Diferencias entre las bases de datos relacionales y la base de datos MongoDB:

Bases de datos relacionales	**MongoDB**
Siguen el estándar SQL	No sigue el estándar SQL
Tablas	Colecciones
Filas	Documentos
Columnas	Claves
Siguen un esquema de datos	No siguen un esquema de datos
Siempre se respeta la 1ª Forma Normal	Documentos simples y complejos

W (Escrituras en los respaldos) + R (Lectura del dato en los respaldos) = Replicas

Propiedades de las bases de datos relacionales con respecto a la consistencia:

En las bases de datos relacionales se cumplen las propiedades ACID.

- Atomicidad – Atomicity: O se ejecutan todas las operaciones que componen una transacción como si fueran una sola o de lo contrario no se ejecuta ninguna.

- Consistencia – Consistency: Los datos de la BDD han de permanecer correctos y completos.

- Aislamiento – Isolation: Una transacción no debe afectar ni verse afectada por otras, por ello hay que establecer cuándo son visibles los cambios realizados para otras transacciones.

- Persistencia – Durability: Una vez acaba una transacción los datos han de sobrevivir, no se podrán perder aunque falle el sistema.

Propiedades de las bases de datos NoSQL con respecto a la consistencia:

En las bases de datos NoSQL se cumplen las propiedades BASE.

- Básicamente Disponible – Basically Avalible: Siempre se garantizará una respuesta a una solicitud aunque los datos almacenados estén obsoletos.

- Estado Flexible – Soft State: Cuando se producen cambios en un nodo del clúster que posee réplicas, los cambios en el resto de los nodos que hacen de réplica no tienen lugar de forma síncrona, si no de forma asíncrona.

- Eventualmente Consistente – Eventually Consistent: Debido a la naturaleza distribuida de los nodos, puede ocurrir que en un momento determinado la información no sea consistente, pero con el paso del tiempo, se volverá consistente.

El teorema CAP:

Propiedades a considerar en las bases de datos distribuidas:

- Consistencia - Consistency: Todas las operaciones de lectura obtendrán como respuesta la escritura más reciente o un error.
    - ¿ Cuándo se escoge esta propiedad ?
        - La información consistente lo es todo.
        - El sistema de información no puede permitirse errores.
- Disponibilidad - Availability: Cualquier petición de un cliente ha de acabar siempre en una respuesta no errónea. No hay garantía de obtener la información de la escritura más reciente.
    - ¿ Cuándo se escoge esta propiedad ?
        - Existen datos que van a ser utilizados con una alta frecuencia.
        - No importa que la información obtenida pueda ser errónea de vez en cuando ya que esta tenderá a actualizarse de continuo para ser lo más fiel posible. Esto se conoce con el nombre de consistencia eventual.
- Tolerancia al particionamiento - Partitioning: El clúster debe funcionar pese a que se produzcan roturas de comunicación entre algunos de sus nodos.
    - ¿ Cuándo se escoge esta propiedad ?
        - Tenemos un sistema de información que depende de varios servidores.
        - Puede que las comunicaciones no siempre sean fiables.

Solo se pueden conseguir dos de estas propiedades a la vez y para cada una de estas combinaciones siempre hay alguna problemática:

- CA: Si falla un nodo la comunicación con un nodo el clúster se cae.
    - Ejemplos reales en los que se escoge esta combinación:
        - SGBDR centralizado → Oracle, MySQL, Postgres, etc.
- CP: No se puede asegurar siempre el acceso a la información.
    - Ejemplos reales en los que se escoge esta combinación:
        - Facebook Messenger, MongoDB, Redis (default), Hbase, sistemas bancarios, etc.
- AP: Tal vez obtengamos datos antiguos o no actualizados.
    - Ejemplos reales en los que se escoge esta combinación:
        - Netflix, Spotify, eBay, Cassandra, Redis (clúster), CouchDB, Infinite Graph, etc.

Iniciar el servidor MongoDB:

Desde la consola del sistema: mongod [--dbpath <DataFolder>] [--port <port>]

Iniciar el servidor cliente MongoDB:

Desde la consola del sistema: mongo

Importar BBDD y colecciones:

Desde la consola del sistema:

mongoimport [--drop] --db <dbName> --collection <collName> --file <PathTo *.json>

Exportar BBDD y colecciones:

Desde la consola del sistema:

mongoexport --db <dbName> --collection <collName> --out <PathTo *.json>

Comandos generales de la shell de MongoDB:

cls → Borra la pantalla.

db → Devuelve el nombre de la base de datos actual.

show databases / show dbs → Mostrar todas las bases de datos del servidor.

show collections → Muestra todas las colecciones de datos de la BDD actual.

use <dbName> → Cambia de base de datos, si no existe la crea.

Definir documentos:

[var] <documentName> = { FieldName: Value, FieldName: [v1 , v2 , …] , … }

Imprimir los datos o la estructura del documento:

1. print(document.<FieldName>); → Imprime el valor del campo.
2. printjson(document); → Imprime la estructura del documento.

Comandos básicos:

db.getName() → Devuelve el nombre de la base de datos actual.

db.getCollection("<collectionName>") → Captura una colección especificada.

db.createCollection("name",{capped:<bool>,size:10^5,max:<int>}) → Crear una colección.

db.createView("<ViewName>","<CollectionName>",<pipeline>) → Crear una vista.

db.getCollectionInfos({ name: <collection> }) → Devuelve información de la colección indicada.

db.dropDatabase() → Borra la base de datos actual.

db.getCollectionNames() → Devuelve todas las colecciones.

var emsg = db.getLastError() → Devuelve un string con el último error ocurrido.
if(emsg){ print("ERROR: " + emsg);} → Muestra el mensaje

var error = db.runCommand({ getLastError: 1 }); → Devuelve objeto error.
if(error.err){print("ERROR: "+error.err);} →Muestra el mensaje.

Operaciones CRUPD:

- Inserción:
  - db.<coleccion>.insert(var/document) → Inserción simple.
    - Si se quiere insertar un literal: NumberInt(<value>) o NumberDecimal(<value>)
  - db.<coleccion>.insert([var/document, …]) → Inserción múltiple.
- Consultas:
  - [var cursor =] db.<collection>.[find/findOne] ({ query } , { project })
    - Operadores:
      - De comparación: { Field: { $operator: Value } }
        - $eq , $ne : Equal , Not Equal.
        - $lt , $lte : Less Than , Less Than Or Equal.
        - $gt , $gte : Greater Than , Greater Than Or Equal.
      - De conjuntos:
        - { FieldName: { $all: ArrayOfValues } } → A IncludedOrEqual B
        - { FieldName: { $in: ArrayOfValues } } → $A \cap B \neq \emptyset$
        - { FieldName: { $nin: ArrayOfValues } } → $A \cap B = \emptyset$

- Lógicos:
  - (NO) Field: { $not: { <condition> } }
  - (Y) { $and: [{ Field: { <condition> } } , …] }
  - (O) { $or: [{ Field: { <condition> } } , …] }
  - (NINGUNO) { $nor: [{ Field: { <condition> } } , …] }

- Operaciones sobre arrays:
  - { "ArrayFieldName.IndexOfArray": { $operator: Value} }
    - Permite establecer una condición sobre el elemento del índice.
  - { "ArrayFieldName": <value> }
    - Devuelve el documento si el Value se encuentra en el array.
  - { Field: { $elemMatch: { <condition> , docField: { condition } , … } } }
    - Busca el primer documento contenido en un array en el cual existe al menos un elemento que cumple todas las condiciones.
  - { Field: { $size: <int> } }
    - Encuentra los documentos que tengan un campo array con un número particular de elementos.

- Otras operaciones:
  - { Field: { $type: <"bsonType"|bsonTypesArray> } }
    - Devuelve los documentos cuyo valor del campo sea el del tipo.
  - { $expr: { <expression> } }
    - Permite el uso de expresiones del framework de agregación.
  - { Field: { $exists: <boolean> } }
    - Encuentra documentos que contengan el campo.
  - { $where: "<JS>" } o { $where: function() { <JS> return <boolean>; } }
    - Permite realizar búsquedas mediante código JavaScript, para referirse al documento actual se usará la palabra reservada this. Esto implica el parseo de cada documento, lo que es muy muy lento, por eso su uso no se recomienda.

        - { "ArrayFieldName.$": 1 }
            - Proyecta el primer elemento de un array que cumpla las condiciones que se especifican sobre el en la query.
        - { Field: { $slice: [StartIndex , EndIndex] } }
            - Se usa para proyectar solo algunas posiciones de un array.
        - { Field: { $regex: <Regular expression> } }
            - Buscar un valor de texto mediante expresiones regulares:
                - /a.a/ → Que contenga “a” cualquier letra y “a”.
                - /text/ → Que contenga el texto especificado.
                - /text$/ → Que acabe con el texto especificado.
                - /^text/ → Que comience con el texto especificado.
    - Valores de proyección:
        - true – 1: El campo se proyectará (por defecto).
        - false – 0: El campo no se proyectará.
    - Valores de ordenación:
        - 1: Ascendente (De menor a mayor).
        - -1: Descendente (De mayor a menor).
    - Expresiones regulares de búsqueda (para campos de texto):
        - /a.a/ → Se buscará que contenga “a” cualquier letra y “a”.
        - /text/ → Se buscará que contenga el texto especificado.
        - /text$/ → Se buscará que acabe con el texto especificado.
        - /^text/ → Se buscará que comience con el texto especificado.

NOTAS:

- Separando las condiciones por comas se aplica el operador AND de estas.
    - Ejem: db.<colName>.find({ FieldX: { conditions } , FieldY: { conditions } });
- No puede aparecer 2 veces la misma clave al mismo nivel, si lo hacemos se queda con la última aparición. Para solucionarlo se agrupan las condiciones en el mismo documento.
    - Ejem: db.<colName>.find({ Field: { cond1 , … , condN } });

- Actualización:

  NOTA: Un update total es aquel que actualiza especificando todo el documento, esto puede evitarse usando los operadores de actualización parcial.

  - db.<collection>.save(<document> , { writeConcern: { w: <int> } })

    - Actua como “upsert:true” si se especifica el campo “_id”, en caso contrario se comporta como una instrucción insert. Realiza una operación de update total.

  - db.<collection>.update(
    { query } , { updates } ,
    {
    upsert: true, // CREA EL DOCUMENTO SI NO EXISTE
    multi : true , // ACTUALIZA TODOS DOCUMENTOS
    writeConcern: { w: <int/string> }
    }
)

    - Aislamiento de la actualización: { <query> , $isolated: true }

      - Evita que otro proceso pueda escribir mientras se actualiza.

    - Garantía en la escritura: { writeConcern: { w: <int/string> } }

      - 0: No se verifica si se llevó a cabo la escritura (por defecto).
      - 1: El servidor principal confirmará la escritura.
      - 2: El servidor principal y el primero de respaldo confirmarán.
      - N: El servidor principal y los (N-1) servidores de respaldo confirman.
      - “majority”: La mayoría confirma (principal + todos los respaldos).

    - Operadores para la realización de actualizaciones parciales:

      { $inc: { Field: +-Valor , … } } → Incrementa o decrementa el valor.

      { $set: { Field:Valor , … } } → Cambia o crea el campo.

      { $mul: { Field: NumberDecimal(“0.01”) , … } } → Multiplica o crea.

      { $max: { Field: Valor , … } } → Actualiza al valor si es mayor.

      { $min: { Field: Valor , … } } → Actualiza al valor si es menor.

      { $rename: { OldField : NewField , … } } → Renombra el campo.

      { $unset: { Field: ”” , … } } → Elimina una clave del documento.

      { $currentDate: { Field: true , Field: { $type: “timestamp/date” } } }

      { $setOnInsert: { Field: Value , … } } → Crea cuando upsert inserta.

- Operadores para la realización de actualizaciones parciales sobre Arrays:
  - Para actualizar el primer elemento de un array que cumple la condición que se especifica en la query:
    - { $updateOperator: { “ArrayFieldName.$” : ValueToUpdate } }
  - Inserta elementos por el final de un array, admite repeticiones:
    - { $push: { arrayName: Value }} → Si no existe, crea el array.
      - Value: Un valor atómico o array se insertan como un solo elemento.
      - Value: { $each: Array , $position: <int> } → $position es opcional.
        - Añade los elementos del Array a partir de la posición indicada.
  - Inserta los elementos por el final de un array, no admite repeticiones:
    - { $addToSet: { arrayName: Value } } → Si no existe, crea el array.
      - Value: Un valor atómico o array se insertan como un solo elemento.
      - Value: { $each: Array } → Position es opcional.
        - Añade cada el elemento del array por el final de este.
  - Elimina del array un elemento o un rango de ellos:
    - { $pull: { arrayName: <value|condition> } }
  - Elimina del array todos los elementos del Array:
    - { $pullAll: { arrayName: ArrayOfValues } }
  - Eliminar un elemento por el principio (-1) o por el final (1):
    - { $pop: { arrayName: [1 / -1] } }
  - Si queremos buscar Arrays en los que se encuentren o no determinados datos:
    - { arrayName: { [$in/$nin] : [data,data,…] } }

- Borrado:
    - db.<collection>.remove({condition} , { justOne: <boolean> })
        - Para todo documento que cumpla la condición: Borra el documento y las entradas de los índices que apuntan al documento.
        - La condición puede ser vacía si deseamos borrar todos, aunque para esta operación es más eficiente el uso de ‘drop’.
    - db.<collection>.drop()
        - Borra todos los documentos e índices de la colección.

Operaciones BULK:

- Las operaciones bulk permiten lanzar lotes de instrucciones CRUPD.
- Se suelen utilizar para realizar tareas de administración de la base de datos.
- Funcionamiento:
    - Especificación del tipo de operación bulk:
        - var bulk = db.<collName>.initializeUnorderedBulkOp();
            - Lanza las instrucciones en paralelo, es decir, desordenadas.
        - var bulk = db.<collName>.initializeOrderedBulkOp();
            - Lanza las instrucciones en serie, es decir, hay un orden de ejecución.
        - Las instrucciones se cargan en el lote mediante bulk.insert(), bulk.find(), bulk.update() y bulk.remove(). Sin embargo estas no se ejecutan.
        - Para ejecutar el lote de instrucciones: bulk.execute();

Copias de seguridad:

- Mongo guarda los datos internos y las copias de seguridad bajo el formato BSON, un formato binario más ligero y rápido que el JSON.
- Para restaurar una copia de seguridad, desde la consola del sistema:
    - mongorestore --db <dbName> <PathTo *.bson>
    - mongorestore --db PorNivel --collection <collName> <PathTo *.bson>
- Para crear una copia de seguridad, desde la consola del sistema:
    - mongodump --db <dbName> --out <Path>
    - mongodump --db <dbName> --collection <collName> --out <Path>

Balidador de esquema de Datos:

Se emplean cuando queremos que una colección siga obligatoriamente algún esquema de datos.

```
db.createCollection("<collectionName>", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "FieldName", "FieldName", "FieldName" ],
      properties: {
        FieldName: {
          bsonType: "<int/objectId/decimal/string/bool/long/object/date/double/array/timestamp>",
          description: "descripcion del campo"
        },
        FieldName: {
          bsonType: "<int/objectId/decimal/string/bool/long/object/date/double/array/timestamp>",
          description: "descripcion del campo"
        },
        FieldName: {
          bsonType: "<type>",
          minimum: <number>,
          maximum: <number>,
          description: “descripcion del campo"
        },
        FieldName: {
          enum: [ <value> , "<value>", … ],
          description: "descripcion del campo"
        },
        FieldName: {
          bsonType: [ <type> ],
          minimum: <number>,
          maximum: <number>,
          description: "descripcion del campo"
        },
        "FieldName.FieldName" : {
          bsonType: <type>,
          description: "descripcion del campo"
        },
        "FieldName.FieldName" : {
          bsonType: "string",
          description: "descripcion del campo"
        }
      }
    }
  }
})

db.createCollection("<collectionName>", { validator: {

        $and: [ { FieldName:  { $type: <bsonType>, "$exists": true } }, … ]

} } );
```

Cursores:

- var cursor = QUERY (find/aggregate) → Carga una variable cursor.
  - Operaciones sobre cursores:
    - .count({condition}) → Cuenta los documentos que cumplen la condición.
    - .distinct("fieldName") → Devuelve los distintos valores de un campo.
    - .skip(<int>) → Salta los N primeros documentos.
    - .limit(<int>) → Limita el número de documentos de salida.
    - .pretty() → Formatea la salida mejorando la presentación.
    - .forEach(function(document) {<javaScript>}) → Recorrer documentos.
    - .next() → Devuelve el documento actual y después pasa al próximo.
    - .hasNext() → Indica si quedan documentos por recorrer.
    - .toArray() → Devuelve un array con los documentos.
    - .sort({ Campo1 : desc(-1) /asc(1) , … , … }) → Ordenación de datos.
      - Se desaconseja cuando la cantidad de datos es excesivamente grande.
    - .map(function(document) { <JS> return document; })
      - Aplica la función a todos los documentos y los devuelve en un array.

NOTAS:

- Internamente los comandos se ejecutan en este orden: Sort → Skipt → Limit. Estas operaciones son conmutativas ya que no importa el orden en el que se pongan, pues siempre se ejecutan de esta manera.

El framework de agregación:

db.<collectionName>.aggregate(stage1 , stage2 , … , stageN);

Cada etapa es del tipo: { $stageOperator: { <body> } } → Se basa en el uso de pipes.

Etapas soportadas:

1) $match → Permite establecer los criterios de selección de documentos.

   { $match: { ExistingField: { <operator-expression> } }

2) $group → Agrupa por un/os campo/os y permite usar operaciones de grupo.

   { $group: { _id: <literal/"$FldName">, FieldName: { $operation: <scalar/"$FldName"> } , … } }

   _id: { FieldName: "$ExistingField" , FieldName: "$ExistingField" , … }

3) $project → Proyección de campos (Existentes, calculados, renombrados).

   {$project:{_id:[1/0], FieldName:{$operation: "$FieldName"} , NewName: "$FieldName"}}

4) $out → Inserta los documentos de salida en la colección que se especifique.

   { $out: "CollectionName" } // NO SE PUEDE USAR SI LA COLECCIÓN ES “CAPPED”.

5) $limit → Especifica el máximo número de documentos a devolver.

   { $limit: <int> }

6) $unwind → Crea nuevos documentos iguales, cada uno con un valor del array.

   { $unwind: "$ExistingField" }

   Útil por ejemplo para pasar a 1FN. Delicado si genera la repetición del _id del documento.

7) $sort → Ordena por el campo/s especificado/s.

   { $sort: { ExistingField: <asc(1)/desc(-1)> , ExistingField: <1/-1> } }

8) $skip → Se salta tantos documentos como se especifique y devuelve el resto.

   { $skip: <int> }

9) $set “o” $addFields → Permite crear nuevos campos a partir de valores o cálculos.

   { $addFields: { NewFieldName: <SimpleValue/{$operation:"$ExistingField"}> , … } }

10) $loookup → LEFT OUTER JOIN, no recomendado en entornos BIG DATA.

```
{ $lookup:
        {
                from:"ExternalCollection",
                localField:"LocalFieldName",
                foreignField:"ExternalFieldName",
                as:"ArrayName" // Todos los documentos que coincidan se guardan aquí.
        }
}
```

11) $bucket → Agrupa los valores en función de los intervalos asociados a una clave.

```
{ $bucket:
        {
                groupBy: <expression>, // Campo o fórmula de la que obtenemos los datos.
                boundaries: [ Val1, Val2, … , ValN ], // Define los rangos [Val1,Val2) …
                default: < accumulator / expression / literal > // Si < que Val1 o > que ValN.
                Output: {
                        NewFieldName: { $operator: <SimpleValue/"FieldName"> },
                        …
                }
        }
}
```

12) $bucketAuto → Hay que decirle el número e intervalos que queremos que calcule.

```
{ $bucketAuto:
        {
                groupBy: <expression>, // Campo o fórmula de la que obtenemos los datos.
                buckets: <int> // Número de intervalos que queremos.
                Output: {
                        NewFieldName: { $operator: <SimpleValue/"FieldName"> },
                        …
                }
                [ , granularity: “string” ] // Serie para el control de la distribución de datos.
        }
}
```

13) $facet → La colección de documentos de entrada será procesada por varios pipelines.

```
{ $facet:
        {
                ArrayNameOfPipeline1: [ stage1, stage2, …, stageN ],
                …,
                ArrayNameOfPipelineN: [ stage1, stage2, …, stageN ]
        }
}
```

Funciones de $group en aggregate:

- $push: Crea un vector que puede contener valores repetidos.
  - FieldName: { $push: "$ExistingField" }
  - FieldName: { $push: { FName: "$EField" , FName: "$EField" } }
- $addToSet: Crea un vector sin valores repetidos.
  - FieldName: { $addToSet: "$ExistingField" }
  - FieldName: { $addToSet: { FName: "$EField" , FName: "$EField" } }
- $sum: Acumula o cuenta.
  - FieldName: { $sum: "$ExistingField" }
  - FieldName: { $sum:1 }
- $max: Le asigna el valor más grande a un campo.
  - FieldName: { $max: "$ExistingField" }
  - FieldName: { $max: ["$ExistingField" , "$ExistingField" , …] }
- $min: Le asigna el valor más pequeño a un campo.
  - FieldName: { $min: "$ExistingField" }
  - FieldName: { $min: ["$ExistingField" , "$ExistingField" , …] }
- $avg: Obtiene la media de un campo.
  - FieldName: { $avg: "$ExistingField" }
  - FieldName: { $avg: ["$ExistingField" , "$ExistingField" , …] }
- $first: Devuelve el primer valor de un campo en un grupo.
  - FieldName: { $first: "$ExistingField" }
- $last: Devuelve el último valor de un campo en un grupo.
  - FieldName: { $last: "$ExistingField" }

Operadores aritméticos de $group y $project en aggregate:

- $add: Realiza la suma de un array de números.
  - FieldName: { $add: ["$ExistingField" , "$ExistingField" , …] }
- $subtract: Devuelve la diferencia de dos números.
  - FieldName: { $subtract: ["$ExistingField" , "$ExistingField"] }

- $multiply: Realiza la multiplicación de un array de números.
    - FieldName: { $multiply: ["$ExistingField" , "$ExistingField" , …] }
- $divide: Devuelve la división de dos números.
    - FieldName: { $divide: ["$ExistingField" , "$ExistingField"] }
- $mod: Devuelve el módulo de la división de dos números.
    - FieldName: { $mod: ["$ExistingField" , "$ExistingField"] }
- $cmp: Devuelve 0 si son iguales, 1 si el primero es mayor que el segundo, -1.
    - FieldName: { $cmp: ["$ExistingField" , "$ExistingField"] }

Operadores de comparación de $group y $project en aggregate:

- $eq: Igualdad lógica.
    - { $eq: ["$ExistingField" , "$ExistingField"] }
- $ne: Desigualdad lógica.
    - { $ne: ["$ExistingField" , "$ExistingField"] }
- $lt: Menor estricto.
    - { $lt: ["$ExistingField" , "$ExistingField"] }
- $lte: Menor o igual.
    - { $lte: ["$ExistingField" , "$ExistingField"] }
- $gt: Mayor estricto.
    - { $gt: ["$ExistingField" , "$ExistingField"] }
- $gte: Mayor o igual.
    - { $gte: ["$ExistingField" , "$ExistingField"] }

Operadores booleanos:

- $and: “Y” lógico de un array: { $and: [<expression> , <expression> , …] }
- $or: “O” lógico de un array: { $or: [<expression> , <expression> , …] }
- $not: “NO” lógico: { Field: { $not: [<expression>] } }

Operadores condicionales:

- $cond: Es el equivalente a IF-THEN-ELSE.
  - { $cond: [<boolean-expression> , <true-case> , <false-case>] }
- $ifNull: Devuelve un dato si la expresión es null, si no se proyecta normal.
  - { $ifNull: ["$ExistingField" , <replacement-expression>] }

Operadores de strings:

- $concat: Concatena dos o mas strings.
  - { $concat: [<expression1> , <expression2> , …] }
- $strcasecmp: Devuelve 1 si la primera string es mayor, -1 si menor, 0 iguales.
  - { $strcasecmp: [<expression1> , <expression2>] }
- $substr: Devuelve una subcadena de un string.
  - { $substr: [<string> , <start> , <length>] }
- $strLenCP: Devuelve el tamaño de un string.
  - { $strLenCP: <string> }
- $toLower: Devuelve un string en minúsculas.
  - { $toLower: <expression> }
- $toUpper: Devuelve un string en mayúsculas.
  - { $toUpper: <expression> }
- $trim: Elimina los espacios en blanco.
  - { $trim: { input: <string> [, chars: “<string>”] } }
- $split: Divide un string y forma un array basándose en un delimitador.
  - { $split: [<string> , <string-delimiter>] }

Operadores de arrays:

- $size: Devuelve el tamaño de un array.
  - { $size: "$arrayName" }
- [$in / $nin] : Devuelve un boolean que indica si <value> está o no en el array.
  - { $in: [<value> , [v1 , v2 , …]] }
- $indexOfArray: Devuelve la posición de un elemento, -1 si no está en el array.
  - { $indexOfArray: [“$array” , <value> , <start> , <end> } → Inicio y fin son optativos.
- $arrayElementAt: Devuelve el elemento de la posición indicada.
  - { $arrayElementAt: [“$aray”, <int> }
- $filter: Selecciona un subconjunto de elementos que cumpla la condición.
  - { $filter: { input: “$array” , as: <string> , cond: <condition> } }

Operadores de fechas:

- $dayOfYear: Devuelve el día del año, un número entre 1 y 366.
  - FieldName: { $dayOfYear: "$FieldDate" }
- $dayOfMonth: Devuelve el día del mes, un número entre 1 y 31.
  - FieldName: { $dayOfMonth: "$FieldDate" }
- $dayOfWeek: Devuelve el día del mes, un número entre 1 (Domingo) y 7.
  - FieldName: { $dayOfWeek: "$FieldDate" }
- $year: Devuelve el año actual.
  - FieldName: { $year: "$FieldDate" }
- $month: Devuelve el número de mes, un número entre 1 y 12.
  - FieldName: { $month: "$FieldDate" }
- $week: Devuelve el número de semana, un número entre 0 y 53.
  - FieldName: { $week: "$FieldDate" }
- $hour: Devuelve el número de hora, un número entre 0 y 23.
  - FieldName: { $hour: "$FieldDate" }

- $minute: Devuelve el número de minuto, un número entre 0 y 59.
    - FieldName: { $minute: "$FieldDate" }
- $second: Devuelve el número de segundo, un número entre 0 y 59.
    - FieldName: { $second: "$FieldDate" }
- $millisecond: Devuelve el número de milisegundo, un número entre 0 y 999.
    - FieldName: { $millisecond: "$FieldDate" }
- $dateToString: Devuelve una fecha como string.
    - FieldName: { format: “%d-%m-%Y” , date: “$FieldDate” }
        - Formatos:
            - %Y: Año (4 dígitos).
            - %m: Mes (2 dígitos).
            - %d: Día (2 dígitos).
            - %H: Hora (2 dígitos).
            - %M: Minutos (2 dígitos).
            - %S: Segundos (2 dígitos).
            - %L: Milisegundos (3 dígitos).

NOTAS:

- "$$ROOT" → Es el documento que se está procesando actualmente.
- "$$ROOT.<FieldName>" → Acceso al campo del documento actual.

Map Reduce:

Es un sistema de procesamiento basado en dos etapas que requiere codificar en JavaScript.
Aunque permite realizar cómputos distribuidos actualmente no se recomienda su uso para ello.

- Etapa Map:
    - Entrada: Un documento.
    - Salida: Uno o varias parejas (Clave,Valor).
- Etapa Reduce:
    - Entrada: Una Clave y un array de Valores asociados a dicha clave.
    - Salida: Un valor que es asociado a la Clave de entrada.

```
function mapFunction() { <JavaScript Code> emit(Key,Value); } // Tenemos this. Key→Var o Doc.
function mapFunction() { <JavaScript Code> return <Value>; }
db.<collectionName>.mapReduce(mapFunction, reduceFuntion, {
        query: { <conditions> }, // Filtra los documentos de la etapa Map.
        out: “CollectionName”  // Colección de salida en la que se vuelcan los datos.
});
```

Índices:

Es una estructura de datos que mantiene ordenados ciertos atributos de una colección.

NOTAS:

- Si en Mongo se crea un índice sobre un campo que no existe, el índice se creará igualmente.
- Respecto al uso de índices:
    - Si se consulta por la primera clave del índice este siempre se usará si se sigue el criterio de ordenación de este, en caso contrario, este también podría usarse ya que lo único que habría que hacer es un recorrido a la inversa.
    - Nunca se se usa si no se tiene en cuenta el orden de las claves en el índice.

Desventajas de los índices:

- Hay que guardar el objeto índice.
- Disminuyen la velocidad de las operaciones de inserción, actualización y borrado.

Ventajas de los índices:

- Incrementan la velocidad de las consultas gracias a una búsqueda eficiente de los datos.
- Como los índices se basan en la ordenación de los valores, estos pueden ahorrarnos el coste de la realización de operaciones de ordenación.

Reglas sobre el uso de índices:

- Usar índices en una colección es beneficioso si el número de operaciones de lectura que realizamos sobre esta es mayor que número de escrituras.
- Hemos de intentar crear índices cuya clave o claves:
    - Permitan realizar una igualdad lógica o bien filtrar por un rango de valores.
    - Nos ahorren el tener que realizar operaciones de ordenación.

Tipos de índices:

- Simples:
    - Sintaxis: db.<colName>.createIndex({ Field: <1/-1> }, ”Index Name”);
    - Crea en la colección especificada un índice con nombre sobre un campo especificado.

- Compuestos:
  - Sintaxis: db.<colName>.createIndex({ Field: <1/-1>,Field: <1/-1>, … },”Index Name”);
  - La clave de ordenación la componen múltiples campos en lugar de uno solo.
  - Pueden ser usados en algunas consultas en las que no se usan todas sus claves.
  - No se recomienda usarlos con operaciones de ordenación si no están todas sus claves.
- Sobre subdocumentos:
  - Sintaxis: db.<colName>.createIndex({ “Document”: <1/-1> }, ”Index Name”);
  - Sintaxis: db.<colName>.createIndex({ “Document.Field”: <1/-1> }, ”Index Name”);
  - Es un tipo de índice compuesto cuando se hace sobre un subdocumento entero o varios campos de este. Si sólo se aplica sobre un campo se comporta como un índice simple.
- Sobre Arrays:
  - Sintaxis: db.<colName>.createIndex({ ArrayField: <1/-1> },”Index Name”);
  - Se crea un índice con todos los valores del array como claves separadas. De esta forma cada documento está relacionado con uno o varios valores del índice.
- Unique:
  - Sintaxis: db.<colName>.createIndex({Field: <1/-1>}, { options }, ”Index Name”);
  - Opciones:
    - $unique: true → Indica que los valores de la clave no puede repetirse y que esta tiene que existir en todos los documentos, esto último se debe a los valores nulos.
    - $dropdups: true → Si la colección ya creada y existen documentos que hacen imposible la creación de este índice, esta opción borrará aquellos documentos que no permitan que el índice se pueda crear.
- Textuales:
  - Sintaxis: db.<colName>.createIndex({ Field: “text” }, ”Index Name”);
  - El índice resultante tiene como elementos las palabras del campo especificado.
  - Mongo solamente soporta un índice de este tipo por cada colección de documentos.
  - Las consultas de este índice hacen un OR sobre las palabras contenidas en el string.
    - Sintaxis: db.<colName>.find({ $text: { $search: “palabra o palabras” } });

- Dispersos:
  - Sintaxis: db.<colName>.createIndex({Field: <1/-1>}, { $sparse: true }, ”Index Name”);
  - Se usan cuando una clave aparece en pocos documentos.
  - Se les pueden aplicar $unique ya que que este tipo de índice ignora los nulos.
- Con caducidad:
  - Sintaxis: db.<colName>.createIndex({Field:Val},{expireAfterSeconds:<int>}, ”Name”);
  - Elimina los documentos con el valor especificado después de los segundos indicados.
- Geoespaciales:
  - Plano cartesiano:
    - db.<colName>.createIndex({ FieldName: “2d” });
    - Inserción: db.<colName>.insert({ FieldName: [<xValue> , <yValue>] });
    - Búsqueda de todos los elementos más cercanos a un punto dado:
      - db.<colName>.find({Field: { $near: [<xV>,<yV>],$maxDistance: <radians>}})
    - Búsqueda de todos los elementos que pueden englobarse dentro de una figura:
      - db.<colName>.find({ FieldName: { $geoWithin: <figure> } });
      - Figuras: $polygon, $box $center
  - Plano esférico:
    - db.<colName>.createIndex({ FieldName: “2dsphere” });
    - Inserción: db.restaurants.insert({ FieldName : [<LongValue> , <LatValue>] });
    - Búsqueda de todos los elementos más cercanos a un punto dado:
      - db.<colName>.find({
        FieldName: {
        $near: {
        $geometry: { type: 'Point', coordinates: [<LongV>,<LatV>] } ,
        $maxDistance: <meters>, $minDistance: <meters>
        }
        }
        })
    - Búsqueda de todos los elementos dentro de una figura:
      - Cambiar el criterio “type” de $geometry a $polygon u otras formas.

Otras opciones para la creación de índices:

- Sintaxis: db.<colName>.createIndex({ Fields }, { options }, ”Index Name”);
- Opciones:
  - $background: true → Cuando se crea un índice por primera vez el servidor debe leer todos los datos de la colección y luego crea el índice especificado. Si se hace en primer plano entonces las conexiones a la BDD se quedan esperando hasta que este proceso termine, algo que puede llevar mucho tiempo. Con esta opción el índice se crea en segundo plano sin bloquear las conexiones y solo deja usar el índice cuando este ya se ha terminado de construir totalmente.

Tipos de operaciones sobre índices:

- Consultar los índices:
  - Sintaxis: db.<colName>.getIndexes();
    - Devuelve los índices de la colección. Por defecto siempre se crea el índice de “_id”.
- Borrado de índices:
  - Sintaxis: db.<colName>.dropIndex(“Index Name”);
  - Sintaxis: db.<colName>.dropIndex({ Field: <1/-1> , Field: <1/-1> , … });
    - Borra el índice con el nombre o el conjunto de claves de ordenación espcificadas.
  - Sintaxis: db.<colName>.dropIndexes();
    - Borra todos los índices de la colección especificada.

Optimización:

- El comando “explain”:
  - Se utiliza para devolver información sobre cómo se han ejecutado las operaciones: aggregate, count, find, group, remove y update.
  - Sintaxis: explica = db.<colName>.explain(<<verbosity>>); explica.<operation>;
    - El valor de <<verbosity>> indica cuanta información se ha de mostrar de la operación que se realiza. Valores:
      - “queryPlanner” (default):
        - Se ejecuta el optimizador de consultas para decidir cual es el mejor plan de ejecución, este plan es el que se devuelve.
      - “executionStats”:
        - Contiene información estadística de la ejecución completa del plan ganador.
      - “*allPlansExecution”:*
        - Analiza todos planes de ejecución posibles, algo muy útil para decidir qué indice es más conveniente para nuestra consulta.
  - Datos de un plan de ejecución:
    - Namespace: Base de datos y nombre de la colección operada.
    - WinningStrategy: Detalla el plan de ejecución como un árbol de etapas.
      - Cada etapa pasa sus resultados (Document / Index Value) a su padre.
      - Los nodos internos manipulan documentos o las claves del índice que provienen de los nodos hijos.
      - Los nodos de tipo hoja acceden a la colección o a los índices.
      - En el nodo raíz se obtiene el resultado final.
      - Una etapa (stage) puede tomar alguno de los siguientes valores:
        1. COLLSCAN: Indica que se ha recorrido la colección entera.
        2. IXSCAN: Cuando se ha realizado un recorrido sobre los índices.
        3. FETCH: Recuperación de documentos, puede usar o no un filtro.
        4. PROJECTION: Proyección de lo obtenido en una etapa anterior.
           - Puede provocar el acceso a memoria (Ejem: ¿ Valor de _id ?)

5. SORT: Ha habido que ordenar los datos en memoria (muy lento).
6. SHARD_MERGED (En clústers): Indica que la información está en varias particiones y que ha habido que examinarlas por separado para luego mezclar los resultados.
7. SHARDING_FILTER: Filtro de una partición.
8. SINGLE_SHARD: Cunado solo ha habido que examinar un miembro de la partición.

- InputStage: Indica una etapa hija.
- InputStages: Indica un array de etapas hijas.
- IndexFilterSet: Booleano que indica si se ha aplicado un filtro de índice.
- ServerInfo: Donde se hace la consulta y datos de la versión de mongo.
- RejectedPlans: Array de planes descartados por el optimizador.
- NReturned: Nº de documentos que encajan con la consulta.
- ExecutionTimeMillis: Tardanza en milisegundos con el mejor plan.
- TotalKeysExamined: Nº total de claves del índice recorridas.
- TotalDocsExamined: Nº total de documentos recorridos / escaneados.
- ExecutionStages: Detalles de la ejecución del plan ganador (Árbol).
- AllPlansExecution: Información parcial de la ejecución del mejor plan.
- Shards: Array de documentos para cada shard accedido.

“Conjuntos de réplica” o “Replica Sets”:

Los datos en MongoDB se almacenan en un clúster, un grupo ordenadores. A a los ordenadores que forman parte de un clúster se les denomina nodos, y los datos de las colecciones están repartidos entre estos, lo que permite almacenar cantidades masivas de datos.

Si el almacenamiento del clúster llega a resultar escaso, siempre podemos añadir nuevos nodos, esto se conoce con el nombre de "escalado horizontal".

Un problema de esta arquitectura es que al tener varios ordenadores distintos es fácil que alguno se estropee e incluso que se puedan llegar a perder datos. Para evitar esto cada dato se copia en un conjunto de nodos. A estos conjuntos de nodos se los conocen como "conjuntos de réplica" o “replica sets”. Con esto, simplemente conseguimos que cada documento esté en varios ordenadores para tener un respaldo.

Los conjuntos de réplica también sirven para mejorar la eficiencia ya que podemos escoger el nodo que devuelve la información más rápido.

Características de los conjuntos réplica en MongoDB:

1. No todos los elementos de la réplica tienen el mismo rango. Existe un único nodo primario (máster) y varios secundarios (slaves), siendo el primario el que tiene el contacto con el cliente, atendiendo las peticiones de lectura, escritura, modificación o borrado.

2. El proceso de réplica es asíncrono. Cuando llegan datos nuevos al nodo primario estos pueden tardar un tiempo en propagarse a los secundarios, durante ese tiempo existe cierta inconsistencia. La consistencia es eventual.

Proceso de comunicación cuando se emplean conjuntos de réplica:

1. Un cliente es atendido por el nodo primario, este modificará los datos.
2. El nodo primario informa a los secundarios de los cambios pidiéndoles que ejecuten la misma instrucción, para esto se usará el fichero de "journal".
3. Cada nodo usa un fichero de "journal", este guarda un informe de las operaciones que va realizando. Si un nodo se cae y más tarde se levanta, este utilizará su fichero "journal" y pedirá información a los demás para ponerse al día con las operaciones realizadas.

Resolver inconsistencias debidas a caídas:

- Cuando la caída es de un nodo secundario: No ocurre nada en particular.

- Cuando la caída es de un nodo primario: Los nodos secundarios deben comprobar que al menos la mayoría han perdido la conexión con el nodo primario. Cuando esto ocurre los nodos secundarios han de llegar a un consenso para ver cual de ellos actuará como el nuevo nodo primario, a este proceso se le denomina votación. Para poder garantizar que siempre que se produzca la caída de un nodo primario se pueda votar la elección de un nuevo primario se establece que el número mínimo de nodos secundarios que debe de haber es de 2. Si se da el caso de que el antiguo nodo primario vuelve a estar disponible, este pasará a ser un nodo secundario.

- Nodos árbitro: Son un tipo de nodos especial, estos no contienen datos y solamente se dedican a participar en las votaciones cuando los nodos secundarios se caen.

Configurar un conjunto de réplicas local:

1. Levantamos los nodos:

mongod --port 27001 --replSet <replicaSetName> --dbpath ./data1 --oplogSize 50
mongod --port <N> --replSet <replicaSetName> --dbpath ./data<N> --oplogSize 50

--oplogSize <int> → Define el tamaño máximo de los ficheros oplog en MB. Estos ficheros se utilizan para todos los nodos del conjunto réplica se mantengan como copias idénticas de los datos del nodo primario. Los nodos secundarios consultan este ficheo del nodo primario para mantenerse actualizados, aunque también pueden consultar los ficheros oplog de otros nodos secundarios.

--fork → No bloquea la consola de mongo al levantar el nodo.

--logpath <file> → Esta opción acompaña a --fork para indicar el fichero en el que se graban los mensajes que deberían aparecer por pantalla.

ps -ef | grep mongo → Ver los procesos locales.

lsof -i TCP -s TCP:LISTEN | grep mongo → Ver los servidores locales.

NOTA:

Eliminar un proceso mongod creado con --fork:

mongod --shutdown --port <LocalPort> --dbpath <PathToDataFolder>

2. Configuramos el conjunto de réplica desde el nodo que ya contenga datos, si no, escogemos un nodo al azar. Desde la consola de mongo escribiremos:

```
config = {
          _id: "<replicaSetName>",
          members: [
                {_id:0, host:hostname()+".local:27001" , <Options> },
                {_id:1, host: "127.0.0.0:<LocalPort>" , <Options> },
                {_id:2, host: "localhost:<LocalPort>" , <Options> },
           ]
 }
```

Options:

priority: <int> → 0 (Nunca será elegido para primario, pero puede votar), 0 – 1 (Indica que es menos probable que sea elegido para primario), 1 (Valor por defecto, todos tienen la misma probabilidad de salir elegidos como primario), mayor que 1 (da más prioridad al nodo para convertirse en primario, el máximo está en 1000).

slaveDelay: <int> → El dato se copiará en este nodo transcurridos esos segundos.
arbiterOnly: <bool> → Si indicamos “true”, el nodo será un nodo árbitro.
hidden: <bool> → Oculto por seguridad, los clientes no pueden conectarse a el.
votes: <0/1> → 0 indica que el nodo no puede votar.

3. Inicializamos el conjunto de réplica mediante el comando: rs.initiate(config)

Averiguar si estamos en la terminal del nodo primario → db.isMaster();

Consultar el estado del conjunto de réplica → rs.status();

Cambiar la configuración del conjunto de réplica:

let config = rs.conf();
config.members[<int>].<option>= <value>;
rs.reconfig(config);

Proceso para quitar un servidor del conjunto de réplica:

- Si el nodo es un nodo secundario:
    1. Ejecutamos en un terminal de mongo:

        let connection = connect("localhost:<LocalPort>/admin");

        connection.shutdownServer();

    2. Ahora mismo el nodo está desconectado, podemos verlo aún con rs.status().
    3. Para eliminarlo, ejecutamos: rs.remove("puck:<LocalPort>");
- Si el nodo es un nodo primario:
    1. Para retirar el nodo primario: rs.stepDown();
    2. Ahora que el nodo ya no es primario: rs.remove("<HostName>:<LocalPort>");

Pasos para añadir nuevos nodos al conjunto de réplica:

1. Crear una nueva carpeta de datos para el nuevo nodo.
2. Levantar el nuevo nodo mediante el comando “mongod”.
3. Desde el nodo primario: rs.add({ host: "<HostName>:<LocalPort>", <NodeOptions> })
4. En caso de querer añadir un nodo externo, este debe ser accesible por DNS.

Acceso a la información de un conjunto de réplica:

- Por lo general, solo se pueden realizar operaciones contra el nodo primario.
- Los nodos primarios sólo soportan operaciones de consulta, pero para poder utilizarlas se ha de lanzar desde el nodo réplica el comando: rs.slaveOk();
- Modo de lectura: <cursor>.readPref(<mode>);
  - Instrucciones:
    - db.getMongo().setReadPref(<mode>); → A nivel de BDD.
    - db.<colName>.find().readPref(<mode>); → A nivel de query.
  - Modos de lectura: Only the modes 'primary', 'primaryPreferred', secondary', 'secondaryPreferred', and 'nearest' are supported.",
    - "Primary": Solo se lee en el nodo primario, se aplica por defecto.
    - "Primary Preferred": Intenta leer del primario, si no, de un secundario.
    - "Secondary": Lee solo de algún secundario.
    - "Secondary Preferred": Intenta leer de un secundario, si no, del primario.
    - "Nearest": Intenta leer del nodo más cercano.
- Leer el fichero oplog de un nodo: db.oplog.rs.find().sort({ts:-1}).limit(<int>)

Consistencia de escritura:

- En MongoDB no existe concepto de commit, lo más parecido sería que un cambio se persistiera en la mayoría de los los nodos de respaldo, el resto de nodos tomarán ese valor más pronto o más tarde.
- Se emplea en las operaciones de inserción y actualización mediante la especificación de los siguientes parámetros:
  - { writeConcern: { w: <int/'majority'>, j: <bool>, wtimeout: <milliseconds>} }
    - w → Número de nodos secundarios que deben asegurar la escritura.
    - j → La operación ha de quedar registrada en el fichero journal.
    - wtimeout → Espera máxima para confirmar la consistencia de escritura.

“Particiones” o “Shards”:

Una partición o shard es un conjunto de réplica pero solo de una parte de una colección, ya que esta es dividida por un campo indexado “shard key” en intervalos [mínimo, máximo). Cada una de estas “particiones basadas en intervalos” se dividen en “trozos” o “chunks”, bloques de datos [mínimo, máximo) de 64 MB. Cada documento estará en un único chunk y cada chunk en un único shard.

La ventaja de tener una división de los datos según una shard key es que las consultas se realizan de forma más eficiente.

Por otro lado, la desventaja de tener una división de los datos según una shard key reside en que una distribución de la clave de forma uniforme puede hacer que los datos se distribuyan de forma desigual entre los shards, produciéndose así una sobrecarga de datos en algunos de ellos.

Para asegurar una correcta distribución de los datos existen el splitting y el balancing:

- Splitting: Cuando un chunk se vuelve demasiado grande este será dividido automáticamente mediante el uso de la función split. Esta divide el rango y los datos del chunk en dos.
    - Implica cambios en los metadatos.
    - Se produce cuando hay inserciones o actualizaciones.
    - No migra los datos, por lo que no afecta a los otros shards.
- Balancing: Cuando la distribución de los datos no es equilibrada el balanceador migrará trozos del shard más sobrecargado al shard menos sobrecargado hasta que mejore el balance.
    - Esta operación puede ejecutarse desde cualquiera de los nodos router del clúster. Esta operación se gestiona en segundo plano, una vez acaba se han de actualizar los metadatos para indicar la localización de los trozos desplazados.
    - Si se produce algún error durante el proceso de migración, la operación será abortada y los datos copiados en el shard destino serán eliminados. Si la operación se completase satisfactoriamente, entonces una vez se han copiado todos los datos en el shard destino, se procedería a borrar estos datos migrados del shard de origen.
    - Cuando se añade un shard al clúster la que la distribución de los datos deja de estar bien balanceada. Esto también se da cuando se elimina un shard del clúster.

Capas de la arquitectura de MongoDB:

1. MongoDB Shard Daemons o Enrutadores: Orientan las peticiones de los clientes hacia los correspondientes servidores de datos. Se crean mediante el servicio *mongos*, que es el servicio al que se conectan los clientes en un entorno con particiones. Un servicio *mongos* puede estar o no en la misma máquina del cliente.
2. MongoDB Config Servers o Servidores de configuración: Contienen datos acerca de cómo están repartidos los datos, este tipo de datos es conocido como metadatos.
3. MongoDB Shard Servers o Servidores de partición: Cada partición se corresponde con un proceso "mongod" y está formada por uno o varios chunks.

Factores para construir un clúster:

- Físicos: Número de particiones, factor de réplica y número de servidores de configuración.
- Lógicos: Diseño de las colecciones y Shard Keys.

Pasos para la creación de un clúster:

1. Creación de los servidores de configuración:
    1. Crear sus directorios: rm -rf <cfg1> … <cfgN> ; mkdir <cfg1> … <cfgN>
    2. Lanzar los procesos:

```
mongod --configsvr --replSet confServ --dbpath cfg1 --port 26050 --fork --logpath cfg1/log –logappend
mongod --configsvr --replSet confServ --dbpath cfg1 --port 26050 --fork --logpath cfg1/log –logappend
mongod --configsvr --replSet confServ --dbpath cfg1 --port 26050 --fork --logpath cfg1/log --logappend
```

    3. Creación del conjunto de réplica de los servidores de configuración:

```
mongo --port 26050
conf = {_id: "configurationServers", members: [{ _id: 0, host: "localhost:26050" },  … ]};
rs.initiate(conf);
```

        NOTAS:

        - La opción --configsvr indica que el servidor es un servidor de configuración.
        - La opción --fork indica que se creará un demonio, es decir, un proceso que queda a la espera, esto tiene la ventaja de que no bloquear el terminal.
        - La opción --fork siempre debe ir acompañada de --logpath que indica dónde grabar el fichero log y cual es su nombre. En este fichero se grabará la misma información que se nos mostraría por el shell del sistema.
        - La opción --logappend permite añadir nuevos registros al final del fichero de log cuando una instancia de *mongod* es reiniciada, ya que sin esta opción *mongod* realizaría una backup del log actual y luego crearía otro fichero de log nuevo.
        - Es obligatorio que los servidores de configuración formen un set de réplica.

2. Creamos los shards:
    1. Crear sus directorios: mkdir <p1_1> <p1_2> … <pN_1> <pN_2>
    2. Lanzar los procesos:

```
mongod --shardsvr --replSet p1 --dbpath p1_1 --logpath p1_1/log --port 27100 --fork --logappend --oplogSize 50
mongod --shardsvr --replSet p1 --dbpath p1_2 --logpath p1_2/log --port 27101 --fork --logappend --oplogSize 50
…
```

3. Creación de los conjuntos de réplicas:

```
$mongo --port 27100
config = {_id: "p1", members: [ { _id:0 , host: "localhost:27100" }, … ] };
rs.initiate(config);

$mongo --port N
config = {_id: "p<N>", members: [ { _id:0 , host: "localhost:<port>" }, … ] };
rs.initiate(config);
```

NOTA: La opción --shardsvr indica que el servidor es un shard.

3. Creación de los enrutadores:

1. Lanzar el procesos:

```
mongos --configdb configurationServers/127.0.0.1:26050, …  --fork --logappend --logpath mongos --port 26601
```

2. Añadimos las particiones:

```
sh.addShard("p1/localhost:27100"); … sh.addShard("p<N>/localhost:<port>");
```

3. Veamos que las particiones funcionan consultando el estado del clúster: sh.status();

NOTA: La opción --gonfigdb indica que el servidor es un enrutador.

4. Partir colecciones:

1. Posicionarnos sobre la BDD en la que se encuentra la colección: use <bddName>
2. Habilitar la BDD para trabajar con colecciones partidas: sh.enableSharding("bddName")
3. Partir los datos por rangos de valores mediante el uso de una shard key:
   - sh.shardCollection("bddName.colName",{keyField: 1},<isUnique?-true/false>);
4. Podemos darle a un shard un identificador o sobrenombre:
   - sh.addShardTag("<shardName>","tagName")
5. Podemos decirle a una partición el intervalo de valores que va a gestionar.
   - sh.addTagRange("<bdd.colName>",{sKey:Bellow},{sKey:Superior},"<tagName>")
6. Podemos escoger si activar el balanceador y el auto split: sh.setBalancerState(true);
7. Si no hemos activado el balanceador y el auto split, podemos escoger activar solamente el auto split mediante: sh.enableAutoSplit();
8. Si en algún momento deseamos consultar la distribución de los documentos y de los chunk podemos hacer esto ejecutando: db.<colName>.getShardDistribution();
9. Ahora ya podemos insertar los datos.

Conceptos sobre las claves de partición o shard keys:

- Debe existir un índice sobre esa clave o conjunto de claves. Si no existe debemos crearlo antes de llamar a shardCollection. Hay una excepción: Si la colección aún no existe, índice es creado automáticamente.
    - El índice de la clave de partición es un índice especial. Los servidores de configuración marcan qué intervalo del índice corresponde a cada partición. Esto permite que se busque muy rápido la información. Por eso este es el primer índice que se intenta utilizar a la hora de ejecutar una consulta. Si este índice no se puede usar habrá que pasar la consulta a cada partición y luego reunir los resultados.
    - Que el índice de la clave de partición sea tan importante no disminuye el valor del resto de los índices ya que dentro de cada partición MongoDB puede decidir usar otros índices para localizar la información deseada. En este caso cada partición mantendrá su propia versión del índice, adaptada para sus datos. Cada una de las particiónes elabora sus propias estrategias y selecciona de entre ellas una estrategia ganadora.
- Elegir una clave que discrimine cuantos más valores mejor.
- Si se puede elegir entre una clave única en lugar de otra que no lo es mejor.
- A veces puede parecer parece que la clave de partición siempre será el _id, sin embargo, si una clave se va a usar mucho para la realización de consultas se puede usar esta clave en lugar del _id.
- En un entorno con particiones no podemos tener una clave única que no sea la de partición.
- Una clave de partición no puede ser un array, pero sí una clave compuesta.