier elemento de un arreglo
\$nin	No coincide con ningún elemento de un arreglo
\$elemMatch	Coincidencia de todos los atributos en un arreglo de documentos anidados
\$or	o
\$nor	Negación de o
\$size	Coincidencia de arreglo de tamaño dado
\$mod	Operador módulo
\$type	Coincidencia si el atributo es de un tipo de dato dado
\$not	Negación de la comparación



Operación	Sintaxis	Ejemplo	Equivalente
Igualdad	{<key>:<value>}	db.mycol.find({"nombre":"Ana"})	where nombre = 'Ana'
Menos que	{<key>:{\$lt:<value>}}	db.mycol.find({"edad":{\$lt:30}})	where edad < 30
Menos que o igual	{<key>:{\$lte:<value>}}	db.mycol.find({"edad":{\$lte:30}})	where edad <= 30
Mayor que	{<key>:{\$gt:<value>}}	db.mycol.find({"edad":{\$gt:30}})	where edad > 30
Mayor que o igual	{<key>:{\$gte:<value>}}	db.mycol.find({"edad":{\$gte:30}})	where edad >= 30
No es igual	{<key>:{\$ne:<value>}}	db.mycol.find({"edad": {\$ne:30}})	where edad != 30

# Comparativa de operadores RDBMS con MongoDB

- Para consultas con rangos de datos, se utilizan los operadores de comparación, empleando la función *ISODate()*, la cuál hace la conversión de una cadena de texto, con el formato *YYYY-MM-DD*, a un objeto de tipo *Date*, como en el siguiente ejemplo, que obtiene los nombres de los empleados que nacieron posterior al 1 de enero de 2000:

```
db.empleados.find({fecha_nacimiento:{$gt:ISODate('2000-01-01')}},{ id:0,nombre:1,apellidos:1})
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'Juan', apellidos: 'López Pérez' }
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# Operadores de comparación binarios

- Para las expresiones de consulta que usen un operador binario, como los operadores *\$or* y *\$and*, es necesario incluir como valor del operador un arreglo de al menos dos elementos, los cuales serán objetos que tienen las condiciones a aplicar:

```
db.empleados.find({$or:[{fecha_nacimiento:{$gt:ISODate('2000-01-01')}},{genero:'F'}]}),
id:0,nombre:1,apellidos:1}
{ nombre: 'Paula', apellidos: 'Martínez López' }
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'Juan', apellidos: 'López Pérez' }
```

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

```
db.empleados.find({$and:[{fecha_nacimiento:{$gt:ISODate('2000-01-01')}},{genero:'F'}]}),
id:0,nombre:1,apellidos:1}
{ nombre: 'Paula', apellidos: 'Martínez López' }
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
```

- Los operadores *\$in*, *\$nin* y *\$all* pueden obtener los valores coincidentes dentro de un arreglo para una expresión condicional. El siguiente ejemplo regresa los empleados que hablan los idiomas *español* o *inglés*:

```
db.empleados.find({idiomas:{$in:['ingles','español']}}, { id:0,nombre:1,apellidos:1,idiomas:1})
{ nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  idiomas: [ 'español' ] }
{ nombre: 'José Juan',
  apellidos: 'Monroy Escobedo',
  idiomas: [ 'español', 'italiano' ] }
{ nombre: 'Juan',
  apellidos: 'López Pérez',
  idiomas: [ 'español', 'inglés', 'francés' ] }
```

- La búsqueda de atributos que estén contenidos dentro de otros objetos (objetos anidados) se realiza mediante el operador `.` (punto) . Debe tener cuidado de poner los atributos entrecomillados para que no se genere un error. En el ejemplo se obtiene el *nombre* y *apellidos* del empleado que radica en la ciudad de *Cuernavaca*:

```
db.empleados.find({'domicilio.ciudad':'Cuernavaca'}, { id:0, nombre:1, apellidos:1 })
{ nombre: 'Juan', apellidos: 'López Pérez' }
```

- Si se desea saber si existe un atributo en el documento, se usa el operador `$exists`. Por ejemplo, para saber si algún empleado tiene datos de los idiomas que habla:

```
db.empleados.find({'idiomas':{'$exists':true}}, { id:0, nombre:1, apellidos:1, idiomas:1})
{ nombre: 'Juan',
  apellidos: 'López Pérez',
  idiomas: [ 'español', 'inglés', 'francés' ] }
{ nombre: 'Paula',
  apellidos: 'Martínez López',
  idiomas: [ null ] }
{ nombre: 'José Juan',
  apellidos: 'Monroy Escobedo',
  idiomas: [ 'español', 'italiano' ] }
{ nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  idiomas: [ 'español' ] }
```

- Se observa en el resultado que un empleado tiene el valor *null* asignado al atributo *idiomas*, que es diferente que si no existiera ese atributo.

## Salida formateada

- La función `pretty()` se usa para presentar los resultados de la función `find()` con una lectura más cómoda, y se escribe al final de la invocación de `find`. Los resultados son los mismos.

```
db.empleados.find().pretty()  
{  
  id: ObjectId("634e1321c0c93b307a64da71"),  
  nombre: 'Ana',  
  apellidos: 'Pérez Gallegos',  
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,  
  genero: 'F',  
  domicilio:  
    { calle: 'Norte 122 #98-C',  
      colonia: 'Agrícola',  
      ciudad: 'Juchitán',  
      CP: 25673 },  
  idiomas: [ 'español' ] }
```

```
{  
  id: ObjectId("634e13cac0c93b307a64da72"),  
  nombre: 'Ana',  
  apellidos: 'Pérez Gallegos',  
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,  
  genero: 'F',  
  domicilio:  
    { calle: 'Norte 122 #98-C',  
      colonia: 'Agrícola',  
      ciudad: 'Juchitán',  
      CP: 25673 },  
  idiomas: [ 'español' ] }
```

## Limitar registros

- Para mostrar solo los primeros documentos, se debe utilizar la función **limit()**. Acepta un argumento numérico, que es el número de documentos que desea mostrar.

```
db.empleados.find({}, { id:0,nombre:1,apellidos:1}).limit(3)
{ nombre: 'Juan', apellidos: 'López Pérez' }
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }
```

# Saltar documentos

- La función `skip()` se utiliza para omitir (saltar) los documentos indicados por el parámetro numérico, a partir del primer resultado. Se emplea junto con `limit()`

```
db.empleados.find({}, { id:0,nombre:1,apellidos:1}).limit(3).skip(1)
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }
{ nombre: 'Paula', apellidos: 'Martínez López' }
```



# Ordenamiento de registros

- La función **sort()** realiza un ordenamiento del resultado obtenido por la función **find()**. Acepta un documento que contiene una lista de atributos junto con su orden de clasificación.
- Para especificar el orden de clasificación se utilizan los valores enteros 1 (ascendente) y -1 (descendente).

```
db.empleados.find({}, {nombre:1, apellidos:1, id:0}).sort({nombre:1, apellidos:-1})
```

```
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }  
{ nombre: 'Guadalupe', apellidos: 'García Barrera' }  
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }  
{ nombre: 'Juan', apellidos: 'López Pérez' }  
{ nombre: 'María', apellidos: 'López Martínez' }  
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# countDocuments() y distinct()

- Para obtener el total (conteo) de los documentos en una colección, se emplea la función `countDocuments()` sin parámetros :

```
db.empleados.countDocuments()
```

```
4
```

- Se puede pasar como parámetro una condición que será evaluada y regresará el conteo de documentos que cumplieron la condición:

```
db.empleados.countDocuments({genero:'F'})
```

```
4
```

- El método `distinct()` devuelve cada valor diferente contenido dentro de un atributo. Se tiene que pasar como primer parámetro obligatorio, el nombre del atributo, como texto, y opcionalmente como segundo parámetro un objeto como condición:

```
db.empleados.distinct('genero')
```

```
[ 'F', 'M' ]
```

```
db.empleados.distinct('nombre',{genero:'F'})
```

```
[ 'Ana', 'Martha', 'Paula' ]
```

# Actualización de un solo documento

---

- La función `updateOne(condicion, operacion)` requiere dos parámetros.
  - El primero es una consulta basada en una condición para obtener el documento a actualizar. Se debe establecer una condición de búsqueda de un documento que garantice el retorno de solo uno.
  - El segundo es un valor u objeto que actualizará o reemplazará al atributo(s) indicado(s). Se usa el operador `$set` para establecer el conjunto de atributos con su correspondiente valor.

```
db.empleados.updateOne({ id:ObjectId("634e29f1c0c93b307a64da77") }, {$set:{idiomas:['español']}})
{ acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0 }
```

```
db.empleados.findOne({ id:ObjectId("634e29f1c0c93b307a64da77") }, {idiomas:1})
{ id: ObjectId("634e29f1c0c93b307a64da77"),
  idiomas: [ 'español' ] }
```

# Actualización de varios documentos

---

- La función `updateMany(condicion, operacion)` funciona de forma similar a la función `updateOne()`, con la diferencia de que el criterio de búsqueda puede regresar más de un documento.

```
db.empleados.updateMany({fecha_nacimiento:{$gt:ISODate("2000-01-01")}},{$set:{tipo:"reciente"}})
{ acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0 }
```

```
db.empleados.find({tipo:"reciente"},{nombre:1,tipo:1})
{ id: ObjectId("634df5f3c0c93b307a64da70"),
  nombre: 'Juan',
  tipo: 'reciente' }
{ id: ObjectId("634e29f1c0c93b307a64da75"),
  nombre: 'Ana',
  tipo: 'reciente' }
{ id: ObjectId("634e29f1c0c93b307a64da77"),
  nombre: 'Paula',
  tipo: 'reciente' }
```

## 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

# Eliminación de documentos

- De igual forma que en la actualización, la eliminación de documentos presenta dos funciones: `deleteOne(criterio)` y `deleteMany(criterio)`. Cuando el `criterio` es verdadero, se eliminará el documento completo.

```
>db.empleados.deleteOne({ id:ObjectId("634e29f1c0c93b307a64da75") })
```

- En caso de que se requiera la eliminación de varios documentos que cumplan el criterio, se debe usar la función `deleteMany(criterio)`.

```
>db.empleados.deleteMany({tipo:"reciente"})
```

# Operaciones en un solo documento

---

- En versiones recientes de MongoDB (4.0+), se agregaron las siguientes funciones:
  - `findOneAndUpdate(condicion, operacion)`
  - `findOneAndDelete(condicion, operacion)`
  - `findOneAndReplace(condicion, operacion)`
- las cuales combinan en una sola función las operaciones de modificación y eliminación de un solo documento.
- Los parámetros son equivalentes que con el método `updateOne()` y `deleteOne()`

# Eliminación de colecciones

---

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

```
>use basedatos
switched to db basedatos
>show collections
empleados
system.indexes
>db.empleados.drop()
true
```



# Eliminación de la base de datos

- La función `dropDatabase()` permite la eliminación completa de una base de datos. Se debe tener en uso la base de datos a eliminar, en caso contrario eliminará la base de datos **test**.

```
>use basedatos
switched to db basedatos
>db.dropDatabase()
>{ "dropped" : "basedatos", "ok" : 1 }
>show databases
local      0.78125GB
test       0.23012GB
```

# Indexado

- MongoDB debe escanear todos los documentos de una colección para seleccionar aquellos documentos que coincidan con la condición. Este escaneo es altamente ineficiente y requiere que 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 atributo específico o conjunto de atributos, ordenado por el valor del atributo especificado en el índice.
- MongoDB proporciona varias estructuras de datos para el indexado, como los árboles B, así como otras adicionales, como los índices geoespaciales, bidimensionales y esféricos.

# Creación de índices

- Para crear un índice, debe utilizar la función `createIndex(atributos)` incluyendo como parámetro el conjunto de atributos a indexar.

```
>db.empleados.createIndex({nombre:1,apellidos:-1})  
'nombre_1_apellidos_-1'
```

- Se emplea un valor entero 1 para el orden ascendente y -1 para el orden descendente.
- Cada vez que se crea una nueva colección, Mongo crea automáticamente un índice para el atributo `id`. Estos índices se pueden listar mediante la función `getIndexes()`:

```
>db.empleados.getIndexes()  
[  
  { v: 2, key: { id: 1 }, name: 'id ' },  
  {  
    v: 2,  
    key: { nombre: 1, apellidos: -1 },  
    name: 'nombre_1_apellidos_-1'  
  }  
]
```



# 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.

---

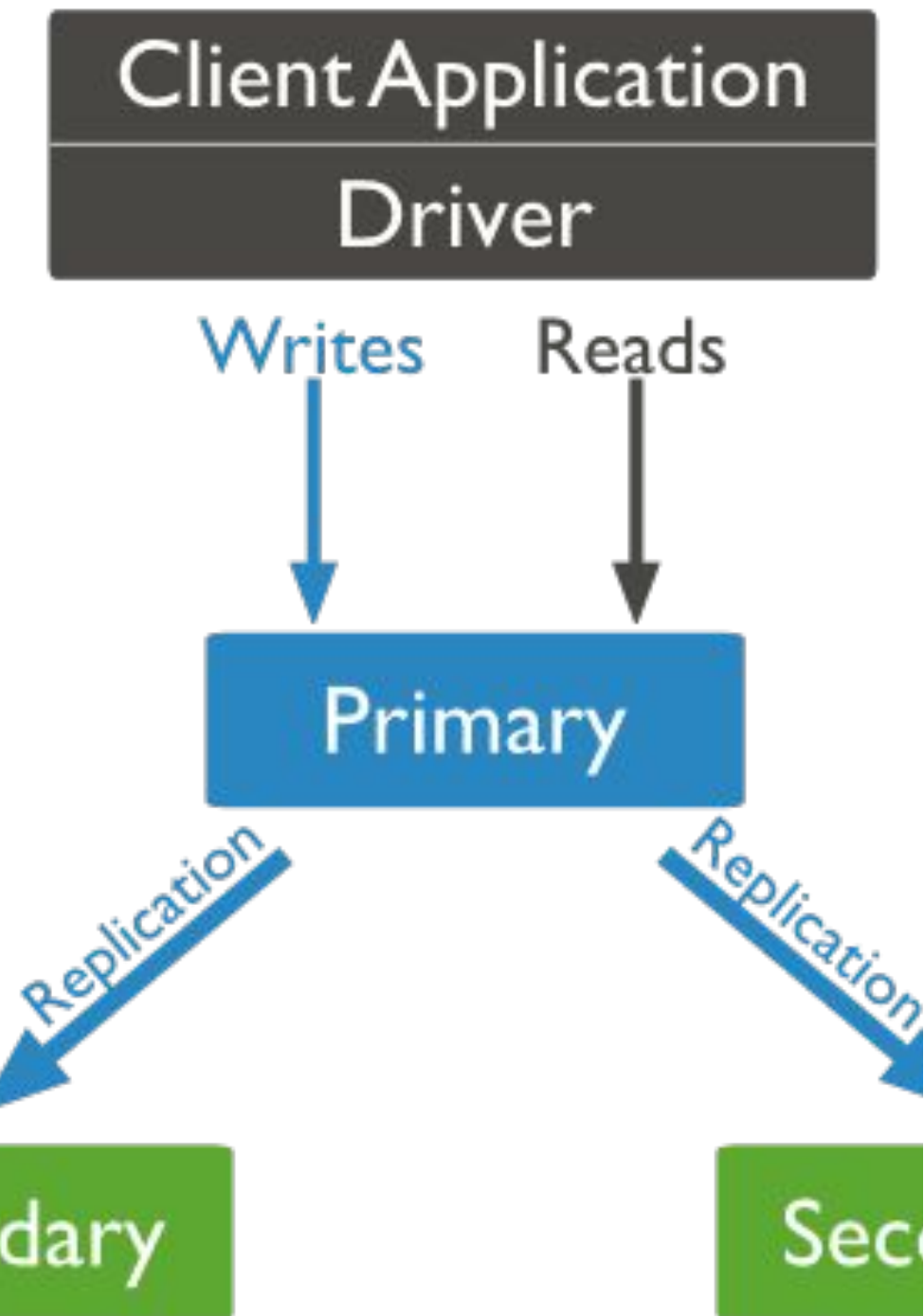
La replicación protege una base de datos de la pérdida de un único servidor.

---

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.



## 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 sólo 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" } })
```

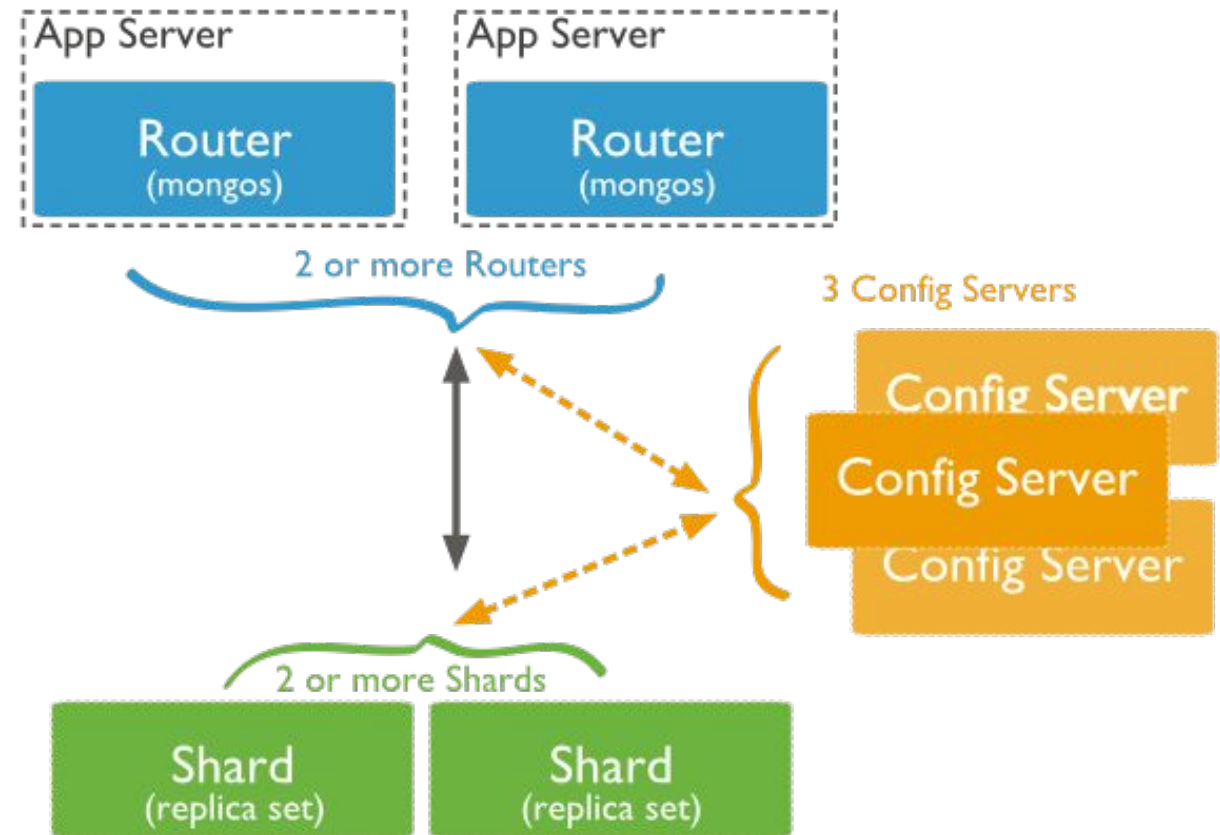
# Fragmentación

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.
- 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 independientes.

**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:\[path]\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 : "empresa" })
{ "ok" : 1 }
> db.runCommand({ shardcollection : "empresa.empleados",
key : {apellidos : 1} })
{ "collectionsharded" : "empresa.empleados", "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 hay problemas de conexión, el fragmento más grande tiene mayoría y puede elegir un maestro y continuar atendiendo las solicitudes. Sin una mayoría clara, no se pudo alcanzar el quórum.

Con un conjunto de réplicas de cuatro nodos, se tendrá al 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 actualizado; el cliente no necesita decidir. La preocupación de Mongo es una fuerte consistencia en las escrituras, y evitar en lo posible un escenario multimaestro.

# 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 Mongo radica en su capacidad para manejar grandes cantidades de datos (y grandes cantidades de solicitudes) mediante replicación y escalamiento horizontal.

- Pero también tiene el beneficio adicional de un modelo de datos muy flexible. No es necesario ajustarse a un esquema.

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 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. 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.