Trabajo final, Base de Datos Distribuidas

Agustina Micaela Zimbello 10100/1
Agustin Ignacio Kanner 10338/3

Contenidos

Trabajo final, Base de Datos Distribuidas

La implementación del trabajo se encuentra en la siguiente URL de GitHub
https://github.com/akanner/BDD-Trabajo

Parte 1: Conceptos base

Alternativas de implementaciones del modelo de datos

Alternativa Uno: Distintas colecciones para cada entidad

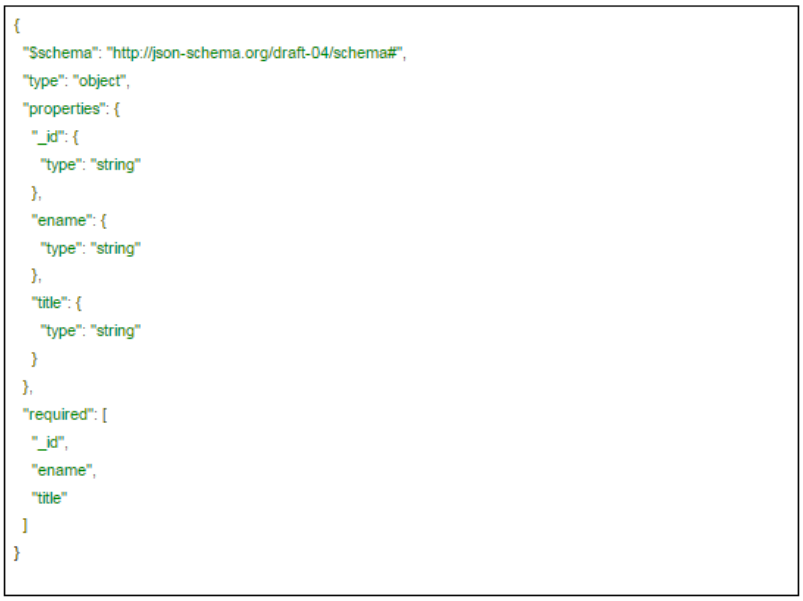
La primera alternativa evidente es utilizar una colección de documentos diferentes para cada tabla del modelo dado, en este caso, la transformación entre las tablas y los documentos de mongodb es directa:

Project, se puede visualizar en la Figura 1.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
  "_id": {
   "type": "string"
  },
  "pname": {
   "type": "string"
  },
  "budget": {
   "type": "integer"
  }
 },
 "required": [
  "_id",
  "pname",
  "budget"
 ]
}
```

Figura 1: Esquema Json [1] del documento "Project"

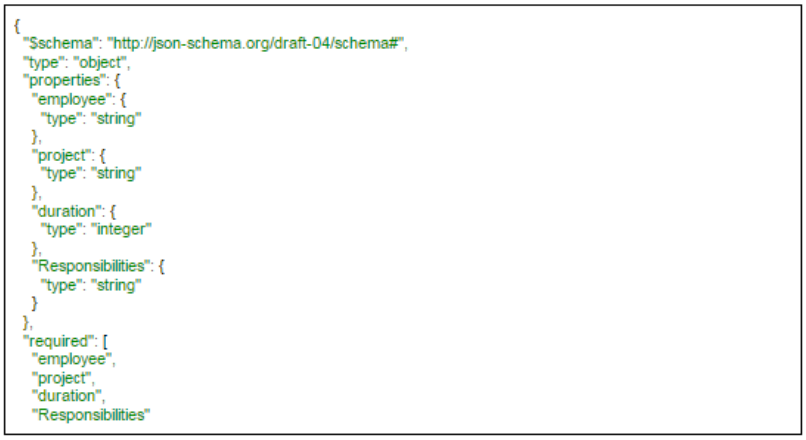
Employee, se puede visualizar en la Figura 2



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "_id": {
      "type": "string"
    },
    "ename": {
      "type": "string"
    },
    "title": {
      "type": "string"
    }
  },
  "required": [
    "_id",
    "ename",
    "title"
  ]
}
```

Figura 2: Esquema Json[1] del documento "Employee"

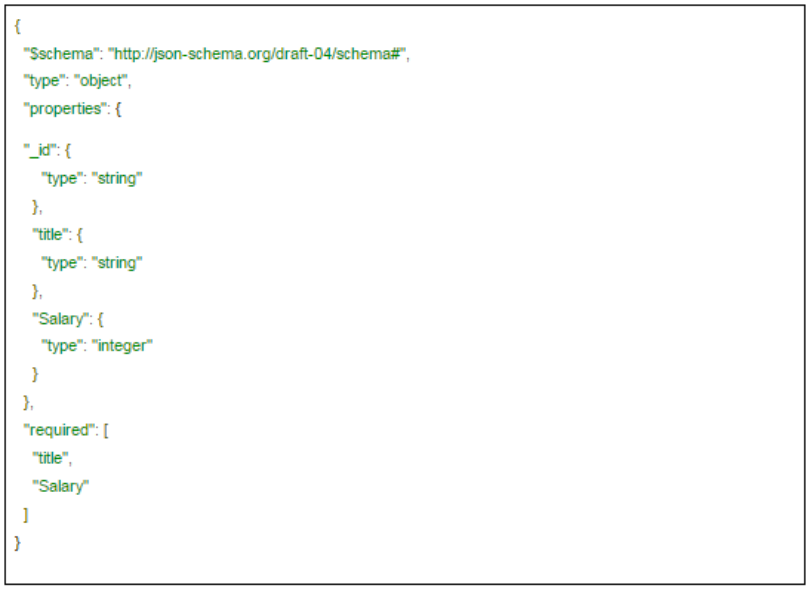
Assignment, se puede visualizar en la Figura 3.



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "employee": {
      "type": "string"
    },
    "project": {
      "type": "string"
    },
    "duration": {
      "type": "integer"
    },
    "Responsibilities": {
      "type": "string"
    }
  },
  "required": [
    "employee",
    "project",
    "duration",
    "Responsibilities"
```

Figura 3: Esquema JSON [1] del documento "Assignment"

Payment, se puede visualizar en la Figura 4.



```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {

 "_id": {
    "type": "string"
  },
  "title": {
    "type": "string"
  },
  "Salary": {
    "type": "integer"
  }
 },
 "required": [
  "title",
  "Salary"
 ]
}
```

Figura 4: Esquema JSON [1] del documento "Payment"

Ventajas de esta alternativa

- El modelo de datos está normalizado.
- Diseño fácil de entender para personas con conocimientos en bases de datos relacionales.

Desventajas de esta alternativa

- El modelo no utiliza ninguna característica de la estructura orientada a documentos, como por ejemplo relaciones embebidas o arrays.
- Se requieren hacer joins para realizar operaciones que requieran actualizar diferentes colecciones, utilizando mongobd, esto se traduce a diferentes consultas a la base de datos.

Alternativa Dos: Entidad "Assignment" Embebida

Una segunda alternativa consiste en "embeber" la entidad "**Assignment**" dentro de los diferentes documentos de la colección "**Projects**", dentro de este enfoque, la entidad "**Project**" tendrá una propiedad llamada "**assignments**" que contendrá un array con las diferentes asignaciones del proyecto, dentro de esta alternativa, el modelo de datos para la entidad "**Projects**" queda de la siguiente manera(ver Figura 5):

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
  "pname": {
   "type": "string"
  },
  "budget": {
   "type": "integer"
  },
  "assignments": {
   "type": "array",
   "items": {
    "type": "object",
    "properties": {
     "employee": {
      "type": "string"
     },
     "responsibilities": {
      "type": "string"
     },
     "duration": {
      "type": "integer"
     }
    }
    "required": [
     "employee",
     "responsabilities",
     "duration"
    ]
   }
  }
 },
 "required": [
  "pname",
  "budget"
 ]
}
```

Figura 5: Esquema JSON [1] del documento "Project" con las asignaciones embebidas

Ventajas de esta alternativa

- No se requiere de varias consultas para obtener las asignaciones de los proyectos.
- Mayor performance en las consultas.

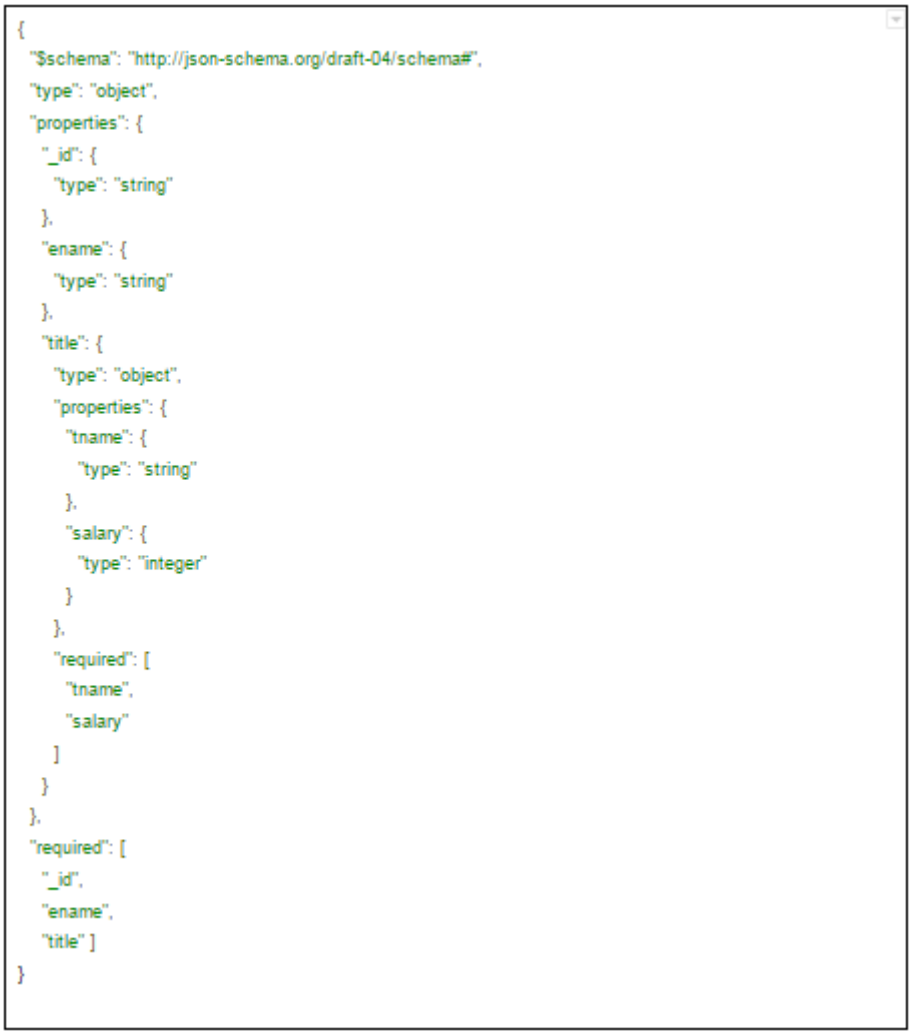
Desventajas de esta alternativa

- Para obtener las asignaciones de una determinada persona se deben buscar las asignaciones de todos los proyectos.

Alternativa Tres: Embeber la entidad Salary dentro de la entidad Employee

Otra alternativa es embeber la entidad "**Salary**" dentro de la entidad "**Employee**", de esta forma cada documento de la colección "**Employee**" contendrá un subdocumento "**Salary**", de esta forma no hace falta realizar joins para conocer el sueldo de cada empleado, así mismo, como cada documento **Employee** posee su propio documento **Salary**, cada empleado puede cobrar un sueldo diferente sin importar el puesto en el que está, sin embargo, para actualizar los sueldos de los empleados con el mismo puesto de trabajo se requiere recorrer toda la colección "**Employee**".

En esta alternativa, la entidad "**Employee**" queda representada de la siguiente manera(ver Figura 6):



```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "_id": {
      "type": "string"
    },
    "ename": {
      "type": "string"
    },
    "title": {
      "type": "object",
      "properties": {
        "tname": {
          "type": "string"
        },
        "salary": {
          "type": "integer"
        }
      },
      "required": [
        "tname",
        "salary"
      ]
    }
  },
  "required": [
    "_id",
    "ename",
    "title" ]
}
```

Figura 6: Esquema JSON[1] del documento "Employee" con la entidad "Payment" embebida

Ventajas de esta alternativa

- No se requieren joins para saber el salario de un empleado.
- Cada empleado puede cobrar un sueldo diferente.
- Se puede actualizar el sueldo de un empleado sin alterar el sueldo del resto.

Desventajas de esta alternativa

- El nombre del puesto (tname) se repite en diferentes empleados con el mismo trabajo.
- Si se desea cambiar el sueldo de todos los empleados en el mismo puesto se deben recorrer todos los empleados de la colección.

Alternativa Elegida

Luego de experimentar con las diferentes versiones, finalmente nos decidimos por la alternativa número uno, es decir, cada entidad del modelo de entidad-relación provisto es correspondido por una colección de mongobd distinta. Nuestros principales justificativos para esta decisión fueron:

- En este modelo, la información no estará repetida (a diferencia de la alternativa tres), por ende, la base de datos será más consistente.
- La performance de la mayoría de las consultas se verá afectada debido a los diferentes joins que podrían existir entre las colecciones, sin embargo, la performance de las mismas puede incrementarse utilizando un nivel de cache.
- Finalmente la alternativa dos(embeber la entidad assignments dentro de la entidad projects lograría una consulta que requiera obtener todas las asignaciones de un proyecto de forma más rápida, debido a que no se requerirían joins para unir los assignments con los projects, sin embargo obtener todas las asignaciones de un empleado determinado requeriría recorrer toda la colección de projects y por cada proyecto, recorrer todas las assignments para identificar aquellas que correspondan con el empleado requerido.

API - Consultas/Tiempos

Las pruebas se hicieron utilizando distintas bases de datos (mismo esquema) con diferente cantidad de tuplas(documentos) en cada una. Las distintas bases de datos son las siguientes:

- BD1: Base de datos vacía
- BD2:
    - Cantidad de tuplas en employees: Cien mil (100.000).
    - Cantidad de tuplas en payments: Mil (1.000).
    - Cantidad de tuplas en assignments: Cien mil (100.000).
    - Cantidad de tuplas en projects: Mil (1000).
- BD3:
    - Cantidad de tuplas en employees: Quinientos mil (500.000)
    - Cantidad de tuplas en payments: Cinco mil (5.000)
    - Cantidad de tuplas en assignments: Quinientos mil (500.000)
    - Cantidad de tuplas en projects: Cinco mil (5.000)

Consultas

Consulta	**Tiempo BD1**	**Tiempo BD2**	**Tiempo BD3**	**Comentarios**
GET payment	25 ms	156 ms	497 ms[1]	Recupera todos los elementos de la colección **payments**.
GET payment/ID	16 ms	46 ms	35 ms	Recupera un documento de la colección **payments**.
POST payment	30 ms	35 ms	54 ms	Guarda un documento nuevo a la colección **payments**.
PUT payment/ID	45 ms	36 ms	27 ms	Actualiza un documento de la colección **payments**.
DELETE payment/ID	35 ms	16 ms	19 ms	Elimina un documento de la colección **payments**.
GET employee	31 ms	8,06 s[1]	-	Recupera todos los documentos de la colección **employees**.
POST employee	23 ms	51 ms	51 ms	Guarda un documento nuevo en la colección **employees**. Valida que el id ingresado en la propiedad **title** exista en la colección **payments**.
GET employee/ID	34 ms	40 ms	35 ms	Obtiene un documento de la colección **employees**. "Popula" el campo **title** con una entidad **payment**.
PUT employee/ID	31 ms	66 ms	33 ms	Actualiza un documento de la colección **employees**. Valida que el id ingresado en la propiedad **title** exista en la colección **payments**.
DELETE employee/ID	24 ms	35 ms	27 ms	Elimina un documento de la colección **employees**.
POST project	30 ms	38 ms	36 ms	Agrega un documento a la colección **projects**.
GET project	42 ms	501 ms[1]	1,9 s[1]	Recupera todos los documentos de la colección **projects**.
GET project/ID	31 ms	90 ms	81 ms	Obtiene un documento de la colección **projects**. Obtiene los documentos de la colección **assignments** cuyos ids se

				encuentran en la colección "assignments" del documento.
PUT project/ID	34 ms	65 ms	57 ms	Actualiza un documento de la colección **project**.
DELETE project/ID	15 ms	62 ms	54 ms	Elimina un documento de la colección **projects**.
POST assignment	47 ms	77 ms	88 ms	Agrega un documento en la colección **assignments**. Verifica que los ids de employee y project correspondan a documentos existentes.
GET assignment	27 ms	10,5 s[1]	-[2]	Obtiene todos los documentos de la colección **assignments**.
PUT assignment/ID	30 ms	48 ms	132 ms	Actualiza un documento de la colección **assignments**. Verifica que los ids de employee y project correspondan a documentos existentes.
DELETE assignment/ID	37 ms	34 ms	68 ms	Elimina un documento de la colección **assignments**. Este es más costoso ya que se eliminan también las referencias al assignment eliminado.

Léase:
[1] La consulta recupera todos los elementos de la colección, por ende, el orden de magnitud de la consulta es de O(n).
[2] El servidor se quedó sin memoria
La generación de estas tuplas se realizó de forma aleatoria a través de llamadas a la API creadas dentro de un script llamado client.js (link)

API - Reutilización de código

- El proyecto incluye un controlador genérico llamado genericController, a través del mismo, se pueden crear nuevos controladores para nuevos recursos de forma más sencilla ya que genericController provee una serie de métodos para acceder, guardar y eliminar a los distintos documentos necesarios.
- La clase responseFormatter permite devolver al usuario respuestas con un formato similar sin importar de los contenidos de la misma (por distintos contenidos nos referimos a un documento simple, una colección de documentos, o un error en el requerimiento.

API - Cosas para mejorar en el futuro

- Crear una capa de servicios entre los controladores y los modelos de Mongoose para lograr una mejor separación de las capas de la aplicación.
- Mongoose: Usar promises en vez de callbacks.

- Utilizar herencia entre los controladores y el controlador genérico.
- Utilizar la sintaxis ES6 para lograr una sintaxis similar a java.
- Agregar más opciones de filtros en los métodos GET de la api, como operaciones del estilo LIKE, GREATER THAN o LESSER THAN[1]

Parte 2 Aplicación de conceptos de BDD

Infraestructura y tecnología involucrada.

Se utilizó Docker[2] para mantener 4 réplicas, donde:

- Dos toman el papel de esclavo
- Una de maestro.
- Una de árbitro

Configuración de docker

Para crear servidores de mongodb utilizamos la imagen oficial de mongo[3], con la misma creamos 4 contenedores utilizando el comando que se puede ver en la Figura 1:

```
sudo docker run \
-p %puerto-host%:27017 \
--name %nombre-contedor% \
--net %nombre-red-docker% \
mongo mongod --replSet %nombre-replica-set%
```

Figura 1: Comando para la creación de 4 contenedores

En donde:

- **%puerto-host%** representa un puerto de la máquina host de los contenedores, el parámetro -p vincula un puerto del contenedor docker (en este caso 27017) con el puerto especificado por **%puerto-host%,** esto provocará que cualquier llamada a localhost:puerto-host realizada desde la máquina host será vinculado con el puerto 27017 del contenedor docker. De esta forma en nuestra aplicación se podrá hacer referencia al servicio de mongodb a través de localhost:puerto-host.
- **%nombre-red-docker%** representa la red virtual en donde se encuentra conectado el contenedor. Para crear estas redes es necesario utilizar el comando sudo docker network create **%nombre-red-docker%,** con esta red nos aseguramos que todos los contenedores tengan acceso a cada uno de ellos ya que utilizamos la misma red para todos los contenedores involucrados
- mongo mongod --replSet **%nombre-replica-set%**
    - mongo es el nombre de la imagen con la cual creamos el contenedor.

---

[1] Existe un paquete de nodejs que permite construir una API Rest de forma automatica partiendo de los modelos de moongose (https://github.com/baugarten/node-restful), el mismo paquete provee una gran cantidad de filtros para las consultas GET.
2 https://www.docker.com/
3 ttps://hub.docker.com/_/mongo/

- mongod inicia el deamon de mongo con el parámetro --replSet que indica el nombre del set de réplicas al que forma parte el contenedor que estamos creando.
- **%nombre-contenedor%** indica el nombre con el que será conocido el contenedor docker

Utilizando este comando creamos 4 contenedores de docker con los datos se visualizan en la Figura 2:

Nombre	Puerto (%puerto-host%)	Red	Replica-Set
mongo1	30001	mongo-replicas-network	mongo-replicas-network
mongo2	30002	mongo-replicas-network	mongo-replicas-network
mongo3	30003	mongo-replicas-network	mongo-replicas-network
arbiter	30004	mongo-replicas-network	mongo-replicas-network

Figura 2: Información de los contenedores de docker utilizados

Luego de ejecutar este comando se inicia el controlador creado, para volver a iniciarlo en el futuro se debe ejecutar:

```
docker start -i #nombrecontainer
```

A continuación, configuramos el set de réplicas accediendo a cualquier contenedor. Para poder realizar acciones sobre el servicio de mongo se debe ejecutar el siguiente comando:

```
docker exec -it #nombrecontainer mongo
```

Este comando nos abrirá una conexión con el servidor de mongo en el contenedor especificado por #nombrecontainer. una vez abierta la conexión debemos ingresar la configuración apropiada para crear el set de réplicas (ver Figura 3):

```
//configuracion basica de un set de réplicas
config = {
        "_id" : "mongo-replicas-network",
        "members" : [
                {
                        "_id" : 0,
                        "host" : "mongo1:27017"
                },
                {
                        "_id" : 1,
                        "host" : "mongo2:27017"
                },
                {
                        "_id" : 2,
                        "host" : "mongo3:27017"
                },
                {
                        "_id" : 3,
                        "host" : "arbiter:27017",
                        "arbiterOnly" : true
                }
        ]
}
rs.initiate(config)
```

Figura 3: Configuración básica de un set de replicas

Cambios necesarios en la API

Para utilizar las réplicas dentro de nuestra API se debieron realizar cambios mínimos dentro del proyecto. Puntualmente, y debido a la capacidad del ODM (object-document mapping) mongoose, solo se debio modificar el conection string y las opciones de conexion a la base de datos, a modo de ejemplo, previo a la implementación de réplicas, la conexión de mongoose con la base de datos se realizaba de la siguiente manera:

```
mongoose.connect("mongodb://localhost:27017/api_dev",errorCallback);
```

Para utilizar el set de réplicas se debió cambiar la línea anterior por la siguiente:

```
mongoose.connect("mongodb://localhost:30001,localhost:30002,localhost:30003/api_db?replicaSet=mongo-replicas-network",dbOptions, errorCallback);
```

Como se puede ver, la url de la base de datos contiene las urls de cada réplica disponible, así mismo contiene el nombre del set de réplicas dado por el parámetro replicaSet, con esta configuración es posible que el servidor se conecte al servidor primario del set y cambie de servidor en caso de alguna elección dentro del set [2]. Así mismo, durante la conexión son seteadas los siguientes parámetros dentro de la variable dbOptions:

```
{
  "replSet": {
    "auto_reconnect": true,
  }
}
```

Donde:
replSet representa un objeto con las propiedades orientadas a configurar el comportamiento del set de réplicas, mongoose pasará estas opciones directamente al driver de mongodb[3], dentro del mismo se encuentra:

- auto_reconnect: Por defecto, la API puede reconectarse al servidor primario en caso de que haya una elección y se elija un nuevo primario. Sin embargo, cuando por alguna razón no se puede elegir un servidor primario, la API no podrá realizar más operaciones, inclusive si luego de un tiempo, el set de réplicas pueda realizar una elección y la misma "devuelva" un nuevo servidor primario. Este parámetro hace que la aplicación intente reconectarse de forma automática durante el caso explicado anteriormente, cuando el set de réplicas vuelva a contar con un servidor primario, la API se reconectara con el mismo y se reanudarán las operaciones.

Errores y soluciones.

Usando mongodb://user:password@server1,server2,server3/database?replicaSet=repname
Encontramos que teníamos que cambiar la versión de 4.7.3 - 4.5.9 de Mongoose, ya que arrojaba el siguiente error MongoError: no primary found in replicaset. Investigando encontramos que es un error que se presentaba en las versiones posteriores de 4.59 de Mongoose.

Experimento: Operaciones de lectura sin servidor primario.

De acuerdo con la documentación de mongodb[4][5], un cliente puede realizar operaciones de lectura sobre un servidor secundario[5], pero solo puede escribir sobre un servidor primario[4], durante el desarrollo de este trabajo intentamos llevar a la práctica esta capacidad de mongodb e intentamos mantener disponibles las operaciones de lectura de la API cuando el set de réplicas subyacente no cuenta con un servidor primario. Para lo cual, utilizamos diversos parametros de conexion del ODM Mongoose, hasta que obtuvimos la configuración adecuada para lograr responder operaciones de lectura sin servidor primario. Los cambios que debimos realizar fueron los siguientes:

1. Permitir que la API lea de servidores secundarios en caso de no haber un primario

Para lograr que la api lea datos de servidores secundarios si no existe un servidor primario tuvimos que agregar un parámetro extra a la URL de conexión a la bd, dicho parámetro es readPreference=primaryPreferred según la documentación de mongo[6] este parámetro hace que las operaciones de lectura se resuelven en el servidor primario, pero, en caso de que este se encuentre fuera de servicio, las mismas serán resueltas en un servidor secundario. Luego del cambio nuestra url quedó de la siguiente manera:

```
mongodb://localhost:30001,localhost:30002,localhost:30003/api_db?replicaSet=mongo-replicas-network&readPreference=primaryPreferred
```

Donde:

- *localhost:30001,localhost:30002,localhost:30003* la lista de hosts de las réplicas de la base de datos.
- *api_db* el nombre de nuestra base de datos.
- *mongo-replicas-network* el nombre de nuestro set de réplicas.
- *primaryPreferred* la estrategia de lectura utilizada.

En caso de no recibir este parámetro, la API encolará las operaciones de lectura hasta que el servidor primario vuelva a estar disponible.

En el caso de las operaciones de escritura, de no existir un servidor primario la API devolverá un error en cualquiera de los casos:

```
{
  "status": 500,
  "message": "no primary server available"
}
```

2. Permitir que la API inicie sin servidores primarios[Opcional]

Al momento de iniciar la API, la misma devolverá un error si no se encuentra disponible un servidor primario, para evitar esto, se puede agregar el siguiente parámetro a las opciones de conexión del método connect de mongoose:

```
{
  "replSet": {
    "connectWithNoPrimary": true,
  }
}
```

connectWithNoPrimary indica que la aplicación arrancara sin tener un servidor primario disponible.
Quedando la configuración completa de la siguiente manera:

```
{
  "replSet": {
    "connectWithNoPrimary": true,
    "auto_reconnect": true,
  }
}
```

Distintos casos del uso de réplicas.

Caso	Esclavo	Esclavo	Esclavo	Maestro	Arbitro	Observaciones
1	Up and running	Up and running	Up and running	Up and running	Up and running	El servidor está levantado sin problemas. Y el cliente puede "pegarle" a la API sin problemas.
2	Up and running	Up and running	Down	Up and running	Up and running	El servidor está levantado sin problemas. Y el cliente puede "pegarle" a la API sin inconvenientes. Observamos que si se detiene cualquier réplica cuya función sea esclavo, el comportamiento de la API sigue siendo el esperado, mientras que exista la cantidad suficiente de réplicas para realizar la votación de elección de un nuevo primario.
3	Up and running	Down	Down	Up and running	Up and running	Se encuentran caídas dos réplicas esclavas(secundarias) ya no se posee la mayoría para realizar la elección del nuevo primario. Por lo tanto, el servidor maestro pasa a ser esclavo. La API encola las operaciones de lectura recibidas por parte del cliente (se puede configurar la API para que utilice un servidor secundario.) y devuelve un error en el caso de las operaciones de escritura. Mientras que el servidor primario no se transforme en secundario, la API continúa realizando operaciones normalmente (esto sucede por una fracción de segundo hasta que la votación sea infructuosa y el servidor primario deje de serlo).
4	Up and running	Up and running	Up and running	Down	Up and running	En este caso, el servidor primario de mongoDB deja de responder, inmediatamente a esto, la API devuelve un error de conexión con la base de datos hasta que el set de réplicas elija un nuevo servidor primario.
5	Up and running	Down	Down	Down	Down	Al igual que el caso 3, no existe la mayoría para realizar la votación del maestro, dependiendo de la configuración de la API, las operaciones de lectura serán encoladas o se realizará sin problemas, en cuanto a las operaciones de escritura, las mismas no estarán disponibles hasta que no se encuentre disponible un nuevo servidor primario.

Figura 4 Distintos casos observados durante la implementación de las replicas

Con respecto al encolamiento, que se mencionó en los distintos casos se realizan las siguientes aclaraciones:

- Mongoose[4] encola operaciones por defecto cuando no se encuentra disponible una base de datos. Lo mismo aplica a cuando no se posee configurado un servidor primario.
- El tamaño de la cola de operaciones es indefinido, como así también configurable. Por lo cual es posible que se deje de encolar en dos situaciones la 1) la memoria se agota o 2) Se logra volver a tener un servidor primario.
    - La manera de configurar la API para determinar la cantidad de operaciones permitidas a encolar, es agregar a la funcion de connect de moongose la siguiente configuracion.
      mongoose.connect(uri, {db:{bufferMaxEntries:100);
      En la línea de ejemplo, se setean 100 operaciones como máximo a ser encoladas. En caso que se definiera el valor 0, significaría que la API va a atender de a un requerimiento a la vez, sin encolar ninguna operación. Esto se encuentra relacionado con el hecho de que node.js es mono thread.

Write Concerns [7]

El write concern determina el grado de consentimiento que debe tener la información dentro de un set de réplicas de mongoDB para que la misma sea válida, este grado de consentimiento no es más que la cantidad de réplicas en las que la información debe existir antes de considerar a la operación de escritura como válida o finalizada.

[4] http://mongoosejs.com/

Por defecto, mongoBD únicamente requiere que los cambios producidos por una operación de escritura se encuentren en el servidor primario para considerar a la operación como finalizada, sin embargo, es posible modificar esa configuración para requerir que los cambios se vean reflejados en 2 o más réplicas antes de dar por finalizada la operación. Esto se logra especificando "cuanto" write concern necesita una operación de escritura.

```
db.products.insert(
  { item: "envelopes", qty : 100, type: "Clasp" },
  { writeConcern: { w: <value>, j: <boolean>, wtimeout: <number> } }
)
```

En el cuadro anterior se observa una operación insert en la cual se especifica el write concern requerido para dicha operación donde:
w especifica la cantidad de réplicas en donde se requiere propagar los cambios antes de dar por finalizada la operación. El valor de w puede ser:

- [0-9]+: indica la cantidad de nodos que deben configurar la escritura de los cambios antes de finalizar la operación.
- "mayority": Indica que los cambios deben ser confirmados por la mayoría de los nodos antes de finalizar la operación.
- <tag>[5]: Los cambios deben ser confirmados por todos los nodos que contengan dicho tag en su configuración

wtimeout indica un timeout para la consulta, la misma disparara un error en caso de agotarse el timeout pero **mongoDB no revertirá los cambios que se hayan producido dentro de la base de datos**, en caso de no definir esta propiedad, si no se logra obtener la cantidad de write concern necesaria la operación quedará bloqueada indefinidamente hasta lograr el consentimiento necesario.
J: indica si la operación debe estar escrita en el journal[6] para ser considerada como completa.

Un ejemplo de write concern podría ser el siguiente

```
db.products.insert(
  { item: "envelopes", qty : 100, type: "Clasp" },
  { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

La ejecución del insert anterior se resolvería de la siguiente manera:

[5] https://docs.mongodb.com/manual/tutorial/configure-replica-set-tag-sets/#replica-set-configuration-tag-sets
[6] https://docs.mongodb.com/manual/core/journaling/

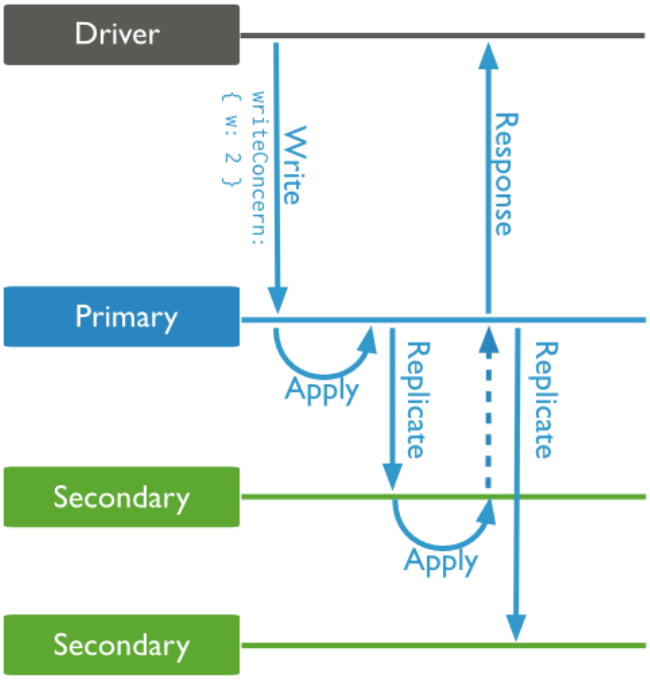


Figura 5 Operación de escritura con Write concern = 2

En caso de querer cambiar el write concern utilizado por defecto (w:1) se puede configurar al set de replicas con los siguientes comandos:

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = { w: 2, wtimeout: 5000 }
rs.reconfig(cfg)
```

Si bien mongoBD no revertirá los cambios en caso de un error, un nivel de consentimiento alto permitirá devolver un error en el caso de que el servidor primario quede fuera de servicio antes de replicar los datos hacia los demás nodos del set.

Write concerns dentro de la API

IMPORTANTE:

Mongoose ignorara el write concern indicado en el set de réplicas subyacente (posiblemente debido a que este especifica un write concern para cada operación provocando un override de la configuración propia del servidor).

Para subsanar esto es preferible definir adicionalmente el write concern dentro de las opciones de conexión a la base de datos mediante:

```
db: {
        w: 2,
        wtimeout: 5000
}
```

A continuación, describiremos algunas operaciones de escritura sobre la base de datos utilizando diferentes opciones de write concern.

Todas las pruebas se realizaron en una base de datos con la siguiente cantidad de tuplas:

- payments: 10.000 tuplas.
- employees: 100.000 tuplas
- projects: 100.000 tuplas.
- assignments: 100.000 tuplas.

Sobre la verificabilidad de los modos de escritura

Lamentablemente no pudimos corroborar que durante las las operaciones anteriores la replicación se haya realizado antes de que la API devuelva los resultados al cliente debido a que en cualquiera de los modos de write concern, la replicación se realiza antes de que podamos consultar a los diferentes nodos de la red, sin embargo, podemos corroborar que, la API devolverá un error en caso de no cumplirse el write concern requerido durante las distintas operaciones por medio del siguiente caso:

Teniendo un write concern "alto" procedemos a apagar una réplica secundaria, haciendo esto no afectamos al servidor primario de base de datos, por ende, las operaciones de escritura se deberían poder resolverse correctamente, sin embargo, al tener la restricción del write concern en un nivel alto, el set de réplicas no lograra obtener el consentimiento necesario para considerar a la operación correcta.

Para probar este caso enviamos el siguiente requerimiento a la URL http://localhost:8000/api/payment/ para dar de alta un nuevo elemento en la colección payments

```
{"title":"no_hay_concern","salary":9999}
```

El set de réplicas subyacente se encuentra en el siguiente estado:

Nombre réplica	Estado
mongo1	online
mongo2	**offline**
mongo3(**primario**)	online
mongo4	**offline**
arbiter	online

Hemos definido un timeout para la operación de 5000 ms (5 segundos) y un write concern de w:4, por ende, el resultado que esperamos ver es un error debido a que el timeout definido se agota antes de obtener el consentimiento necesario para completar la operación.

Efectivamente, luego de transcurrir el tiempo definido por el timeout, la API devuelve el siguiente mensaje:

```
{
  "status": 500,
  "message": "waiting for replication timed out"
}
```

Sin embargo, podemos ver que en los servidores que se encuentran online, el documento se encuentra persistido correctamente:

```
mongo-replicas-network:PRIMARY> db.payments.find({"title" : "sin_concern"})
{
  "_id" : ObjectId("5887a39c43fa358a1fe53dfb"),
  "title" : "sin_concern",
  "salary" : 9999,
  "__v" : 0 }
}
```

output en mongo3 (servidor primario)

```
mongo-replicas-network:SECONDARY> db.payments.find({"title": "sin_concern"})
{
  "_id" : ObjectId("5887a39c43fa358a1fe53dfb"),
  "title" : "sin_concern",
  "salary" : 9999,
  "__v" : 0
}
```

output en mongo1 (réplica)

De esta forma se puede ver que, si bien la operación no alcanzó el nivel de consentimiento necesario (w:4), mongo no revierte los cambios producidos en los diferentes nodos de la base de datos.

Read Concern [8]

El read concern determina los datos que el set de réplicas devolverá a cada una de las consultas realizadas sobre el mismo, existen 3 niveles de read concern:

- **Local**: Este es el nivel por defecto, la consulta devolverá los datos más recientes del servidor consultado. No existen garantías de que la información devuelta se encuentre en los demás nodos del set.
- **Majority**: La consulta devolverá la información que ha sido confirmado por la mayoría de los nodos del set.
- **Linealizable**: La consulta devolverá los datos que hayan sido confirmados mediante una writeConcern con w: “mayority” y estas confirmaciones se hayan producido antes de la consulta de lectura.

Experimento: Falta de consentimiento durante lectura

Utilizaremos el nivel majority de read concern, para simular una falta de consentimiento durante una lectura dentro del set de réplicas vamos a utilizar el comando slaveDelay7, este comando indica a una réplica que debe esperar una cierta cantidad de tiempo para sincronizarse con el resto del set, lo cual provocará una diferencia de datos con los nodos que no posean un delay.

Para esta prueba 2 de los cuatro nodos del set tendrán un delay de una hora, lo que provocará que, inmediatamente después de "setear" el delay, las réplicas no actualizarán ningún dato hasta que se cumpla la hora definida (luego actualizarán siempre con una hora de retraso). El set de réplicas queda configurado de la siguiente manera:

Nombre replica	Estado
mongo1(**primario**)	online
mongo2	online (**delay: 1 hora**)
mongo3	online (**delay: 1 hora**)
arbiter	online

ACLARACIÓN: para habilitar el read concern majority tuvimos que crear un nuevo set de réplicas pasando la opción --enableMajorityReadConcern, al comando de docker quedando de la siguiente manera:

```
sudo docker run \
-p %puerto-host%:27017 \
--name %nombre-contedor% \
--net %nombre-red-docker% \
mongo mongod --replSet %nombre-replica-set% --enableMajorityReadConcern
```

Con esta configuración, procedemos a establecer el read concern mediante un parámetro en la conexión de mongoose con el servidor de la base de datos

```
db: {
        readConcern: {
                level : "mayority"
        }
}
```

Y procedemos a hacer la siguiente consulta a la API:

```
POST http://localhost:8000/api/payment/
```

Con el siguiente contenido

```
{"title":"repl_con_delay","salary":9999}
```

[7] https://docs.mongodb.com/manual/tutorial/configure-a-delayed-replica-set-member/

La API nos devuelve lo siguiente


```
{
  "status": 200,
  "message": {
    "__v": 0,
    "title": "repl_con_delay",
    "salary": 9999,
    "_id": "5888f79b678fbabc5379eab7"
  }
}
```


Por consiguiente, buscamos ese nuevo documento enviando el siguiente requerimiento a la API

```
GET http://localhost:8000/api/payment/5888f79b678fbabc5379eab7
```

En consecuencia, la API queda bloqueada completamente, esto es debido a que la query queda esperando al concern indefinidamente (en este caso esperara 1 hora), para solucionar esto modificamos la consulta para incluir (temporalmente para este experimento) un timeout como sugiere la documentación de read concern de mongo[8] quedando la invocación de la consulta

```
Model.findById(entityId).maxTime(10000) //máximo 10 segundos
```

En este caso, el read concern no logra a completarse en los 10 establecidos y la API devuelve


```
{
  "status": 500,
  "message": "operation exceeded time limit"
}
```


Conclusiones sobre la implementación del replica set

Antes de la implementación del set de réplicas, la API dependía exclusivamente de un solo servidor de bases de datos, en caso de que el mismo dejase de funcionar la API ya no podía brindar ninguno de sus servicios, con cambios muy pequeños en el código fuente (cambios en la url de la base de datos) se logró que la API contará con un set de réplicas de base de datos, lo cual permite a la API tolerar fallos en algunos nodos del set y recuperarse de los mismos luego de que los servidores vuelvan a funcionar e inclusive, con un poco más de esfuerzo, es posible mantener funcionando las operaciones de lectura ante un fallo más grave, como es el caso de no encontrarse activo un servidor primario dentro del set de réplicas.

Con respecto a la implementación y mantenimiento del set de réplicas en sí, claramente es más complejo que implementar y mantener un solo servidor de mongoDB, sin embargo, más allá de la cantidad de trabajo que lleve levantar cada réplica del set, la configuración del set en

sí mismo (como conjunto de nodos actuando como uno solo) no es compleja y se puede configurar un set básico en un corto plazo de tiempo.

En cuanto a los tiempos de ejecución, si bien la la propagación de los cambios a través de los nodos del set agrega más carga a los servidores (en cuanto a acceso a disco, memoria,red) creemos que, esta carga adicional no resultará en una degradación del servicio proporcionado por la API o cualquier aplicación que utilice un set de réplicas, y si, esta carga llegara a ser un problema, la misma siempre podrá ser mitigada con la incorporación de hardware más potente.

Otra ventaja de la implementación del set de réplicas es que es posible realizar las operaciones de lectura sobre el nodo "más cercano"(utilizando un readPreference: nearest), sin bien esto es irrelevante para nuestra API, un sistema con múltiples servidores web sería capaz de responder consultas de lectura más rápidamente utilizando los servidores de base de datos que tengan menos latencia con el servidor web utilizado.

En cuanto a los write/read concern, resulta claro que, a medida que el consentimiento requerido para una determinada operación aumenta, la misma inferirá más tiempo para ser completada (quizás no se vea reflejado en nuestros experimentos ya que nuestra infraestructura no refleja la realidad de un verdadero sistema distribuido en diferentes servidores físicos). Sin embargo, un nivel de consentimiento alto ayudará a mitigar inconsistencias de datos cuando se producen fallos en los distintos nodos del set, aunque requeriría de un esfuerzo extra de parte de los desarrolladores para realizar rollbacks en aquellas réplicas donde se produjeron los cambios de operaciones que, finalmente, no lograron obtener el consenso necesario para completarse.

Finalmente, como una opinión personal, creemos que, luego de haber realizado este trabajo, veríamos con buenos ojos la implementación de un set de réplicas en un sistema real, especialmente cuando exista la posibilidad de mantener las lecturas y escrituras solo en el servidor primario y utilizar las réplicas como simples "suplentes" en caso de que el mismo quede fuera de servicio debido a que, prácticamente, no hay que realizar cambios dentro del código de la aplicación

Parte 3 Conceptos avanzados de BBDD
Fragmentación

Sharding[9]

Sharding(fragmentación) es un método para distribuir datos a través de múltiples máquinas. MongoDB utiliza sharding para soportar deployments con conjuntos de datos muy grandes y operaciones que requieren alto rendimiento.

Los sistemas de bases de datos con grandes conjuntos de datos o aplicaciones de alto rendimiento pueden desafiar la capacidad de un solo servidor. Por ejemplo, las altas tasas de consulta pueden agotar la capacidad de la CPU del servidor. El tamaño de los working set (conjuntos de trabajo) mayores que la RAM del sistema tensionan la capacidad de E / S de las unidades de disco.

Existen dos métodos para abordar el crecimiento de los sistemas: escalamiento vertical y horizontal.

Vertical Scaling implica aumentar la capacidad de un solo servidor, como usar una CPU más potente, agregar más memoria RAM o aumentar la cantidad de espacio de almacenamiento. Las limitaciones de la tecnología disponible pueden restringir que una sola máquina sea suficientemente potente para una carga de trabajo determinada. Además, los proveedores basados en la nube tienen techos rígidos basados en configuraciones de hardware disponibles. Como resultado, hay un máximo práctico para la escala vertical.

Horizontal Scaling implica dividir el conjunto de datos del sistema y la carga en varios servidores, añadiendo servidores adicionales para aumentar la capacidad según sea necesario. Si bien la velocidad o capacidad de una sola máquina puede no ser alta, cada máquina maneja un subconjunto de la carga de trabajo global, proporcionando potencialmente una mejor eficiencia que un único servidor de alta velocidad de alta velocidad. La ampliación de la capacidad de la implementación sólo requiere la adición de servidores adicionales según sea necesario, lo que puede ser un costo total menor que el hardware de gama alta para una sola máquina.El trade off es una mayor complejidad en infraestructura y mantenimiento para el deployment.

MongoDB soporta Horizontal Scaling a través de fragmentos.

Sharded Cluster

Un sharded cluster es un conjunto de Replica Sets (shards) cuya función es la de repartirse uniformemente la carga de trabajo, de tal manera que nos permite escalar horizontalmente nuestras aplicaciones para así poder trabajar con grandes cantidades de datos.

Un sharded cluster en MONGODB está compuesto por los siguientes componentes:

- **shard** (fragmento): Cada fragmento contiene un subconjunto de los datos compartidos. Cada fragmento puede ser deployado como una réplica set.

- **mongos**: los mongos actúan como un enrutador de querys, proporcionando una interfaz entre las aplicaciones clientes y el sharded cluster.

- **config servers**: Almacenan metadata y las configuraciones seteadas para el cluster. Los config servers deben ser deployados como una replica set (CSRS).

En la Figura 3.1 se puede ver la interacción entre los componentes dentro de un sharded cluster.

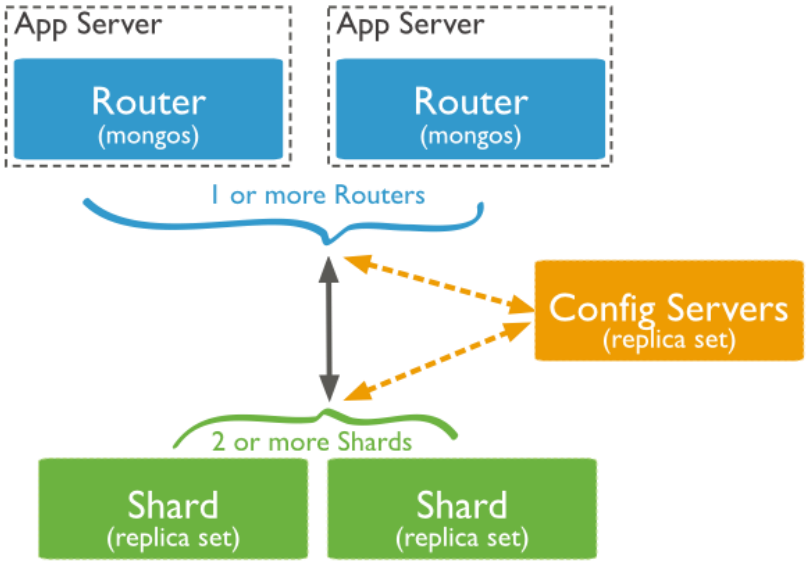


Figura 3.1: Componentes dentro de un Sharded Cluster

MONGODB fragmenta los datos a nivel de colección, distribuyendo las colecciones de datos a través de los fragmentos en el cluster.

Shard Keys

Para distribuir los documentos en una colección, MongoDB particiona la colección usando una Shard Key (clave de fragmento). La clave shard consiste en uno o más campos inmutables que existen en cada documento de la colección a fragmentar.

La shard key es elegida cuando se fragmenta una colección. La elección de la shard key no se puede cambiar luego de que se realizó fragmentación, y una colección fragmentada sólo puede tener solamente una Shard key.

Para fragmentar una colección no vacía, la colección debe tener un índice que comience con la shard key. Para las colecciones vacías, MongoDB crea el índice si la colección aún no tiene un índice adecuado para la shard key especificada.

La elección de la shard key afecta el rendimiento, la eficiencia y la escalabilidad de un clúster fragmentado. Un clúster con el mejor hardware e infraestructura posible puede ser afectado de manera negativa (puede causar un cuello de botella) por la elección de la shard key. La elección de la shard key y su índice de respaldo también puede afectar a la estrategia de fragmentación que el clúster puede utilizar

Chunks

MongoDB particiona los fragmentos de datos dentro de chunks. Cada chunk tiene un rango inclusivo inferior y exclusivo superior basado en la shard key.

MongoDB migra los chunks a través de los fragmentos en el sharded cluster utilizando el equilibrador de shared cluster. El equilibrador intenta conseguir un equilibrio uniforme de chunks a través de todos los fragmentos del clúster.

Ventajas de la fragmentación

Reads / Writes

MongoDB distribuye la carga de trabajo de lectura y escritura a través de los fragmentos del sharded cluster, permitiendo que cada fragmento procese un subconjunto de operaciones de clúster. Las cargas de trabajo de lectura y escritura se pueden escalar horizontalmente en el clúster añadiendo más fragmentos.

Para las consultas que incluyen la shard key o el prefijo de una shard key compuesta, los mongos pueden derivar la consulta a un fragmento (o conjunto de fragmentos) específicos. Estas operaciones dirigidas son generalmente más eficientes que realizar un broadcasting a cada fragmento en el shard cluster.

Capacidad de almacenamiento

La fragmentación distribuye los datos a través de los fragmentos del clúster, permitiendo que cada fragmento contenga un subconjunto del total de datos del clúster. A medida que el conjunto de datos crece, los fragmentos adicionales aumentan la capacidad de almacenamiento del clúster.

Alta disponibilidad

Aunque no se puede acceder al subconjunto de datos de los fragmentos no disponibles durante el tiempo de inactividad (during the downtime), las lecturas o escrituras dirigidas a los fragmentos que si están disponibles pueden realizarse con éxito.

A partir de MongoDB 3.2, se pueden deployar config servers como réplicas sets. Un sharded cluster con un Config Server Replica Set(CSRS) puede continuar procesando lecturas y escrituras siempre y cuando la mayoría del conjunto de réplicas esté disponible. En los entornos de producción, los fragmentos individuales deben deployarse como réplica sets, así proporcionan mayor redundancia y disponibilidad.

Auto balanceo de carga a través de los shards

El balanceador decide que datos migrar y a que shard para que los datos se encuentren uniformemente distribuidos entre todos los servidores.

Consideraciones antes de la fragmentación

Los requisitos y la complejidad de las infraestructuras de Sharded Cluster requieren una cuidadosa planificación, ejecución y mantenimiento.

Una consideración cuidadosa en la elección de la shard key es necesaria para asegurar el rendimiento y la eficiencia del cluster. No se puede cambiar la clave shard después de sharding, ni se puede "desfragmentar" una colección fragmentada.

Colecciones fragmentables y no fragmentables

Una base de datos puede tener una mezcla de colecciones fragmentables y no fragmentables. Las colecciones fragmentadas se dividen y se distribuyen a través de los fragmentos del clúster. Las colecciones no almacenadas se almacenan en un fragmento primario. Cada base de datos tiene su propio fragmento primario.

En la Figura 3.2 se puede observar el ejemplo de la "Collection1", la cual está fragmentada en Shard A y Shard B, mientras que la "Collection" no se puede fragmentar y se coloca en el fragmento primario Shard A.

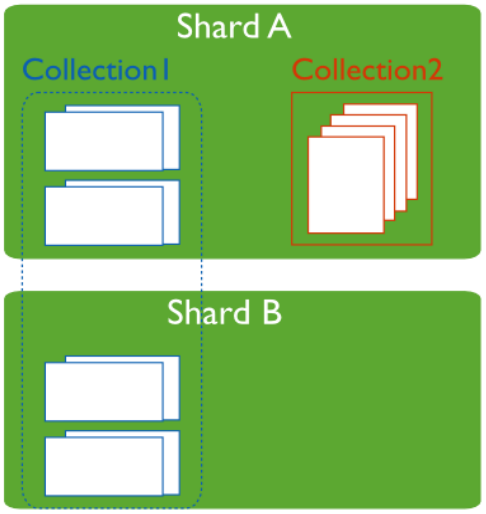


Figura 3.2: Colecciones y fragmentación.

Conexión al Sharded Cluster

Se debe conectar a un router mongos para poder interactuar con cualquier coleccion que se encuentre en el cluster fragmentado. Esto es tanto como para las colecciones fragmentables y las no fragmentables. Los clientes nunca deberían conectarse a un solo fragmento para realizar las operaciones de lectura o escritura.

Un ejemplo de conexión puede verse en la Figura 3.3.

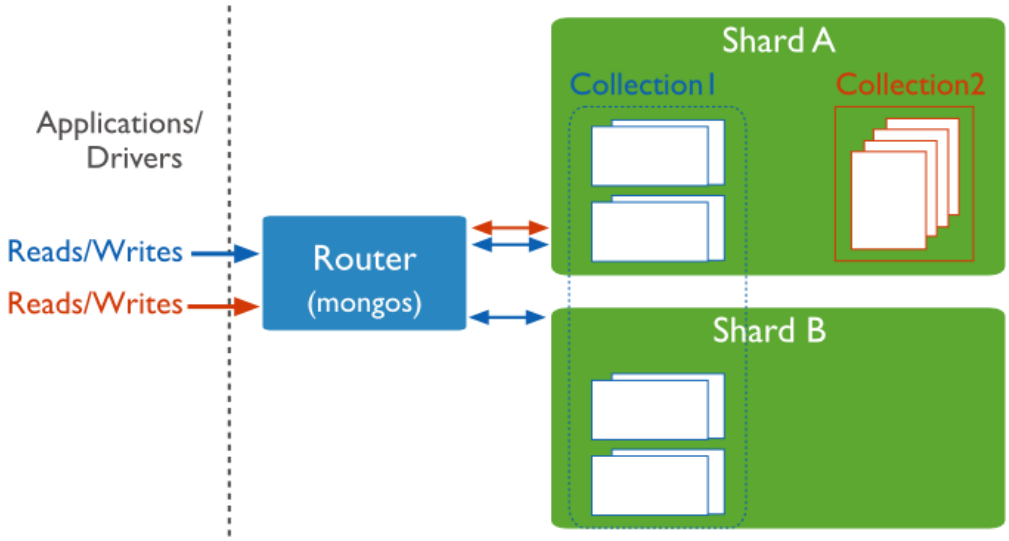


Figura 3.3: Ejemplo de conexión

La conexión a un mongos se realiza de la misma manera que cuando se realiza una conexión a un mongod, ya sea a través de la shell de mongo o con un driver de MongoDB.

Estrategias de Fragmentación

MongoDB soporta dos tipos de estrategia de fragmentación para la distribución de los datos a través de Sharded Clusters.

Hashed Sharding

Involucra hacer un cálculo un hash del valor del campo de la shard key. A cada fragmento se le asigna un rango basado en los valores del hash de la shard keys.En la Figura 3.4 se puede ver un ejemplo de Hashed Sharding.

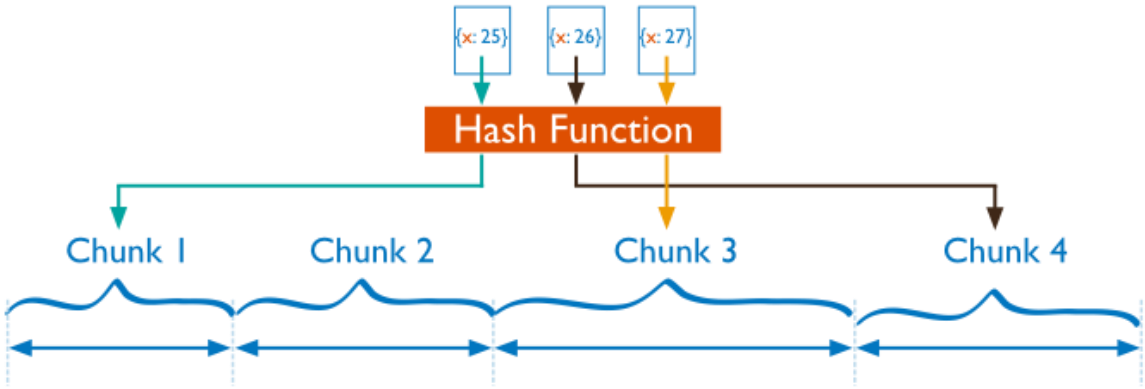


Figura 3.4: Hashed Sharding

Mientras que el rango de las shard key puede ser cercano, es casi imposible que sus valores del hash se encuentren en el mismo chunk. La distribución de datos basada en valores de hash facilita una distribución más uniforme de los datos, especialmente en conjuntos de datos en los que la shard key cambia de forma constantemente.

Sin embargo, la distribución utilizando hash significa que las consultas basadas en rangos en la shard key son menos propensas a apuntar a un único fragmento, lo que resulta en más de broadcaster en todo el clúster

Ranged Sharding

El Ranged sharding implica dividir los datos en intervalos basados en los valores de shard keys. Cada chuck se le asigna un rango basado en los valores de la shard key.

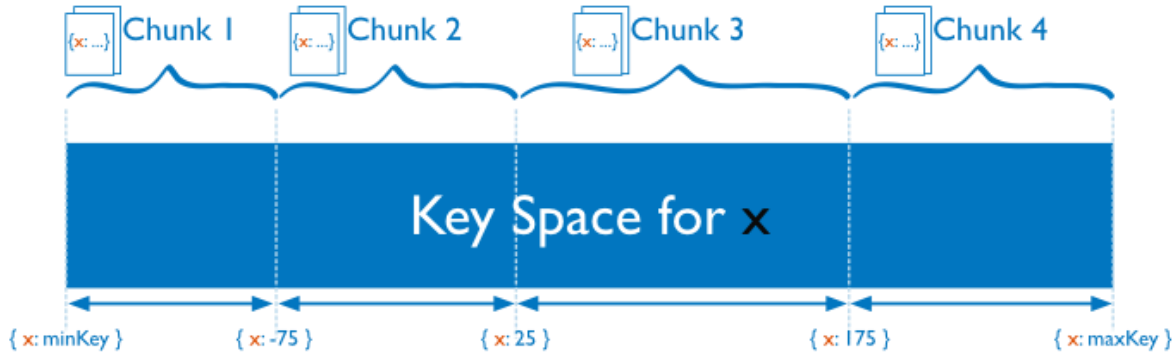


Figura 3.5: Ranged Sharding

Un rango de shard key cuyos valores son "cercanos" tienen más probabilidades de residir en el mismo fragmento. Esto permite operaciones específicas, ya que los mongos pueden encaminar las operaciones sólo a los fragmentos que contienen los datos requeridos.

La eficiencia del ranged sharding depende de la shard key elegida. Las shard key mal consideradas pueden resultar en una distribución desigual de los datos, lo que puede anular algunos beneficios de la fragmentación o puede provocar cuellos de botella en el rendimiento.

Zonas en los sharded cluster

En los sharded cluster, se pueden crear zonas de datos fragmentados basados en la shard key. Se asocia cada zona con uno o más fragmentos del clúster. Un fragmento puede asociarse con cualquier número de zonas no conflictivas. En un cluster equilibrado, MongoDB migra los chunks de acuerdo a la zona.

Cada zona cubre uno o más rangos de valores de la shard key.

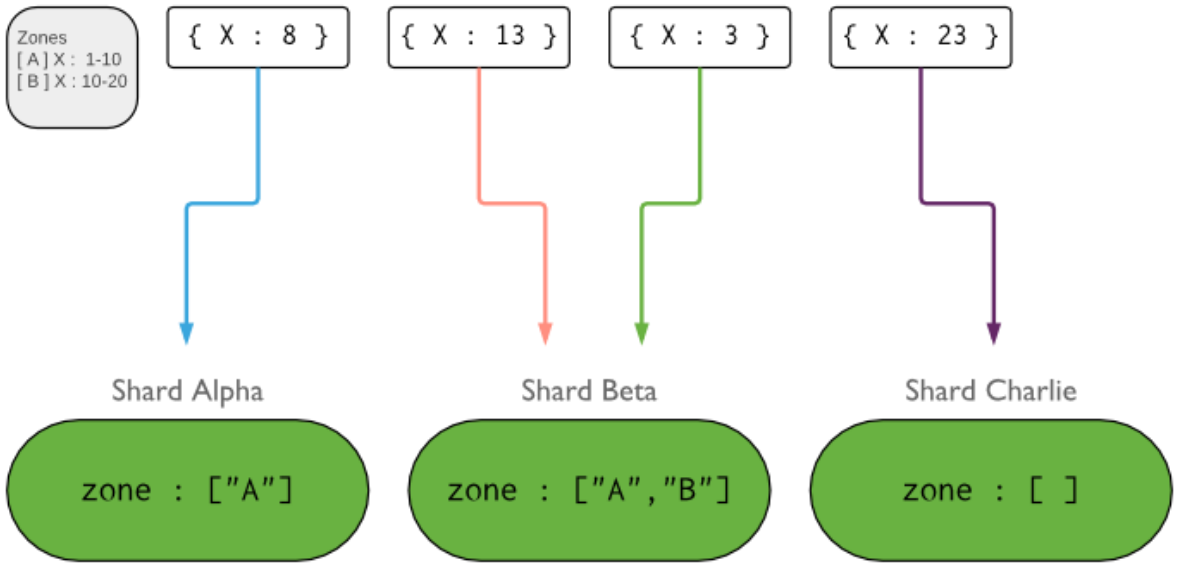


Figura 6: Zonas en los Shard Cluster.

Se debe utilizar los campos contenidos en la shard key al definir un nuevo rango a cubrir por una zona.

Las zonas sirven para mejorar la ubicación de los datos en los Sharded Cluster que abarcan múltiples centros de datos.

Cambios necesarios en la API

Para implementar sharding en nuestra API será necesario realizar 3 tareas diferentes:

1. Crear la infraestructura necesaria de base de datos.
2. Particionar las colecciones.
3. Modificar las conexiones de la API para apuntar a uno o más servidores del tipo mongos

Infraestructura de base de datos

Para implementar sharding debemos crear el cluster de bd necesario, esta infraestructura requiere:

- **Shards**: nodos en donde se encuentran los datos fragmentados, cada shard puede encontrarse replicado.
- **Mongos**: Estos nodos abstraen a los clientes de los shards del cluster, los clientes se conectan a estos nodos y estos dirigen las consultas a los shards indicados por los config servers.
- **Config servers**: estos servidores contendrán información para “rutear” las consultas de los servidores mongos a través de los distintos shards del sistema.

Para crear estos nodos se deben levantar diferentes instancias de mongod, estas instancias pueden ser levantadas a través de docker, utilizando distintos puertos o directamente en diferentes distintos servidores físicos.

Config Servers

Los config servers se crean iniciando mongod con el parámetro --configsvr. Utilizando nuestro enfoque anterior donde utilizamos docker, se utilizará el siguiente comando:

```
sudo docker run \
-p 30001:27017 \
--name %nombre-contedor% \
--net %nombre-red-docker% \
mongo mongod --configsvr
```

De esta forma podemos crear la cantidad de config servers que sean necesarios.

Shards

Los Shards son creados iniciando mongod utilizando el comando --shardsvr, cada shard puede estar replicado, esto es posible mediante el parámetro --replSet que hemos usado anteriormente, utilizando docker, los shards se definen utilizando el siguiente comando:

```
sudo docker run \
-p 30002:27017 \
--name %nombre-contedor% \
--net %nombre-red-docker% \
mongo mongod --shardsvr --replSet shard-a
```

Mongos

Finalmente se deben crear los nodos que actuarán como intermediarios entre el cliente y los distintos shards del cluster, estos nodos son llamados mongos y se crean mediante el comando mongos y requiere conocer las ips de los config servers.

```
mongos --configdb config-server-1-url,config-server-2-url,config-server-3-url
```

Utilizando docker, el comando sería el siguiente:

```
sudo docker run \
-p 30002:27017 \
--name %nombre-contedor% \
```

```
--net %nombre-red-docker% \
mongo mongos --configdb config-server-1-url,config-server-2-url,config-server-3-url
```

Configurando los sets de réplicas para cada shard

Cada shard puede estar formado por un set de réplicas, por lo tanto, cada set debe ser configurado, para lograrlo, se puede utilizar una configuración como la siguiente:

```
{
  _id : <replicaSetName>,
  members: [
    { _id : 0, host : "shard1-mongo1.example.net:27017" },
    { _id : 1, host : "shard1-mongo2.example.net:27017" },
    { _id : 2, host : "shard1-mongo3.example.net:27017" }
  ]
}
```

y luego utilizar un nodo del set de réplicas para aplicar la configuración

```
rs.initiate(config)
```

Agregando los shards al cluster

Para agregar los shards previamente creados, se debe conectarse a un servidor mongos, esto se puede lograr utilizando el comando mongo especificando el host mediante el parametro --host.

```
mongo --host url-mongos1.net
```

Una vez dentro del mongos, se procede a agregar a los shards de la siguiente manera:

```
sh.addShard( "s1-mongo1.example.net:27017")
//en caso de que cada shard tenga un set de replicas
sh.addShard( "<replSetName>/s1-mongo1.example.net:27017")
```

Una vez hecho esto, el cluster ya se encuentra disponible para utilizar sharding

Particionar las colecciones

Para poder utilizar sharding dentro de una base de datos es necesario ejecutar el siguiente comando:

```
sh.enableSharding("<database>")
```

Finalmente se pueden particionar las diferentes colecciones mediante la definición de una shard key, en este caso utilizando rangos

```
sh.shardCollection("<database>.<collection>", { <key> : <direction> } )
```

Tomando un ejemplo de nuestra API, la partición de la colección payments podría ser de la siguiente manera:

```
sh.shardCollection("api_db.payments", { salary : 1 } )
```

El comando anterior creará una shard key utilizando el campo salary con orden ascendente (direction: -1 es descendente)

Una vez creada la shard key, mongo balanceara los datos automáticamente a través de todos los shards disponibles.

Apuntando a los nuevos mongos

Lo unico que deberiamos cambiar en nuestra API para implementar sharding es cambiar las URLs de los servidores de mongo para que apunten a los servidores mongos, para ello debemos cambiar las urls a las que apunta mongoose en el método connect, debido a que utilizamos un archivo de configuración para guardar las URLs de los servidores, podríamos cambiar dicho archivo (link) y definir los siguientes hosts:

```
"hosts": [
        "mongos1.net",
        "mongos2.net"
  ]
```

Por otro lado, no debemos utilizar el set de réplicas que habíamos definido, por ende debemos eliminar el parámetro de la url de conexión con la bd. Dentro del archivo

/src/utils/mongoConnectionStringHelper.js (link) se debería modificar el método getMongoConnectionString para que quede de la siguiente manera:

```
exports.getMongoConnectionString = function(){
        var mongoUrl = config.Mongo.client + "://" + getDbHostsString(config.Mongo.hosts)
+ "/" + config.Mongo.dbName + ";
        return mongoUrl;
}
```

Habiendo hecho esto, la API quedaría configurada para utilizar los servidores mongos que, a su vez, utilizarían los shards definidos anteriormente.

Referencias

[1] JSON Schema, http://json-schema.org/
[2] Replica sets,Mongoose,
http://mongoosejs.com/docs/connections.html#replicaset_connections
[3] Opciones de conexión,Mongoose, http://mongoosejs.com/docs/connections.html#options
[4] Primary Servers, MongoBD, https://docs.mongodb.com/manual/core/replica-set-primary/
[5] Secoundary servers, MongoDB, https://docs.mongodb.com/manual/core/replica-set-secondary/
[6] Read preference, MongoDB, https://docs.mongodb.com/manual/core/read-preference/
[7] Write Concerns,MongoDB,https://docs.mongodb.com/manual/reference/write-concern/
[8] Read Concerns,MongoDB, https://docs.mongodb.com/manual/reference/read-concern/
[9] MongoDB
https://docs.mongodb.com/manual/sharding
[10] MongoDBSpain:
http://www.mongodbspain.com/es/2015/01/26/how-to-set-up-a-mongodb-sharded-cluster/
[11] Varios:
https://youtu.be/WU5rIUKJ9Fo (shard key selection)
https://youtu.be/L7V8Izihttc (planning a mongoDB sharded Cluster)