UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Estudio de rendimiento en MongoDB sobre arquitecturas centralizadas y distribuidas
Autores: Pablo José Soldi
Director: Prof. Fernando G. Tinetti
Asesor profesional: Lic. Franco Agustín Terruzzi
Carrera: Licenciatura en Sistemas

Resumen

La tesina de grado tiene como objetivo principal investigar de manera experimental las facilidades de replicación y distribución de datos de una base de datos NoSQL en particular, siendo MongoDB el seleccionado.
Como caso de estudio se ha elegido el almacenamiento de datos que no son totalmente estructurados, de paso para aprovechar la experimentación con datos NoSQL. A través de la escritura y lectura de los datos, se busca obtener tiempos de respuesta y a su vez tener chequeos de consistencia. Se espera que los diferentes experimentos provean información relevante tanto para la instalación como para la operación de estas formas de almacenamiento y recuperación de datos.
La experimentación parte de una primera estructura centralizada, luego pasa a una estructura centralizada con réplicas y finaliza en una distribuida.

Palabras Claves

Bases de datos distribuidas - NoSQL - MongoDB - Réplica - Fragmentación - Tiempos de respuesta - Consistencia

Trabajos Realizados

Se creó un manual detallando todos los pasos para la configuración, instalación y puesta en marcha de las estructuras distribuidas que van a analizarse de MongoDB.
Se creó una aplicación web que permite unificar todas las funcionalidades necesarias para experimentar sobre MongoDB.

Conclusiones

Las bases de datos implementadas en arquitecturas distribuidas permiten mejorar los tiempos de respuesta de una centralizada. Mientras que comparando entre estructuras distribuidas se puede mejorar en prácticamente un 50% los tiempos de respuesta si se tienen tres shards en lugar de dos.
MongoDB demostró ser consistente para escribir y leer en el manejo de grandes volúmenes de datos. Por otro lado su configuración inicial para las distintas estructuras requiere de una curva de aprendizaje grande para su uso.

Trabajos Futuros

Resulta interesante investigar qué otras opciones además de Mongo se pueden utilizar para almacenamiento distribuido de documentos y analizar entre ellas similitudes y diferencias.

Fecha de la presentación: DICIEMBRE 2017

A mi mamá y a mi papá que los amo con el corazón.
A mi hermana y mis hermanos que los amo con el corazón.
Al abuelo Mario, a la tía Elsa y a mis tíos.
A mis amigos del glorioso Colegio Nacional y a mis amigos del glorioso Club Everton.

A todos ellos gracias.

Índice

1. Introducción

La tesina de grado a realizar tiene como objetivo principal analizar de manera experimental las facilidades de replicación y distribución de la información de una base de datos NoSQL. Se ha elegido MongoDB como motor de base de datos para el análisis de

a) Características (dificultad/sencillez, hardware necesario, etc.) para configuración y utilización de replicación y distribución de datos.

b) Rendimiento en diferentes escenarios de almacenamiento de datos: no replicados, replicados y distribuidos con replicación.

Para llevar a cabo los distintos estudios, se intentará insertar una colección de datos que no sean totalmente estructurados. Para lo cual, uno de los desafíos será tener una gran variedad de archivos de diferente tipo, contenido y tamaño. A través de la escritura y lectura de los distintos archivos y sus metadatos, se busca poder obtener los tiempos de respuesta y a su vez tener chequeos de consistencia en la información guardada y recuperada. Se espera que los diferentes experimentos, provean información relevante tanto para la instalación como para la operación de estas formas de almacenamiento y recuperación de datos.
La propuesta parte de una primera estructura centralizada, pasa a una estructura centralizada con réplicas y finaliza en una distribuida con sharding (donde un shard contiene solamente una parte del total de datos de la base de datos) que son replicados. Dentro de la opción distribuida, se analiza el rendimiento de diferentes cantidades de shards entre los que se reparte la información.
Esta secuencia a su vez tiene como objetivo mostrar las diferentes alternativas y dificultades que se presentan en cada caso.
Si bien las pruebas se realizan en escenarios planteados desde MongoDB, gran parte de los conceptos a utilizar y analizar son transversales a las bases de datos distribuidas.

1.1 Uso de bases de datos distribuidas

Se define [1.1] una base de datos distribuida como una colección de múltiples bases de datos interrelacionadas lógicamente en una red de computadoras. Un sistema de Bases de Datos Distribuida se define como un sistema de software, que permite la gestión de las bases de datos distribuidas de tal forma que un usuario puede acceder los datos en cualquier parte de la red exactamente como si estos fueran accedidos de forma local. En una forma transparente para el usuario.

1.1.1 Camino hacia NOSQL

El intenso crecimiento de distintas aplicaciones y la masividad en el uso generado por internet en los últimos tiempos [1.2, 1.3], trajo consigo un gran crecimiento en el

volumen de los datos, los cuales generalmente se vinculan con la necesidad de obtener una respuesta en tiempo real al momento de procesarlos. Las consecuencias inmediatas a esto, fueron los problemas que los sistemas de gestión de bases de datos centralizados ocasionaron para responder con eficiencia frecuentes demandas de rápido acceso/escritura sobre las bases con grandes cantidades de información.
Las bases de datos distribuidas empezaron a cobrar especial importancia en el área de almacenamiento gracias a su capacidad para ejecutarse en un clúster de gran tamaño. El término clúster (del inglés cluster, "grupo" o "racimo") se aplica a los conjuntos o conglomerados de computadoras construidos mediante la utilización de hardware estándar de bajo costo y que se utilizan como si fuesen una única computadora. Otra forma de verlo, desde el punto de vista de redes, es directamente como una red local (LAN: Local Area Network) instalada para cómputo paralelo/distribuido.
A pesar de este primer paso hacia los sistemas distribuidos que permitían mejorar el rendimiento en comparación del esquema centralizado, los problemas de eficiencia seguían latentes. El modelo relacional era el paradigma sobre el que se habían desarrollado las bases de datos durante décadas. Sin embargo la búsqueda en la mejora de los tiempos de escritura o lectura encontraba limitaciones en este modelo.
El constante crecimiento de los datos y la necesidad de aumentar el rendimiento, derivó en la conclusión de que en muchos casos era más importante la respuesta en tiempo real, que la coherencia de los datos en la que las bases de datos relacionales dedicaban una gran cantidad de tiempo de proceso.
A todo este panorama se sumó el hecho de que gran parte de los productos desarrollados para bases de datos distribuidas eran software privativo. No solo era imposible modificar o conocer el código, sino que generalmente se debían pagar costosas licencias.
Como respuesta a todos estos factores empezaron a aparecer una nueva generación de bases de datos. Una generación que gana cada vez más fuerza y espacio en lo académico y en el mercado bajo licencias open source.
Los sistemas NoSQL proponen una estructura de almacenamiento más versátil que la relacional. Aunque sea a costa de perder ciertas funcionalidades como las transacciones que engloban operaciones en más de una colección de datos, o en algunos casos la incapacidad de ejecutar el producto cartesiano de dos tablas, teniendo que recurrir a la desnormalización de datos.

1.2 NoSQL

El término NoSQL hace referencia a un grupo de bases de datos que difieren en algún aspecto del modelo relacional clásico. En general proponen una estructura de almacenamiento versátil. Pero los distintos sistemas de tipo NoSQL tienen características muy variadas entre sí. Por lo tanto, en la actualidad se traduce el

término “NoSQL” principalmente como “no sólo sql” [1.4]. Sin embargo, esta acepción no es del todo acertada. Si bien es cierto que la mayoría de los manejadores de datos hacen el intento de tener un lenguaje de consulta parecido a SQL porque implica un menor costo en el aprendizaje por el lado del programador. Consultas simples como el JOIN en el esquema relacional requieren mucho trabajo para ser simuladas en los ambientes NoSQL.

1.2.1 Tipos de bases de datos NoSQL

Dentro de la gran variedad de bases de datos NoSQL se destacan cuatro tipos

1.2.1.1 Clave-Valor

Una base de datos clave valor se asemeja mucho a un diccionario o hashtable del mundo de la programación. Con la única diferencia de que está diseñada para almacenar grandes cantidades de información. Consisten en modelar los datos a través un arreglo asociativo, el el cual los datos se representan como atributos nombre junto con su valor [1.5].
El mayor exponente de este tipo en la actualidad es Redis [1.6]. Redis se caracteriza por ser un motor de base de datos que por defecto almacena sus tablas de hashes (clave-valor) en memoria. Esta característica le permite lograr tiempos de lectura y escritura muy por encima de aquellos motores que persisten en disco. Sin embargo, su utilización se limita a casos donde la durabilidad de la información no es algo crítico.

1.2.1.2 Familia de columnas

Una familia de columnas es una base de datos que se compone de columnas de datos relacionados [1.7]. Todas sus tupla son pares. Consisten en un clave-valor, donde la clave se asigna a un valor que es un conjunto de columnas. Haciendo una analogía con las bases de datos relacionales, una familia de columnas es como una tabla y cada par de clave-valor es una "fila".
Su mayor exponente en la actualidad es Cassandra. Se explicará en detalle el funcionamiento en la sección [2.2.1 Apache Cassandra]

1.2.1.3 Documentos

En la noción de un documento, cada entrada o registro puede tener un esquema de datos diferente, con atributos que no tienen por qué repetirse de un registro a otro. Un documento puede contener muchos pares de clave-valor, o clave-arreglos o incluso otros documentos anidados [1.8]. MongoDB es uno de los motores exponentes de este subtipo de bases NoSQL. El capítulo 3 se desarrolla en detalle sus características.

1.2.1.4 Grafos

Una base de datos orientada a grafos (BDOG) representa la información como nodos de un grafo y sus relaciones con las aristas del mismo [1.9], de manera que se pueda usar teoría de grafos para recorrer la base de datos ya que esta puede describir atributos de los nodos (entidades) y las aristas (relaciones).
Una BDOG debe estar absolutamente normalizada, esto quiere decir que cada tabla tendrá una sola columna y cada relación tan solo dos, con esto se consigue que cualquier cambio en la estructura de la información tenga un efecto solamente local.
Neo4j [1.10] desarrollado en java y perteneciente al software libre, es el motor de base de datos orientado a grafos más utilizado en la actualidad [1.11].

1.2.2 Distribución de las Bases de Datos en clusters

Dependiendo del modelo de distribución que tenga la base de datos, es posible almacenar la información, distribuyendo las cargas de procesamiento sobre los distintos nodos del cluster. Un cluster puede procesar un mayor tráfico de operaciones de lectura/escritura en comparación con un esquema centralizado. Pero como contrapartida, introduce una mayor complejidad, dada su estructura, en el manejo de la base de datos [1.12].
Existen dos maneras de distribuir datos en un cluster, la réplica y el sharding, las cuales son técnicas ortogonales, es decir que, se puedes utilizar cualquiera de ellas o ambas.

1.2.2.1 Réplica

El propósito de la réplica puede consistir en múltiples motivos [1.13]

1. Disponibilidad de la información. A través de la réplica se eliminan los puntos de acceso únicos a la información. Si un nodo pasa a un estado inaccesible, se puede acceder a otras réplicas.
2. Rendimiento. La réplica permite ubicar la información cerca de los distintos puntos de acceso, lo cual contribuye a una reducción en los tiempos de respuesta.
3. Escalabilidad. Si una aplicación crece, consecuentemente crecerá en términos de acceso a la información. La réplica permite distribuir y mantener los tiempos de respuesta en tiempos aceptables.
4. Requerimientos de Aplicación. Una aplicación puede tener como requerimiento específico mantener múltiples copias de la información

1.2.2.2 Sharding

El sharding o fragmentación es una técnica que consiste en particionar los datos en distintos nodos agrupándolos con algún criterio específico.

Se lo utiliza para gestionar de manera distribuida la carga de los servidores. Al distribuir los datos entre distintos shards se reparte el procesamiento al realizar consultas e inserciones [1.14].
Un beneficio que ofrece la fragmentación es la disminución de la latencia en base a la geolocalización de los servidores y las aplicaciones que consultan sobre ellos.
En la actualidad muchas bases de datos NoSQL ofrecen sharding automático. Lo que significa que la base de datos asume la responsabilidad de asignar datos a distintos nodos teniendo en cuenta factores como equilibrio en la distribución.

1.3 Elección de MongoDB

En la actualidad MongoDB es la base de datos de tipo NoSQL más utilizada en la industria del software [1.15]. Por otro lado a octubre de 2017 es el quinto manejador de base de datos más utilizado a nivel global [1.15]. Encontrándose por debajo de las cuatro opciones SQL más utilizadas. Oracle [1.16], MySQL [1.17], SQL Server [1.18] y PostgreSQL[1.19]. Al ser principalmente una herramienta colaborativa y de código abierto no es menor que se utilice por gran parte de la comunidad del software. Este uso se refleja en mayor contenido para la utilización de la herramienta.
MongoDB administra de manera automática la actualización de las réplicas y el balanceo de la información sobre los fragmentos o shards. Si bien es necesario tener una curva de aprendizaje en la configuración inicial, su utilización permite delegar algunas funcionalidades complejas. Finalmente, otro motivo para tener en cuenta en su elección fue la forma para escribir y recuperar información. Dejando a un lado comandos como los “join”, muchas otras de sus instrucciones tienen similitudes con el lenguaje SQL. Si bien hay diferencias notables entre el enfoque estructural-relacional y NoSQL, MongoDB capitaliza la experiencia del lenguaje SQL permitiendo escribir consultas lo más parecidas posibles [1.20].

1.4 Organización del contenido

Para el desarrollo del documento se decidió que el contenido de los capítulos se enfoque en los conceptos esenciales y necesarios para comprender la tesina. En base a este objetivo, algunos conceptos son desarrollados en detalle en apéndices.
En los capítulos subsiguientes se desarrollarán los siguientes temas:
Capítulo 2: Se describen conceptos básicos del almacenamiento de datos distribuidos analizados desde las bases de datos relacionales. Además se analizan otros tipos de bases de datos NoSQL alternativas a MongoDB.
Capítulo 3: Se describen las características y propiedades básicas de MongoDB para su funcionamiento en entornos centralizados y distribuidos.

Capítulo 4: Se detalla el diseño de los distintos experimentos que se llevarán a cabo para estudiar el rendimiento de las arquitecturas centralizadas y distribuidas.
Capítulo 5: Descripción de las funcionalidades necesarias para realizar los estudios de rendimiento y todo lo necesario para su implementación.
Capítulo 6: Análisis de los resultados obtenidos y las conclusiones que se desprenden de los mismos.
Capítulo 7: Se hace una conclusión sobre el trabajo realizado y se plantean posibles líneas de investigación para experimentar a futuro.
Capítulo 8: Bibliografía.
Apéndice I: Se describen los distintos tipos de actualización para réplicas y los distintos tipos de fragmentación en almacenamientos distribuidos.
Apéndice II: Descripción del funcionamiento interno de MongoDB para el descubrimiento de los nodos que componen su red. Por otro lado se analiza en detalle cómo se realiza la actualización de los nodos secundarios.
Apéndice III: Instructivo desarrollado a lo largo de la tesina para configurar una estructura distribuida en MongoDB.

2. Trabajos Relacionados

2.1 Diseño de bases de datos relacionales distribuidas

Existen varias diferencias entre las bases de datos NOSQL y las relacionales. Sin embargo, cuando se analiza la tecnología desde el punto de vista distribuido, la réplica y la fragmentación son dos funcionalidades que aparecen en ambos enfoques. Quizás como en el caso de MongoDB con distintos nombres pero siempre son temas similares.
Analizar los distintos enfoques de estas funcionalidades en un esquema relacional distribuido y ver las complicaciones que surgen en base a ellos puede brindar un elemento más para la comparación y el entendimiento de ambas tecnologías [2.1].

2.1.1 Replicación y consistencia

Las bases de datos distribuidas generalmente implementan algún nivel de réplica. Como se explica antes [1.2.2.1] existen diversas ventajas o necesidades que implican el uso de las réplicas tales como:

1. Disponibilidad de sistema.
2. Rendimiento.
3. Escalabilidad.
4. Requerimientos de aplicación.

2.1.2 Estrategias para el manejo de actualizaciones

Si bien la replicación tiene diferentes beneficios existe un desafío en mantener todas las copias actualizadas. Cuando se intente agregar o actualizar un registro la instrucción debe ejecutarse en todas las réplicas. Por lo tanto, a diferencia de la lectura, la escritura será una operación más costosa que en un esquema centralizado ya que implica la actualización en todas las réplicas del sistema.
Los protocolos de replicación son clasificados en base a cuándo las actualizaciones son propagadas a las copias (eager o lazy/ activo o pasivo) y cuándo se permite ejecutar dicha actualización (manejo centralizado o distribuido). Se puede leer el detalle de las estrategias en el **[Apéndice I]**

2.1.3 Fragmentación

La fragmentación consiste en dividir una base de datos en porciones de menor tamaño y administrar cada fragmento como un objeto de base de datos separado.
Existen dos alternativas generales de fragmentación: horizontal y vertical.

La fragmentación horizontal consiste en dividir una relación a lo largo de sus tuplas. Así cada fragmento tiene un subconjunto de tuplas relacionadas bajo algún discriminador.
La fragmentación vertical consiste en dividir una relación a lo largo de sus atributos. Cada subconjunto se compone de algunos atributos (columnas) de la tabla original.
Se encuentra en el **[Apéndice I]** el desarrollo en detalle de los tipos de fragmentación.

2.1.4 Problema de asignación

El problema de la asignación surge al buscar la manera óptima de asignar una colección de fragmentos a través de una colección de nodos que componen a una red de computadoras distribuidas.
El concepto de lo óptimo se puede definir con respecto a dos medidas [2.3]:

1. Costo mínimo: La función del costo consiste en sumar el costo de almacenar todos los fragmentos en los distintos nodos, el costo de consultar sobre todos los fragmentos en todos los nodos, la actualización de todos los fragmentos y el costo de la comunicación entre nodos.
   Abordando la mejor alternativa desde el punto de vista del costo, se intenta buscar un esquema que minimice la combinación de todos los factores.
2. Rendimiento: La estrategia de asignación se diseñan para mantener una métrica de rendimiento. Conociendo la relación entre los fragmentos y las consultas a ejecutar por los clientes, se debe buscar llevar al mínimo los tiempos de respuesta y maximizar el rendimiento del sistema en cada sitio.

Si bien estos conceptos se los propone como medidas diferentes para definir lo óptimo, está claro que ambos factores no son opuestos y que detrás de lo óptimo deberían estar conjugados.

En las bases de datos relacionales las restricciones de consistencia implican cierta complejidad en las modificaciones. El mantenimiento de esquemas fuertemente estructurados en sistemas donde los datos son cambiantes puede volverse complejo. La creación de tablas auxiliares para no modificar un esquema o la adición de campos que puedan ser vacíos suelen ser ejemplos del problema.
Por el contrario utilizar un enfoque NoSQL permite esquemas flexibles que se adaptan con facilidad a los cambios. Restricciones como por ejemplo el tamaño de un campo no son impedimentos.
El estudio se va a desarrollar con datos semiestructurados en donde no siempre se sabe de antemano el tipo o el tamaño del archivo a guardar. Teniendo esto en cuenta se decide avanzar entonces en la investigación de las opciones NoSQL.

2.2 Alternativas a MongoDB

2.2.1 Apache Cassandra

Figura 2.5: Logo de la base de datos NoSQL Cassandra.

Cassandra [Figura 2.5] se define como una base de datos NoSQL de software libre, distribuida y masivamente escalable con la capacidad de escalar linealmente. [2.4, 2.5]. Como se indicó antes pertenece a la categoría de las bases de datos NoSQL de familia de columnas.
Está desarrollada y mantenida por Apache Software Foundation.
Implementa una arquitectura Peer-to-Peer [2.6], lo que elimina los puntos de fallo único y no sigue patrones maestro-esclavo como otros sistemas de almacenamiento. De esta manera cualquiera de los nodos puede tomar el rol de coordinador de una lectura o escritura. Será el driver el que decida qué nodo quiere que sea el coordinador.
Una de sus características es la posibilidad de escalar linealmente. Lo que significa que el rendimiento crece de forma lineal respecto al número de nodos que tenga. Por ejemplo, si con dos nodos se soportan cien mil operaciones por segundo, con cuatro nodos se soportan dos cientos mil. Esto da mucha predictibilidad para la administración del sistemas.
Como la gran mayoría de las bases NoSQL escala de forma horizontal. Lo que significa, que aumenta su capacidad de procesamiento mediante la técnica de sharding.

2.2.2 Apache CouchDB

CouchDB [Figura 2.6] es una base de datos distribuida orientada a documentos, de código abierto y mantenida por Apache igual que Cassandra [2.7].

Figura 2.6: Logo de la base de datos NOSQL Couch.

El proyecto está desarrollado en Erlang, un lenguaje de programación basado en el paradigma funcional, cuyas características se destacan la robustez y el alto rendimiento en el área del paralelismo [2.8].
Para la realización de consultas utiliza un motor orientado a tablas utilizado mediante JavaScript al igual que MongoDB.
Como característica distintiva ofrece ACID [2.9] al igual que las bases de datos relacionales, pero permitiendo siempre la lectura de documentos a través de un control de concurrencia multi-versiones (MVCC, Multiversion concurrency control). Es decir, que las lecturas nunca se bloquean, ya que al iniciarlas se crea una copia del documento que tras ser consultada se destruye, así no hay necesidad de bloquear los elementos de datos mientras se hace la actualización [Figura 2.7].

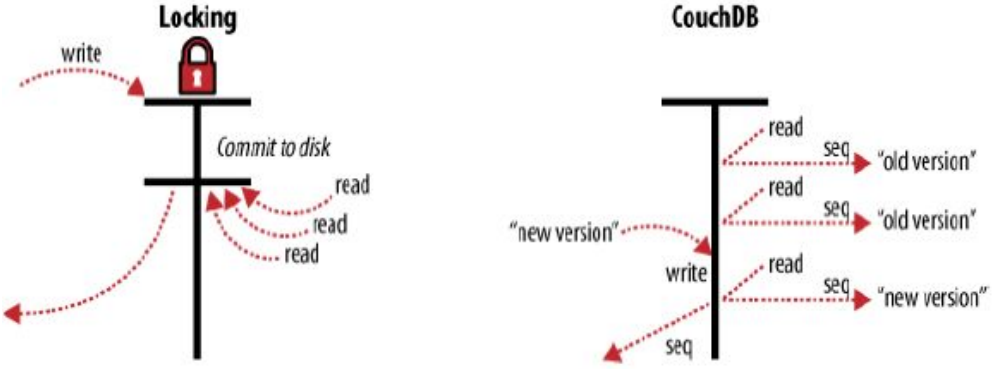


Figura 2.7: Comparación entre sistema de bloqueo y la MVCC de CouchDB.

La arquitectura en CouchDB se basa en pares. Esto permite a los usuarios y servidores acceder y actualizar los mismos datos compartidos mientras están desconectados. Esos cambios pueden luego replicarse bidireccionalmente más tarde.

Los modelos de almacenamiento, visualización y seguridad de documentos están diseñados para trabajar juntos de forma tal que permitan mejorar la eficiencia y la confiabilidad en el manejo de datos en la replicación bidireccional.
Tanto los documentos como los diseños se pueden replicar, permitiendo que las aplicaciones completas de bases de datos (incluyendo diseño de aplicaciones, lógica y datos) se repliquen en computadoras para su uso fuera de línea o se repliquen en servidores en oficinas remotas donde las conexiones son lentas o poco fiables [2.10].
El proceso de replicación es incremental. Una réplica sólo examina los documentos actualizados desde la última sincronización de la réplica. Luego, sobre cada documento actualizado, solo los campos que han cambiado se replican en la red.
Si la replicación falla en algún paso, debido a problemas de red o falla, la próxima replicación se inicia en el mismo documento donde se dejó.
Otra característica de CouchDB es que se pueden crear y mantener réplicas parciales. La replicación se puede filtrar mediante una función de JavaScript, de modo que sólo se repliquen documentos particulares o aquellos que cumplan con criterios específicos. Esto puede permitir a los usuarios tomar subconjuntos de una gran aplicación de base de datos compartida fuera de línea para su propio uso, manteniendo una interacción normal con la aplicación y ese subconjunto de datos.

3. MongoDB

MongoDB [Figura 3.1] es un sistema de base de datos multiplataforma orientado a documentos de código abierto y licenciado como GNU AGPL 3.0 [3.1, 3.2]. Su nombre proviene de la palabra “humongous” cuyo significado se lo puede traducir como enorme o gigantesco.

Figura 3.1: Logo de MongoDB.

El desarrollo de Mongo empezó en el año 2007 por la compañía 10gen, empresa que más tarde cambiaría su nombre a “MongoDB, Inc” [3.3]. Como se mencionó anteriormente es la base de datos de tipo NoSQL más utilizada a octubre de 2017 y es el quinto manejador de base de datos más utilizado a nivel global [3.4].

3.1 Características de MongoDB

3.1.1 Documentos

El manejador de base de datos es de esquema libre. Esto significa que cada entrada o registro puede tener un esquema de datos diferente, con atributos que no tienen por qué repetirse de un registro a otro. A su vez los mismos registros tienen la posibilidad de ir mutando con el tiempo. A cada uno de los registros se los denomina documento.

Los documentos se componen de pares clave-valor. Donde el valor puede contener a su vez, otra estructura anidada de claves-valor. La composición de los documentos son similares a las estructuras de los JSON objects [Figura 3.2].

Dada esta la libertad de esquema, el valor de un campo puede ser otro documento o incluso una colección de N documentos. Un correcto uso de esta técnica permitiría reducir la necesidad de joins al momento de consultar sobre la base de datos.

Figura 3.2: Ejemplo de estructura de un documento de MongoDB.

En MongoDB los documentos son almacenados en formato BSON, o Binary JSON, una versión modificada de JSON orientada a búsquedas rápidas de datos. BSON guarda de forma explícita las longitudes de los campos, los índices de los arrays, y demás metadata útil para el posterior escaneo de datos. De esta forma se introduce un considerable incremento en la velocidad de localización de información dentro de un documento. Sin embargo en la práctica, el cliente nunca ve el formato en que verdaderamente se almacenan los datos. Trabaja siempre sobre un documento en JSON tanto al almacenar como al buscar la información.

3.1.2 Colecciones

Cada registro o documento insertado en la base de datos se lo agrupa en colecciones. Las colecciones se podrían ver como el equivalente a las tablas en una base de datos relacional. La diferencia recae en que las colecciones pueden almacenar documentos con formatos muy diferentes, en lugar de respetar un esquema fijo.
Por otro lado, si bien hay una libertad de esquema MongoDB toma algunos principios de las bases de datos relacionales teniendo en cuenta las posteriores consultas que se harán sobre la base de datos.

3.1.2.1 ObjectId

En el manejo de colecciones se introduce en cada inserción de documentos el campo “_id”. Dicho identificador es inmutable y sirve para referenciar unívocamente a un documento. En general en los esquemas relacionales este campo es numérico y autoincremental. En Mongo en cambio existe el tipo de dato ObjectId.
ObjectId es un tipo especial de MongoDB diseñado para garantizar unicidad en entornos distribuidos [3.5]. A los ojos del cliente el campo “_id” contiene un string que representa un valor hexadecimal. Sin embargo analizando a bajo nivel ese hexadecimal tiene otro significado.
Todo ObjectId está compuesto por doce bytes [Figura 3.3]

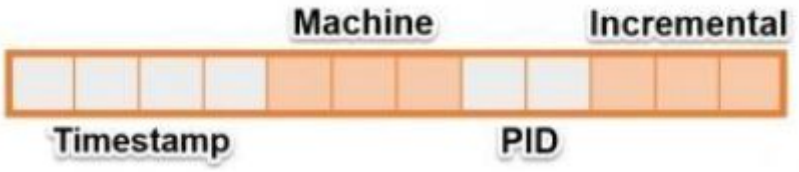


Figura 3.3: Composición de un ObjectId en 12 bytes.

Los primeros cuatro bytes representan un timestamp. Particularmente los segundos transcurridos desde el epoch 0 de unix [3.6]. El tiempo epoch se define como la cantidad de segundos transcurridos desde la medianoche UTC del 1 de enero de 1970 hasta el momento actual. Los siguientes tres bytes representan a la máquina sobre la cual se realiza la inserción. En los entornos distribuidos cada máquina tiene un identificador. Los siguientes dos bytes representan el ID del proceso sobre el que corre el servicio de Mongo. Puede ocurrir en algunas configuraciones que haya más de un servicio corriendo sobre la misma máquina. Para finalizar los últimos tres bytes son un campo autoincremental. Con tres bytes se pueden crear 2^24 números diferentes.
En conclusión, este diseño permite almacenar por cada segundo sobre un proceso que corre en una máquina determinada 16.777.216 inserciones.

3.1.2.2 Índices

Los índices son otro ejemplo de funcionalidades que MongoDB toma de las bases de datos relacionales para mejorar los tiempos de respuesta de consultas.
Los índices son estructuras auxiliares donde se almacenan un campo o una a pequeña porción de campos de los documentos que pertenecen a una colección. Particularmente la estructura que utiliza MongoDB para el manejo de índices es un árbol-B [3.7]. Los árboles-B se caracterizan por mantener los datos en forma ordenada y permitir accesos de lectura, escritura o eliminación en un orden logarítmico sobre la cantidad total de elementos [3.8].
Para mejorar la eficiencia en las búsquedas, los índices se utilizan limitando la cantidad de documentos que serán inspeccionados. Al momento de recorrer una colección se valida si alguno de los parámetros de filtro corresponden con algún índice. En caso de coincidir, la inspección de documentos se reduce de la totalidad de la colección a aquellos cuyo campo o conjunto de campos coincida sobre la estructura del índice.
Existen también subclases de índices. Como por ejemplo los índices de unicidad. Así como en la inserción MongoDB agrega por defecto el campo “_id” también se encarga de crear por defecto un Uniq Index sobre dicho campo. El Uniq Index tiene la particularidad de que además de almacenar en una estructura auxiliar el valor del “_id” valida en cada inserción que el valor ingresado sea único para no guardar valores duplicados.

3.2 Aspectos de bases de datos distribuidas abordados desde MongoDB

Las bases de datos distribuídas suelen permitir fragmentación vertical u horizontal en los datos que alojan. Particularmente en MongoDB se implementa a través de la fragmentación horizontal. El término utilizado para hacer referencia a esta forma de distribución es el Sharding.

3.2.1 Sharding

La fragmentación horizontal en MongoDB consiste en dividir la cantidad total de documentos que componen a una colección, entre los distintos fragmentos disponibles. La fragmentación es horizontal debido a que los datos que componen un registros siempre se mantienen unidos en el contexto de un documento. Para poder dividir los documentos entre todos los shards es necesario tener alguna forma de separar la colección en subconjuntos.

3.2.1.1 Shard Key

La shard key consiste en un campo o un conjunto de campos unívocos e inmutables que existen en todos y cada uno de los documentos de la colección particionada. Para optimizar el manejo de la estructura distribuida, se crea al momento de definir la clave de distribución (shard key) un índice sobre el o los campos que la componen [3.9].
El uso de esta clave permite establecer, en base a sus posibles valores, intervalos que luego serán distribuidos entre los fragmentos o shards disponibles de la estructura. Estos intervalos son conocidos como chunks.

3.2.1.2 Distribución de Chunks

El chunk o pedazo en español, a diferencia de un shard es un fragmento lógico. El cual se basa en la shard key para indicar que todos los documentos cuyo valor unívoco pertenezca a un intervalo definido entre [X;Y] será insertado en ese chunk específico.
MongoDB posee la lógica para autoadministrar los chunks, distribuyendo equitativamente, siempre y cuando sea posible, sobre los shards que tiene disponibles [3.10].
Un chunk tiene un tamaño máximo por defecto de 64MB. Al ocurrir un insert y alcanzar la cota máxima MongoDB ejecuta los siguientes pasos:

1. Divide el chunk a la mitad generando dos chunks de 32MB cada uno.
2. Actualiza el intervalo lógico en base a esta subdivisión. Si antes el intervalo comprendía todos los valores de [X;Y]. Ahora existirán dos intervalos con los valores [X;Z] y [Z+1;Y].

3. Valida la distribución de chunks entre shards. Dado que se generó un nuevo chunk puede ocurrir que las cantidades de chunks en cada shard (nivel físico) están desbalanceadas.
4. En caso de estar efectivamente desbalanceadas se redistribuyen los chunks. Este proceso implica mover uno o más fragmentos lógico entre shards, para luego actualizar la metadata cluster. La ejecución de este último paso se hace de manera transaccional. Lo cual implica en primer lugar que la base de datos no admite inserciones durante esta ejecución. En segundo lugar que no puede quedar en un estado inconsistente. En caso de no poder redistribuir los chunks se debe volver al estado previo.

3.2.2 Sharded Cluster

El Sharded Cluster comprende a todos los actores necesarios para el funcionamiento de la estructura distribuida de MongoDB. Lo que en una posible traducción significa “Conjunto Fragmentado”. Un Sharded Cluster está compuesto por tres actores. Shard, Mongos o Router y Config Server [3.11]. Cada uno de ellos poseen funciones diferentes y corren sobre procesos de mongo totalmente independientes entre sí [Figura 3.4].

3.2.2.1 Shard

Se puede comprender a un shard como el espacio físico en donde se almacena un subconjunto de los documentos que componen a una colección.

3.2.2.2 Config Server

Encargado de almacenar y conocer la configuración y el estado actual de cada uno de los componentes que pertenecen al sharded cluster. Datos como por ejemplo los distintos intervalos definidos sobre el shard key y a qué shard del cluster pertenece.
Toda esta información adicional y necesaria para orquestar el funcionamiento del Sharded Cluster se la denomina como metadata.

3.2.2.3 Mongos o Router

Actua como un ruteador o direccionador de consultas. Provee una interfaz entre los pedidos de las aplicaciones del cliente y la estructura distribuida. Se lo puede entender también como el punto de entrada al cluster.
Las instancias de Mongos mantienen en memoria gran parte de la información del Config Server. Información que luego la utilizan para direccionar operaciones de lectura o escritura al shard correspondiente.

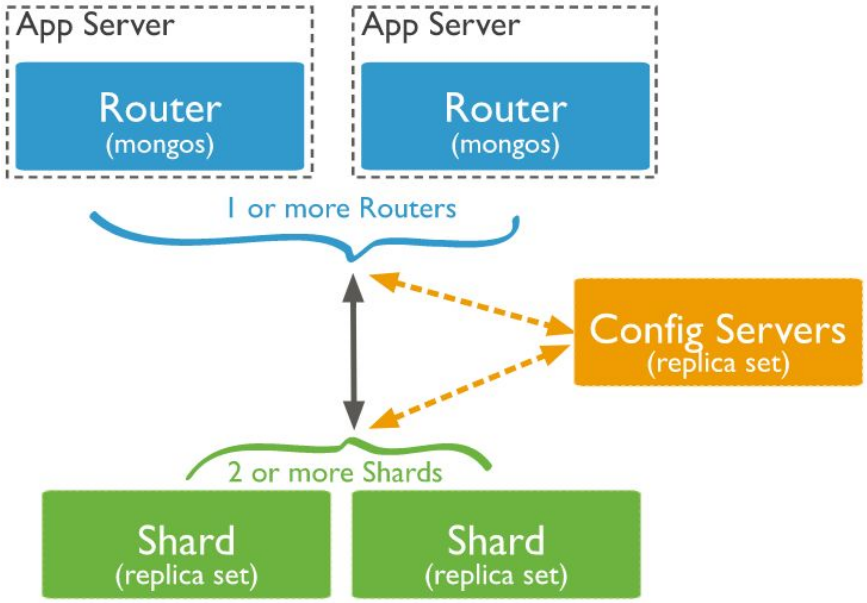


Figura 3.4: Representación gráfica de los componentes de un Sharded Cluster en MongoDB.

Para entender cómo se relacionan todos estos componentes de manera práctica, se puede analizar el comportamiento del cluster frente a una instrucción de inserción.
Suponiendo que una aplicación le indica al sharded cluster que debe insertar un registro compuesto por un determinado JSON.
El punto de entrada del comando de inserción es a través del Mongos. El servicio se encarga de revisar en base a la metadata que tiene en caché y los datos del registro a insertar, a qué shard corresponde enviarle la inserción. Acto seguido el shard elegido recibe el comando, lo ejecuta y le responde al Mongos el resultado de la inserción. Resultado que luego es redirigido a la aplicación que dió inicio a la ejecución.
Es posible que luego de la inserción haya un desbalance en la cantidad de datos que almacena cada shard y sea necesario un reacomodamiento de los chunks. Si fuera necesario modificar los intervalos de los shards y redistribuir los documentos en la estructura, se actualizará el Config Server. Ya que es aquí donde reside la metadata con esta información. En caso de actualizarse correctamente, los Mongos actualizarán también los registros de su caché para poder calcular correctamente futuras queries.

3.2.3 Replica Set

Otra característica de los sistemas distribuidos son las réplicas. Al tener la información fragmentada las bases de datos distribuidas se pueden volver inaccesibles con la imposibilidad de leer o escribir en una simple porción de los

datos. Dividir una gran cantidad de información en volúmenes manejables para instancias de menores recursos puede mejorar el rendimiento. Pero de poco servirá si alguno de esos recursos termina bloqueando la estructura distribuida.
La solución de MongoDB a este problema se basa en la redundancia de la información. La redundancia plantea que cuantas más réplicas existan de una porción de datos más disminuirá la posibilidad de que sea inaccesible. Ya que si bien pueden existir fallas, es muy probable que alguna de todas las réplicas pueda responder en representación del conjunto consultado. Se conoce a esta funcionalidad en MongoDB como Replica Set [3.12].
Los Replica Set o conjuntos de réplicas son un grupo de procesos mongo que mantienen el mismo conjunto de datos. El objetivo es proveer redundancia de la información y por consiguiente alta disponibilidad. Con múltiples copias de datos alojadas en distintos servidores se incrementa el nivel de tolerancia a fallos en comparación con la posible pérdida de acceso a un simple servidor.

Un Replica set está compuesto siempre por un nodo primario y N nodos secundarios. Opcionalmente en base a la configuración puede existir un nodo de tipo árbitro [Figura 3.5, 3.6].

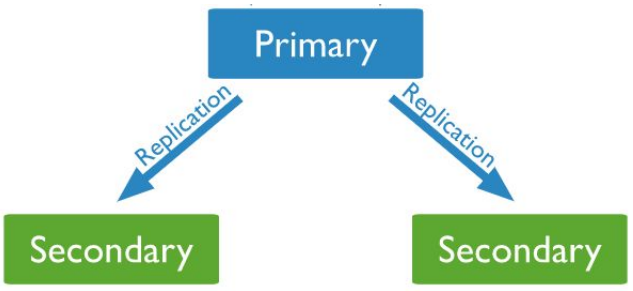


Figura 3.5: Organización de un Replica Set compuesto de nodos primarios y secundarios.

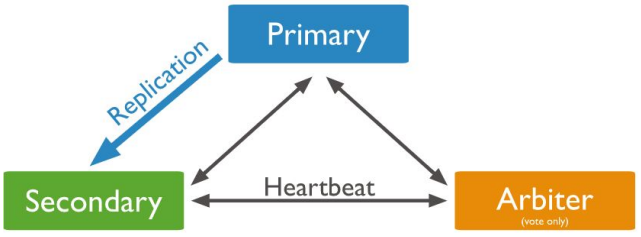


Figura 3.6: Organización de un Replica Set compuesto de nodos primarios y secundarios y un árbitro.

3.2.3.1 Nodo Primario

El nodo primario es el único nodo capaz de recibir y responder a las operaciones de escritura. En un Replica Set solo puede existir un nodo primario al mismo tiempo. En consecuencia el nodo primario es aquel que siempre tiene el último estado de su conjunto de datos.
Es responsabilidad del nodo primario registrar en una estructura alterna todos los cambios en el orden preciso en que fueron surgiendo. Se denomina a este log de operaciones como oplog.

3.2.3.2 Nodos secundarios

Los nodos secundarios se encargan de ir copiando el oplog generado en el nodo primario. En base a este registro de operaciones se ejecutan las instrucciones en segundo plano hasta llegar a reflejar los mismos datos que el nodo primario.
En un Replica Set por defecto las operaciones de lectura también son dirigidas al nodo primario. Pero a diferencia de las operaciones de escritura, pueden distribuirse entre nodos primarios y secundarios. Lo cual permite potenciar el nivel de respuesta en la lectura. Sin embargo, hay que tener en cuenta que si la respuesta la genera un nodo secundario que aún no terminó de sincronizarse con el primario, la lectura puede no coincidir con la información correspondiente.
La otra característica de los nodos secundarios es que en caso de que el nodo primario se vuelva inaccesible en la red, pueden cambiar su rol en el Replica Set convirtiéndose en el primario.
Cuando un nodo primario es inaccesible, se bloquean todas las operaciones de escritura y comienza a ejecutarse un protocolo de elección de nuevo primario. Un nodo se encarga de “llamar a votar” a todos los nodos habilitados. (Por defecto cada voto vale un punto, pero se puede customizar con el fin de darle mayor prioridad a algún nodo específico). El protocolo consiste en ir registrando los votos, como en cualquier elección, y el que tenga mayor puntaje es designado primario. Se debe tener en cuenta que existe la posibilidad de que en la votación se genere un empate. Para romper con los posibles empates existe el nodo de tipo árbitro.

3.2.3.3 Árbitro

En caso de tener una cantidad par de nodos habilitados para votar por un nuevo primario, siempre existirá la posibilidad de un empate. La función del nodo árbitro es desempatar elecciones.
A diferencia de los nodos primarios y secundarios, el árbitro es un proceso con los recursos mínimos e indispensables para cumplir con la función de desempatar. No posee una copia de los datos del primario. Por lo tanto, el nodo árbitro nunca podrá cambiar su rol dentro de un Replica Set como lo hacen los otros nodos. El único momento que es considerado como un par entre los nodos de la réplica es al

momento de la elección. Cabe destacar que los árbitros sólo pueden votar una única vez por elección

Para que funcione una estructura de Replica Set todos los nodos deben estar debidamente identificados, pero también es necesario que cada instancia conozca la topología y el estado actual de la red, ya sea para actualizar la información o para actuar frente a fallos. En el **[Apéndice II]** se explica en detalle cómo es la interacción entre todos los nodos para mantener funcionando el ecosistema de réplicas.

4. Sucesión de experimentos, validaciones y análisis de resultados

Se plantea una sucesión de experimentos con varios objetivos, algunos de ellos casi directamente relacionados con el trabajo de tesis en particular. En principio, se podría ver la sucesión de experimentos como una migración de un sistema centralizado y quizás clásico de SQL a un sistema de bases de datos NoSQL con replicación y distribución/escalado horizontal/fragmentación. Los términos distribución, escalado horizontal y fragmentación son usados aquí como sinónimos aunque en realidad identifican diferentes aspectos de las bases de datos NoSQL en general y MongoDB en particular.
La secuencia de experimentos tiene como objetivos poder configurar en un entorno como MongoDB distintas opciones o estructuras posibles de una base de datos, en donde posteriormente se permita evaluar distintas métricas sobre el rendimiento. Particularmente se evalúan tres aspectos sobre cada estructura: el tiempo de ejecución en cuanto a la inserción (cantidad masiva de inserciones, para ser más precisos), la consistencia de los datos guardados y los tiempos de respuesta para recuperar la información.

El experimento será repetido sobre cuatro configuraciones diferentes de MongoDB. Teniendo como punto de partida una estructura centralizada y finalizando en una estructura distribuida.

4.1 Esquema centralizado

Consiste en un único servidor con MongoDB que se encarga de recibir y administrar todas las lecturas y escrituras que un cliente puede hacer sobre una base de datos.

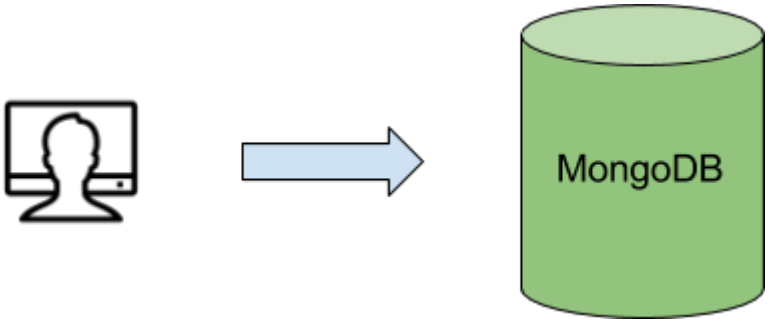


Figura 4.1: Representación de esquema centralizado.

En una base de datos centralizada, se almacena la totalidad de la base de datos en un único lugar físico [Figura 4.1]. Sería como el equivalente con MongoDB de una

base de datos SQL centralizada. Es decir que no cambiaría mucho más que el manejador de la base de datos, con todo lo que eso implica por supuesto (de una base de datos SQL a una NoSQL, en particular).

4.2 Esquema centralizado con réplicas

Al igual que en el centralizado hay un servidor primario, encargado de recibir y administrar las peticiones a la base de datos, pero al mismo tiempo hay otras dos instancias secundarias con una copia completa del nodo primario que se va actualizando asincrónicamente a medida que el nodo primario es modificado

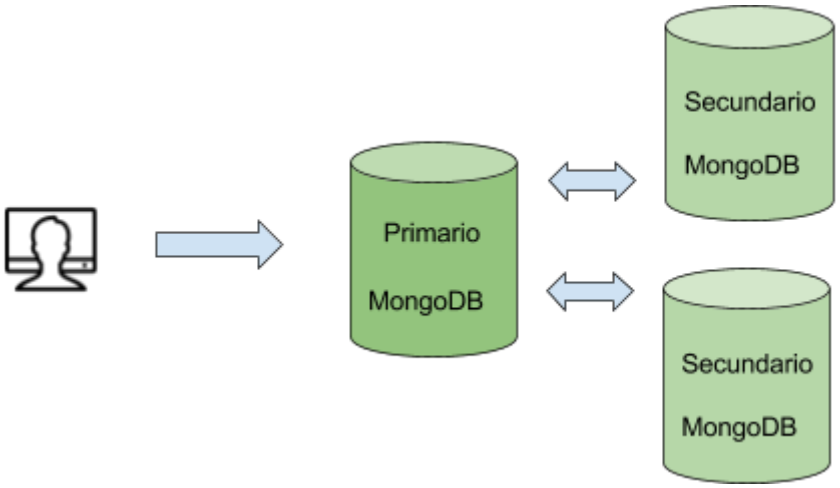


Figura 4.2: Representación de esquema centralizado con réplicas.

A diferencia de un esquema centralizado puro, se almacena la totalidad de la base de datos en tres lugares físicos diferentes. Permitiendo que frente a algún fallo del nodo primario exista un nodo secundario capaz de ocupar su rol frente a las futuras peticiones [Figura 4.2].

4.3 Esquema Distribuido sobre 2 shards con réplicas

Lo que en sistemas de bases de datos distribuidas, usualmente se denomina fragmentación en MongoDB se lo llama sharding. En este caso el esquema distribuido consiste en dividir físicamente la información almacenada en la base de datos en 2 partes y a su vez replicar la información de cada parte o shard como se hizo previamente con el esquema centralizado con réplicas [Figura 4.3].

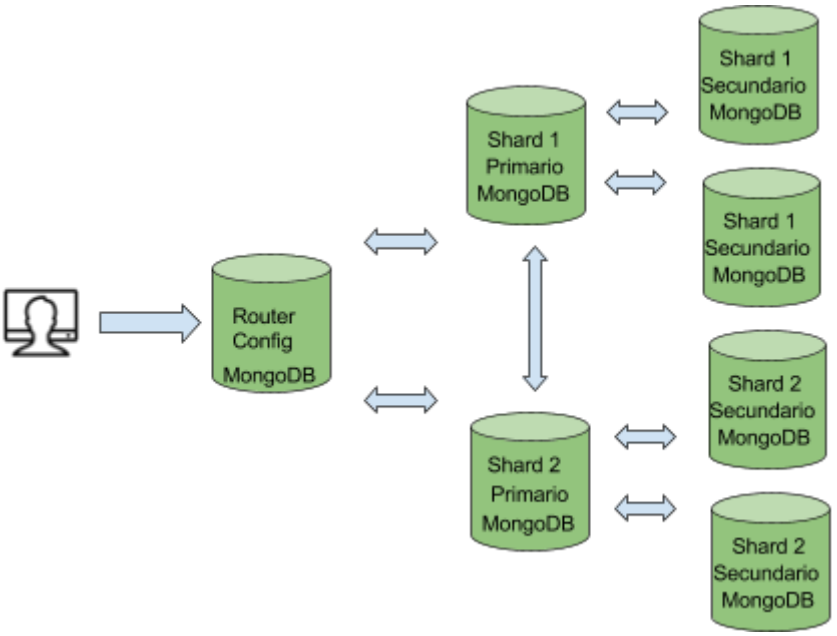


Figura 4.3: Representación de esquema distribuido con réplicas.

En los esquemas distribuidos, además de los shards que almacenan los datos que constituyen propiamente a la base, son necesarios otros dos actores para mediar entre las peticiones que puede hacer un cliente y la estructura distribuida.
El router es un servicio que se encarga de procesar las consultas de la capa del cliente y determinar la ubicación de la información necesaria para completar la operación. Por otro lado, el config guarda la metadata de la estructura distribuida. La metadata refleja el estado y la organización de todos los datos y componentes que hay en el cluster. Sobre esta información se apoya el servicio de router para enviar las operaciones a los shards correspondientes.

En entornos de producción, se recomienda separar en distintas instancias de Mongo el Router y el Config y a su vez tener la metadata del Config replicada tal como se muestra con los nodos primarios y secundarios de cada shard.
Sin embargo, dado que los distintos experimentos se realizan en un entorno controlado, como lo es el cluster de la facultad, y no es un objetivo agregar a los resultados la latencia que pueda haber entre Router y Config, se opta por unificar estos roles en un mismo servidor, como se refleja en la [Figura 4.3].

4.4 Esquema Distribuido sobre 3 shards con réplicas

Igual al esquema anterior pero con un shard más a la hora de fragmentar los datos [Figura 4.4].

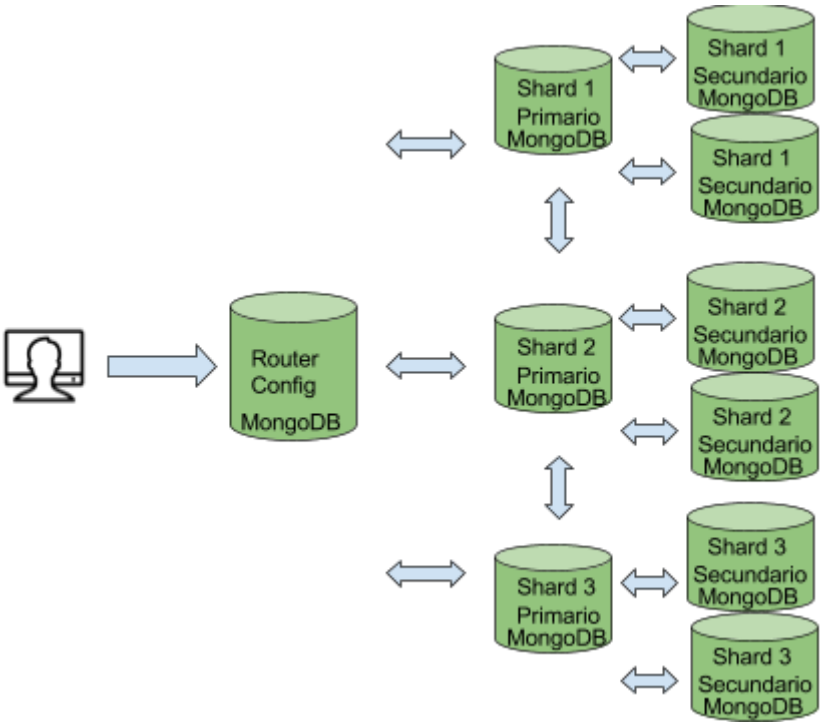


Figura 4.4: Representación de esquema distribuido con réplicas.

Una de las posibilidades que permite analizar el sharding sobre MongoDB es la escalabilidad horizontal. Agregando componentes se divide el total de la información en porciones menores sobre los distintos shards. Agregando un shard se redistribuyen los datos y a su vez se distribuye el procesamiento.

4.5 Carga de datos

Para evaluar tiempos de escritura, lectura e integridad de la información es necesario tener una carga considerable de datos para poder obtener resultados con un grado de certeza aceptable. La carga de datos para hacer las evaluaciones de las estructuras descriptas está conformada por una colección de archivos de distinto tipo y tamaño.

Debido a que MongoDB tiene una funcionalidad llamada “GridFS” [4.1] exclusivamente orientada a guardar y recuperar de manera correcta y eficiente archivos mayores a 16 MB; Y por otro lado en la documentación oficial se indica cargar el binario de todos los archivos de menor tamaño (sin hacer uso de dicha herramienta) se tendrá como condición que ningún archivo puede ser mayor a 16 MB. El objetivo es poder recopilar resultados que no sean influenciados por funcionalidades que potencian algún aspecto de la estructura distribuida por sobre la centralizada, más allá de la naturaleza inherente de cada estructura.

Utilizando los campos que usa el protocolo de http para enviar un archivo a un servidor [4.2] se define la estructura [Figura 4.5] que tiene la colección de “archivos” sobre la que se cargarán todos los datos. Como se aprecia en la estructura descrita, además de la metadata usual de cada archivo se sumará un hash y el binario del archivo.

```
File Structure
{
    Id,
    filename,
    name,
    extension,
    mime_type,
    length (bytes),
    data (binary),
    xxhash,
    upload_date
}
```

Figura 4.5: Estructura del archivo a guardar.

La técnica del hashing, se basa en tomar una entrada producir una cadena de longitud fija con la particularidad de que la misma entrada producirá siempre la misma salida y cualquier cambio sobre dicha entrada generará un cambio sobre el hash. El hashing sirve al propósito de garantizar la integridad, es decir, si algo cambió del binario del archivo, podrá ser detectado [4.3].
Por otro lado, para guardar el binario de un archivo MongoDB contempla un tipo “Binary Data” usualmente llamado binData [4.4].

4.6 Evaluar el tiempo de respuesta

Para evaluar los tiempos de respuestas sobre las distintas arquitecturas se han implementado consultas que podrían considerarse como los peores casos teniendo en cuenta que para obtener un resultado el manejador de base de datos necesitará recorrer todos los documentos de la colección. Las consultas deben ser de orden N. Lo cual implica que el motor de base de datos debe recorrer todos los registros contenidos para dar una respuesta.
Para poder validar la coherencia de las estadísticas recopiladas sobre la consulta de O(n) hay que realizar distintas consultas que tengan el mismo orden de magnitud. La comparación en los tiempos de respuesta de estas consultas, dentro de una misma arquitectura y sobre una misma cantidad de datos debería ser similar.

4.7 Elección de clave en estructuras distribuidas

Al momento de diseñar un esquema distribuido, se debe elegir la clave de distribución o también llamada, dentro del entorno de Mongo, “Shard Key” [4.5]. La clave determina la distribución de los documentos entre los fragmentos del cluster. El o los campos elegidos como clave, deben ser obligatorios en todos los registros que se quieran insertar sobre la colección. Esto se debe a que se utiliza dicha clave para decidir en qué shard o nodo se terminará almacenando el registro.
Sobre una colección particionada, Mongo arma separaciones lógicas con intervalos de valores en donde va insertando los documentos en base a su clave. A su vez, estos intervalos lógicos están distribuidos sobre las distintas particiones físicas.
En concordancia con la elección del destino del registro, Mongo agrega un índice sobre la clave definida. Dado que se usa el campo indexado para elegir su destino, también se lo utiliza para una rápida respuesta en caso de consulta.

En los experimentos se utiliza como clave de distribución el campo “id” generado por Mongo. Cuando se selecciona una clave de distribución, es útil tener una perspectiva de los resultados que se necesitan recuperar de la base de datos en un futuro. La naturaleza de los experimentos planteados apunta a contrastar resultados entre estructuras centralizadas y distribuidas. En caso de optar por una shard key que involucre algún otro campo podría impactar sobre las conclusiones finales. Las consultas a realizar implican recorrer todos los documentos de la colección. Si la clave incluyera otro campo además del id, las consultas sobre las estructuras distribuidas tendrían claras ventajas sobre las centralizadas. No necesariamente por su estructura, sino porque, como se aclaró previamente, se genera un índice sobre la clave elegida [4.6]. Con lo cual la comparación se resumiría a consultas sin índice y con índice.

5. Implementación

La implementación de toda la investigación fue realizada empezando con pequeñas funcionalidades sencillas que tornaron hacia desarrollos más complejos. Los experimentos y configuraciones iniciales se definen de manera suficientemente acotada como para tener verificaciones específicas. A partir de los resultados obtenidos con los primeros experimentos se definirán pruebas/escenarios más elaborados y quizás también más significativos o representativos de los sistemas en producción.

5.1 Estructuras de prueba en un entorno local

El primer objetivo para llevar a cabo el estudio sobre el rendimiento, consiste en instalar, configurar y hacer pruebas sobre las estructuras previamente planteadas con MongoDB. Durante este proceso de aprendizaje, se genera un reporte técnico **[Apéndice III]** documentando cada uno de los pasos necesarios para llegar a una instalación productiva de una estructura distribuida.
Para tal tarea se utilizan máquinas virtuales con Lubuntu [5.2], una versión liviana (que requiere relativamente pocos recursos) de Ubuntu, y distintas configuraciones-instancias de MongoDB. Con el ambiente ya configurado y funcionando se realizan las pruebas locales que incluyen el desarrollo de las funcionalidades necesarias para enfocar el estudio en el rendimiento de MongoDB. Todo el software desarrollado fue sobre el lenguaje ruby [5.3] y su código está disponible en un repositorio público de Github [5.4].

5.2 Desarrollo de las funcionalidades

5.2.1 Inserción de archivos

La inserción consta de los siguientes pasos:

1. Dado un path o ruta específica, se deben buscar recurrentemente todos los archivos, cualquiera sea su extensión, con tamaño menor a 16 Megabytes.
2. Dado un archivo, se deben extraer sus metadatos. Por ejemplo: Nombre, extensión, mime type, etc.
3. Dado un archivo, se debe obtener el binario que lo compone.
4. Sobre la composición del binario, generar un hash para futuras validaciones de integridad.
5. Unir todos los datos en un documento json, para luego realizar la inserción sobre la colección de MongoDB.
6. Devolver el documento insertado en caso de éxito o levantar una excepción informando lo ocurrido en caso contrario.

5.2.2 Validación de consistencia

Para validar la consistencia es utilizado el hash generado en el punto 4 de la inserción. Al recuperar el archivo insertado y el algoritmo genera un hash con el binario como entrada y lo compara con el hash previamente guardado en la colección de archivos. Si coinciden, se puede asegurar que Mongo devolvió exactamente el mismo archivo almacenado.
Existen muchos algoritmos de hashing, en este caso se utiliza xxHash. xxHash es un algoritmo no criptográfico (no genera cambios sobre la entrada) que se caracteriza por su rapidez [5.5]. Al mismo tiempo, completa el conjunto de pruebas que evalúa las cualidades de colisión, dispersión y aleatoriedad de las funciones hash (SMHasher [5.6]). Esto asegura que el uso de su función genere hashes idénticos en todas las plataformas donde se los ejecute.

5.2.3 Consultas para evaluar el tiempo de respuesta

Las consultas desarrolladas implican leer todos los registros de la colección para obtener una respuesta final. El objetivo de estas búsquedas lineales es repetirlas a lo largo de las distintas arquitecturas planteadas de MongoDB para obtener su tiempo de respuesta. Luego en base a la comparación de los tiempos se podrán obtener conclusiones. El desarrollo consistió en las siguientes consultas:

1. Obtener todos los archivos que son de un tipo particular

Para recuperar todos los archivos de un tipo en particular, por ejemplo: “.js” se utiliza la siguiente consulta [Figura 5.1]

```
DB.collection('archivos').find( { extension: random_extension } ).count
```

Figura 5.1: Consulta para obtener todos los archivos con una extensión determinada.

DB representa la conexión a la base de datos a través del driver de Mongo. Luego se selecciona la colección sobre la que se quiere buscar, en este caso “archivos”. Finalmente se define el filtro deseado como parámetro sobre la función de find().

2. Obtener un archivo específico por nombre

La consulta es muy similar a la anterior, únicamente cambia el campo de filtro [Figura 5.2]

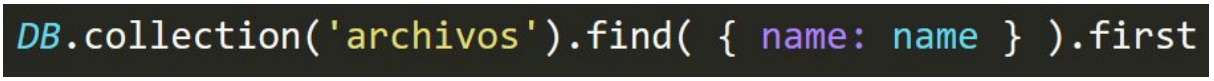


Figura 5.2: Consulta para obtener todos los archivos con un nombre determinado

En las dos consultas anteriores se recibe por parámetro el valor a buscar. Estos son "`random_extension`" y "`name`". En ambas consultas se tiene como condición recuperar al menos un resultado para poder validar que funcionó correctamente. Por lo tanto se debe tener la seguridad de que los parámetros enviados pertenezcan a algún archivo previamente insertado. Con lo cual, todos los nombres y extensiones insertados en la base de datos, son posteriormente guardados en una estructura auxiliar, una colección de ruby. Al momento de ejecutar las consultas, un elemento aleatorio de cada colección es removido y utilizado como parámetro de las funciones. Remover el elemento es relevante para las consultas posteriores, ya que puede ocurrir que Mongo guarde en memoria el resultado de las últimas consultas, lo que a posteriori puede implicar una distorsión sobre los tiempos relevados.

3. Obtener todos los archivos mayores a X bytes

La primera parte de esta consulta se mantiene igual a las anteriores [Figura 5.3].

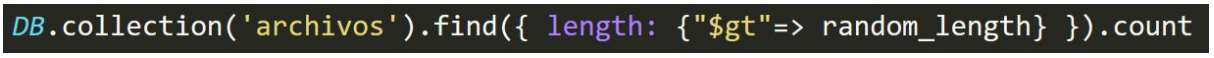


Figura 5.3: Consulta para obtener todos los archivos mayores a una cantidad de bytes.

En la sección de filtrado, a diferencia de las consultas anteriores no hay una estructura auxiliar con los tamaños de los archivos insertados. Se genera una cantidad de bytes aleatoria para setear la función. Por otro lado, se utiliza el operador "`$gt`", lo que representa el filtro "greater than" o más conocido en las bases de datos relacionales como mayor que ">".

4. Obtener todos los archivos que pertenezcan a un shard determinado

Esta consulta solo es aplicable a las últimas dos estructuras. Estructuras que necesariamente implementan sharding.
La dificultad de la consulta radica en que dada la configuración de la distribución a evaluar, no se puede saber de manera anticipada la ubicación de todos los archivos de un shard. Para poder obtener el resultado correcto es necesario tener de antemano en una estructura externa todos los ids que pertenecen a un shard determinado. Por otro lado, la elección del shard debe ser aleatoria, para no tener siempre la respuesta del mismo nodo.
Para un cliente la manera de interactuar con una estructura distribuida de MongoDB es a través del router (proceso encargado de interactuar con el config server, los

shards y las peticiones externas). Sin embargo, a cada shard se lo puede entender como una instancia completa. Su única diferencia con esquema centralizado, es que almacena únicamente una porción de la información total. Por lo tanto es posible conectarse a un único shard y hacer consultas sobre el rango de información que almacena.
El pre-procesamiento antes de resolver la consulta, implica conectarse a un shard, obtener el id de todos los registros almacenados y guardarlos en archivo externo [Figura 5.4]. Archivo que luego será leído y parseado para armar la consulta final apuntada a la estructura distribuida.

```
def medoto_para_la_obtencion_de_todos_los_ids_de_un_shard
  #Conenxion random a un shard
  1) Se crea la conexión a un Shard particular de la arquitectura distribuida
     (elegido de manera aleatoria)

  2) Se crea un archivo vacio 'coleccion_de_ids.txt'

  3) Se recorre la coleccion de elementos insertando al final
     del archivo creado previamente el valor del campo "id"

  4) Se cierra la conexión creada en el punto 1

  5) Se crea la conexión a la estructura distribuida (el router de MongoDB)

  6) Se construye la query en base a los ids recopilados en el punto 3

       DB.collection('archivos').find('_id':
         {'$in': [COLECCION_CON_LOS_IDS_GUARDADOS_EN_EL_ARCHIVO]).count

  7) Se borra el archivo 'coleccion_de_ids.txt' para futuras ejecuciones
end
```

Figura 5.4: Algoritmo para obtener todos los archivos que pertenece a un shard determinado.

Como se observa en la [Figura 5.5] a diferencia de las consultas previas, el filtro está compuesto por el operador **"$in"** el cual recibe un arreglo de ids. Arreglo conformado por el análisis sintáctico del archivo previamente generado y posterior transformación del id en formato string a ObjectId. ObjectId es el hexadecimal que utiliza mongo por defecto como clave primaria. Por último, resta almacenar el tiempo de ejecución que le llevó a la base de datos responder la consulta.

```
DB.collection('archivos').find('_id':
  {'$in': [COLECCION_CON_LOS_IDS_GUARDADOS_EN_EL_ARCHIVO]} ).count
```

Figura 5.5: Consulta para obtener todos los archivos que pertenece a un shard determinado.

5.2.4 Registrar los tiempos de ejecución

Para registrar los tiempos de respuesta en consultas de lectura Mongo cuenta con una función nativa ".explain()" [5.7] que provee información de la lectura ejecutada. Sin embargo esta función es solo aplicable a las instrucciones de búsqueda. No retorna el tiempo de ejecución para la escritura. Dado que se procura tener una funcionalidad lo suficientemente genérica que pueda retornar el tiempo indistintamente de la naturaleza de la instrucción, se debe implementar una solución un poco más compleja.

MongoDB provee la posibilidad de crear una conexión con la base de datos en modo debug. Si se indica un path al archivo de logs, se delega en el manejador de base de datos la responsabilidad de escribir al final del archivo la última instrucción ejecutada y su tiempo de ejecución en milisegundos. Con lo cual, el desafío se simplifica en programar un script con la siguiente funcionalidad [Figura 5.6]:

1. Se debe ejecutar siempre e inmediatamente después de que se ejecute una instrucción en la base de datos.
2. Se debe obtener la última línea del archivo de logs.
3. Se debe poder interpretar esta última línea de manera tal que recupere sólamente los milisegundos de la instrucción.
4. Se debe almacenar el tiempo de la ejecución de la instrucción obtenido en el punto 3 para su posterior análisis.

```
#time in Miliseconds
def get_last_query_time
  ((`tail -n 1 mongo.log`).split("|").last.strip.gsub("s","").to_f * 1000.0).to_i
end
```

Figura 5.6: Función para obtener el tiempo de ejecución de la última instrucción ejecutada sobre la base de datos.

5.2.5 Registrar el tamaño de la base de datos

Al momento de guardar el tiempo de ejecución, surge la necesidad de guardar el tamaño de la base de datos. Tener el dato de los milisegundos de una consulta, no representa un dato relevante sin la información del contexto. Por ello, los registros del archivo donde se guardan los datos de rendimiento, están compuestos por dos campos. Tiempo en milisegundos y tamaño de la base de datos en Megabytes.

Para obtener el tamaño, de una colección Mongo cuenta con la función "`storageSize()`" que devuelve la cantidad total de almacenamiento asignado a una colección. La función para recuperar el tamaño almacenado de una colección es la siguiente [Figura 5.7].

En la documentación oficial de MongoDB se describe que la función StorageSize() devuelve en bytes el total del espacio alocado de una colección [5.8]. Por lo tanto, como se puede observar sobre el resultado final, se divide el número de bytes para realizar el pasaje a MB.

```
#storageSize in MB
def total_storage
  (DB.command({collStats:'archivos'}).documents.first["storageSize"] / 1_048_576.0).round
end
```

Figura 5.7: Función para obtener el tamaño de una colección determinada de MongoDB.

Por otro lado, se debe destacar que la práctica indica que el resultado de esta función no es un dato del todo completo. El tamaño que retorna dicha función es en general una tercera parte de lo que realmente se almacena en disco. Se llega a esta conclusión luego de repetidas pruebas en el entorno local que luego sobre la implementación de las arquitecturas, ya sean centralizadas o distribuidas, daban el mismo resultado.
Cuando se realiza la misma consulta desde el filesystem sobre el path del sistema operativo que MongoDB persiste sus datos, retorna un tamaño similar a la colección de datos previa, utilizada para la inserción de archivos. Unas 2 veces más de lo que devuelve la función de Mongo. A pesar de esto, es notable que en todas las evaluaciones existe cierta reciprocidad de 1 a 3, entre lo que devuelve "`storageSize()`" y lo que se está almacenado a nivel filesystem. Se llega entonces a la conclusión de que en realidad la función interna de MongoDB está evaluando una parte del total de la información.

5.3 Periodicidad en la ejecución de consultas

Para evaluar cada una de las estructuras antes planteadas, se definió insertar alrededor de 60 GB de datos distribuidos entre diferentes archivos. Como se describió en la sección [5.2.5] desde las funciones que nos provee MongoDB para consultar el tamaño de la base, el resultado será un tercio del tamaño total a nivel filesystem. Esto quiere decir que en el gráfico final los 60 GB van a ser equivalentes a 20 GB.
Los tiempos de ejecución a evaluar se basan en consultas de lectura y escritura. Como se busca analizar el rendimiento de dichas instrucciones se plantea tener dos parámetros configurables para salvaguardar el tiempo de ejecución sobre cada tipo de instrucción.
En términos de escritura se registrará cada 200 MB de datos insertados, el tiempo que conlleva escribir un archivo en base de datos. Con lo cual, a lo largo de 20 GB se habrán recolectados unos 100 registros diferentes. Por el lado de las consultas o lecturas, los intervalos son más pronunciados y variados si comparamos la cantidad

de datos almacenados y el momento de su ejecución. Las consultas serán realizadas cuando se hayan almacenado la siguiente cantidad de datos:

- 1 GB
- 2.5 GB
- 5 GB
- 10 GB
- 20 GB

5.4 Unificar funcionalidades en una única interfaz

Una vez terminado y testeado individualmente el desarrollo de las funcionalidades para la evaluación de las estructuras, se unifican los distintos scripts sobre una única interfaz. El punto de partida es generar una aplicación que unifique las funcionalidades y que pueda mostrar de una manera sencilla para cualquier usuario todo lo desarrollado a nivel código.

Sobre este objetivo se creó una aplicación web. El desarrollo está basado sobre Cuba [5.9], un micro framework desarrollado en ruby. En cuanto al diseño, se busca una interfaz simple cuyo diseño se basa en bootstrap [5.10]. La aplicación consta de dos pantallas, una para la carga de archivos y otra para la visualización de los resultados.

En la pantalla de “Cargar archivos” [Figura 5.8] el usuario debe ingresar un path válido para luego iniciar el proceso desde el submit del formulario. A nivel código, el servidor toma el path, valida su existencia y busca de manera recursiva sobre el directorio todos los archivos menores a 16 Megabytes, cualquiera sea su extensión. Luego empieza a cargarlos en la base de datos mientras valida la integración de los datos guardados. Tal como se describió en la funcionalidad de inserción y validación de la integridad. Una vez terminado el proceso, retorna a la misma pantalla informando la cantidad de archivos procesados y cargados.

Por otro lado en el log del servidor, se va escribiendo el nombre y la ruta de cada archivo procesado. Esta última funcionalidad, nació de la necesidad de obtener información sobre el estado del proceso de carga, ya que para las pruebas finales el proceso podía durar alrededor de cuatro horas.

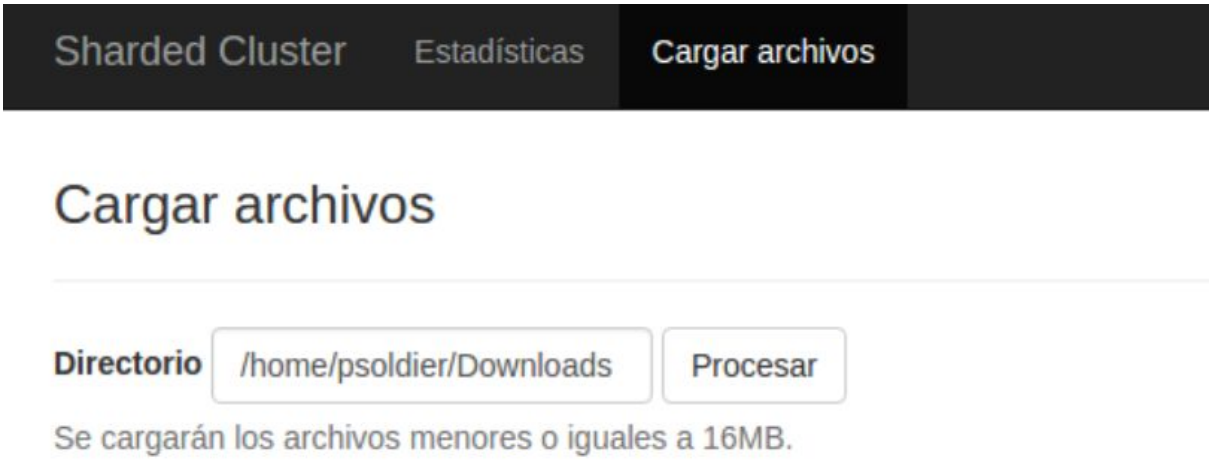


Figura 5.8: Pantalla inicial del sistema para la carga de archivos.

La pantalla de estadísticas transforma toda la información recolectada de los experimentos en un gráfico de simple lectura. Los datos que está mostrando la [Figura 5.9] son los correspondientes al entorno local obtenidos durante la etapa de desarrollo de la aplicación. En la sección seis se verán los resultados concretos de los estudios.
Usando una librería de javascript “chart.js” [5.11] se representa sobre el eje “X” la cantidad de datos almacenados en Megabytes al momento de ejecutar la búsqueda y sobre el eje “Y” el tiempo que transcurrió para procesar la instrucción en milisegundos.

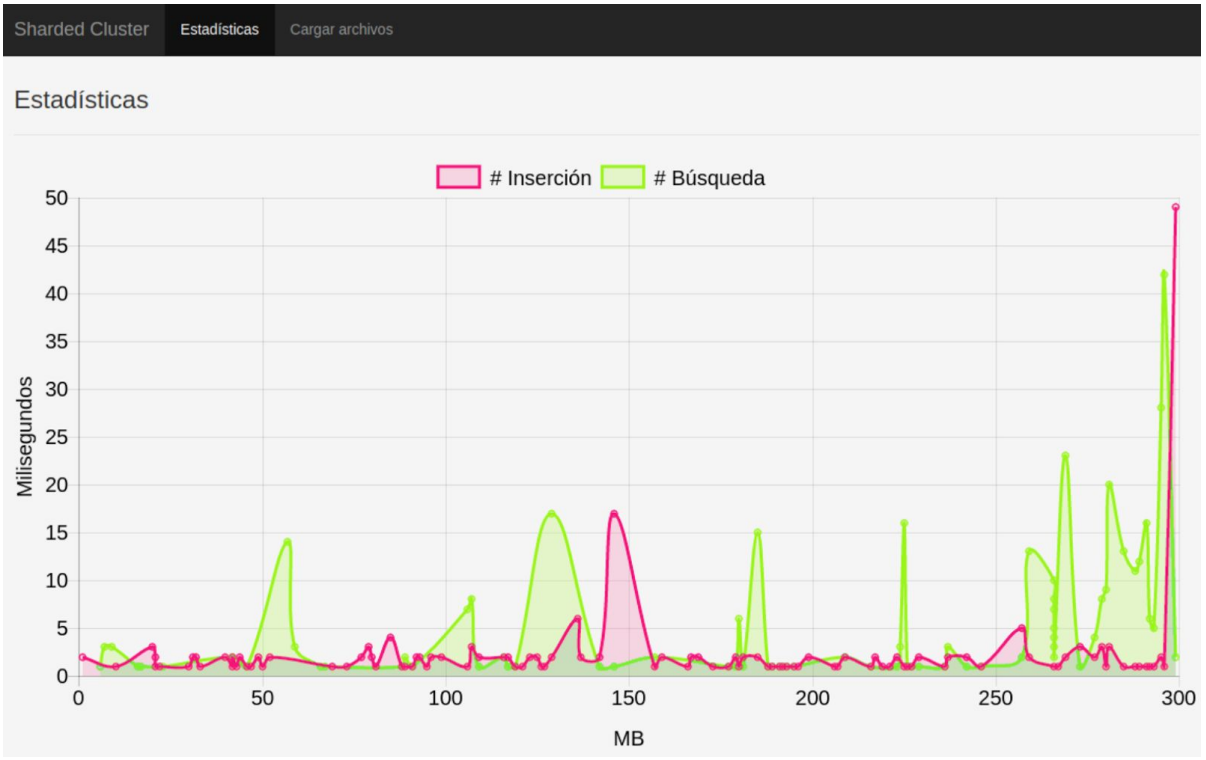


Figura 5.9: Pantalla de estadísticas. Los resultados de este gráfico corresponden a la etapa de desarrollo del sistema.

5.5 Recopilación de datos para los estudios

Una vez que el desarrollo de las funcionalidades a evaluar está unificado y a su vez está armado el instructivo para la configuración de las distintas estructuras, queda un último paso para realizar el estudio. La colección de archivos que se utilizarán para la carga de la base de datos.
El desafío consiste en tener una colección de archivos de muchos tipos y muchos tamaños diferentes (con la salvedad de que sean menores a 16 MB). Por otro lado debido a los puntos a evaluar, se concluye que es necesario obtener unos 60 Gigabytes de archivos.
La primera alternativa fue descargar los códigos fuentes de sistemas operativos, documentación y demás software libre que en principio su tamaño estaba en el orden de varios gigabytes. Sin embargo, una vez hecha la depuración o selección de archivos menores a 16 MB quedaban pocos archivos utilizables. Surgió entonces la necesidad de encontrar una fuente de recursos suficientemente grande para poder recopilar 60GB de archivos. Archivos que a su vez, deben ser de tipos variados ya que luego de la inserción se validará la consistencia estos mismos.
La solución a este problema fue “descargar la wikipedia”. Wikipedia soluciona dos problemas importantes en la recolección de archivos. El primero obviamente se relaciona con el tamaño o la cantidad de datos que se pueden obtener. El segundo es su característica de enciclopedia libre. Su contenido textual está bajo las licencias GNU y Creative Commons. Si bien, algunas imágenes o archivos de sonidos son consideradas no libres, es un porcentaje prácticamente insignificante en comparación con el volumen de datos libre [5.12].
El algoritmo de la descarga consta de una única línea de código [Figura 5.10]

```
~$ wget --random-wait -r -p -e robots=off
   -U mozilla https://en.wikiquote.org/wiki/Main_Page
```

Figura 5.10: Algoritmo de descarga recursiva sobre Wikipedia.

Wget es una herramienta libre que permite la descarga de contenidos desde servidores [5.13] referenciados por una url. En este caso la url es la página principal de Wikipedia.
--random-wait hace que el tiempo entre solicitudes sea aleatorio entre 0.5 y 1.5 segundos. De esta forma Wikipedia que posee un sistema de verificación para bloquear el uso de este tipo de librerías no interrumpe la ejecución del script.
-r indica que la descarga sea recursiva a través de los links que se encuentran a medida que se lee la página accedida.

-p esta opción indica que se descarguen todos los archivos necesarios para mostrar la página sin la necesidad de descargar otros componentes de internet que hagan a su visualización. Por ejemplo css, js, imágenes, videos, etcétera.
-e robots=off El protocolo de robots.txt es un método para evitar que ciertos bots que analizan los sitios web públicos o privados accedan y/o indexen alguna sección particular del sitio. El comando robots=off ignora la indicación de dicho archivo.
-u mozilla En algunos casos, un host puede bloquear las peticiones de wget mirando el registro del agente de usuario sobre el header de http. En wget por defecto es "wget/número de versión", el comando -u mozilla se usa para simular que la petición se está haciendo desde un browser [5.14].

Una característica a tener en consideración, es que el script se ejecuta por tiempo indeterminado. En esta implementación no un hay flag para indicar un tiempo máximo de ejecución o una cantidad total de descarga. Al mismo tiempo, asegurar 60 Gigabytes en datos, puede llevar varias horas. Por lo tanto la forma de mantener dicha ejecución hasta cumplir la meta, consiste en consultar desde otra consola el tamaño de la carpeta donde se van guardando los archivos descargados [Figura 5.11].

```
~$ du -ha /home/user/wikifiles
```

Figura 5.11: El comando "du" en linux estima el espacio usado por los ficheros.

Luego de la descarga de todos los archivos para la evaluación, restan dos pasos.
El primero consiste en filtrar los archivos menores a 16 Megabytes para asegurar que la colección mantenga el tamaño [Figura 5.12].

```
~$ find . -type f -size -16M -exec mv {}
   /home/user/wikifiles/ ; find . -type f -size -16M -exec
   mv {} /home/user/tesina/documents/ ;
```

Figura 5.12: Mover todos los archivos menores a 16 MB a un único directorio.

Por último, hay que tener en cuenta que Wikipedia está escrita en muchos idiomas, y el nombre de los archivos descargados en muchos casos coinciden con el título del artículo. Con lo que puede haber textos en ruso, chino o cualquier otro idioma que necesite un encoding distinto para representar los caracteres del nombre. Para optimizar los tiempos de carga, no se hace un encoding de los archivos desde la aplicación desarrollada. Se pone como condición que los archivos ya estén normalizados para poder leer su metadata y su contenido correctamente. Al igual que en los pasos previos, se utiliza un comando del shell de linux. El comando

[Figura 5.13] debe filtrar todos los archivos de un directorio y sobre escribir los caracteres especiales, como por ej: â, ë, ü, con “_” (guión bajo)

```
~$ find . -type f -print0 |
  perl -n0e '$new = $; if($new =~ s/[^[:ascii:]]//g) { print("Renaming
  $_ to $new\n"); rename($_, $new); }'
```

Figura 5.13:Encoding de filenames. Normaliza los nombres de los archivos.

5.6 Sobre arquitectura del cluster

Todos los estudios fueron realizados en un cluster de la Facultad de Informática. El mismo está compuesto por máquinas con las siguientes características

- Ubuntu 16.04 arquitectura de 64 bits.
- CPU i5 - 2.8 GHz
- 8 GB de memoria
- 100 GB libres (aproximadamente) de disco rígido

En todos los casos de estudio se asigna una máquina para cumplir con el rol de cliente-servidor donde corre la aplicación web. Aplicación encargada de unificar todas las funcionalidades previamente desarrolladas.
Con respecto a la base de datos, dependiendo de la estructura a evaluar sobre MongoDB se asigna de manera dedicada una máquina a cada rol. Cada nodo primario y cada réplica va a correr sobre un máquina dedicada a tal función. Sin embargo como se observó anteriormente el rol de router y config sobre los entornos distribuidos compartirán el mismo recurso concurrentemente.

5.6 Resumen de las herramientas desarrolladas y utilizadas

La [Figura 5.14] muestra cómo fue la evolución de los desarrollos a lo largo del tiempo para llegar al sistema final encargado de automatizar las inserciones, chequeos de consistencia y registro de tiempos de rendimiento.

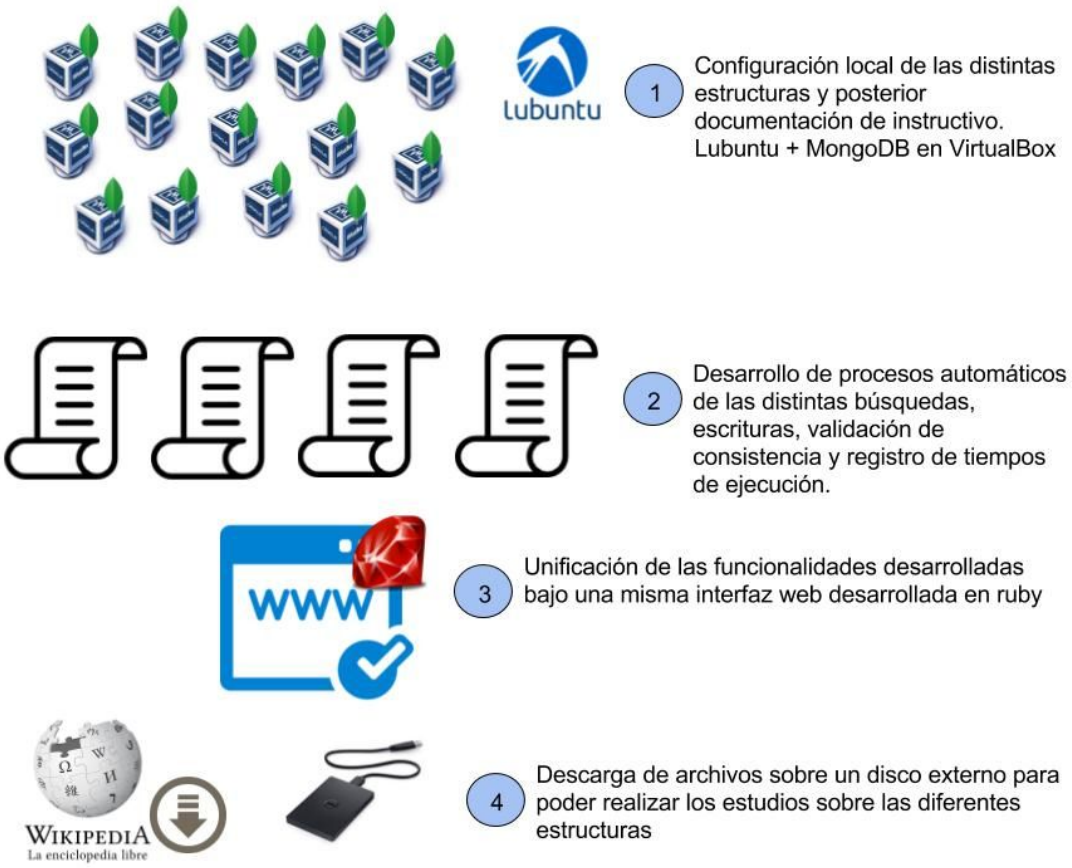


Figura 5.14: Evolución cronológica de las herramientas utilizadas para realizar las evaluaciones prácticas de cada arquitectura.

La [Figura 5.15] muestra cómo interactúan las herramientas desarrolladas y descritas a lo largo del capítulo 5 en en el contexto del estudio de alguna de las cuatro estructuras planteadas.

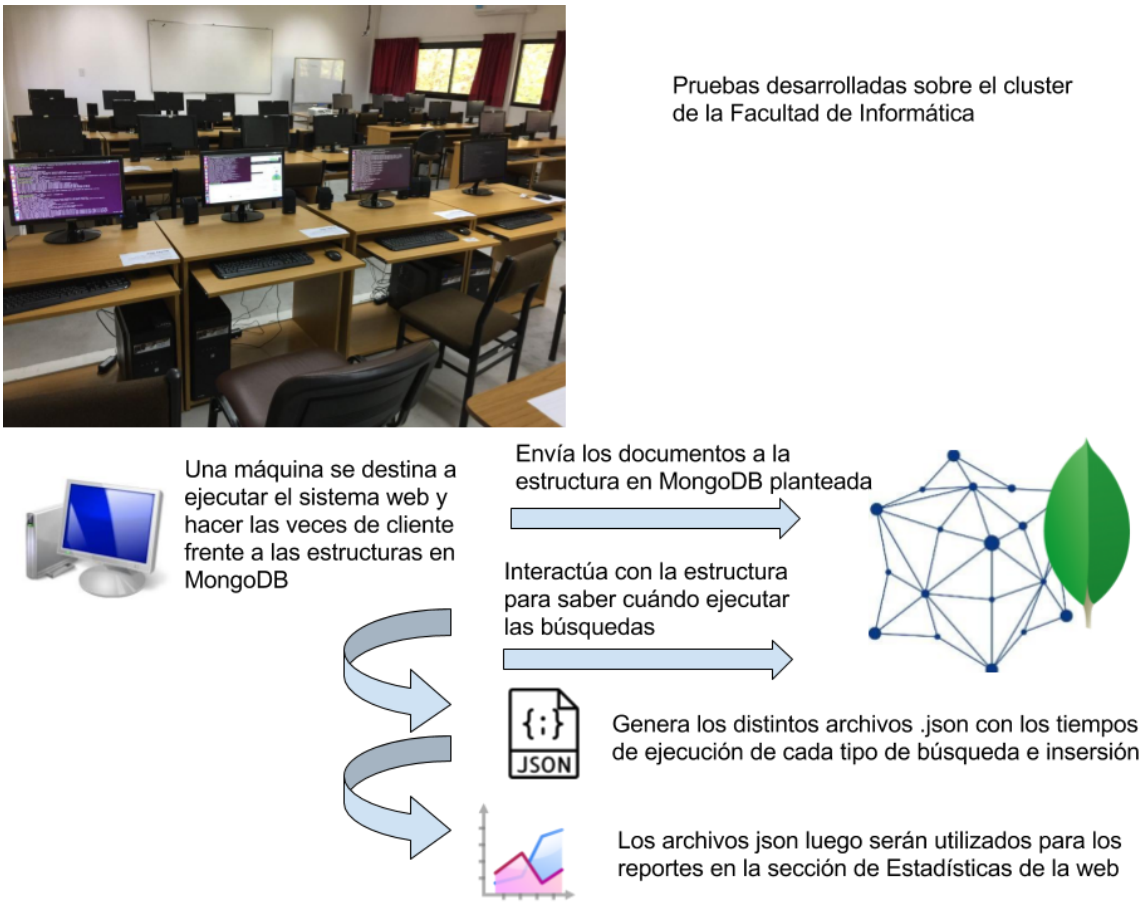


Figura 5.15: Esquema de interacción entre los actores y las herramientas desarrolladas para el estudio del rendimiento.

6. Resultados obtenidos y conclusiones

En el siguiente capítulo se mostrará en un comienzo el detalle de los resultados obtenidos para cada uno de los distintos tipos de diseño. En segundo término se analizarán las similitudes, ventajas y desventajas de cada estructura en concordancia con dichos resultados.

6.1 Estructura centralizada sin réplicas

La estructura centralizada consta de un único servicio encargado de procesar todas las consultas del cliente [Figura 6.1]

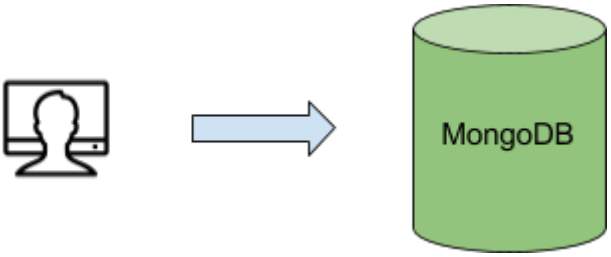


Figura 6.1: Representación de esquema centralizado.

Analizando los resultados en la [Tabla 6.1] se puede observar que la escritura de un documento sobre la base de datos no varía en cuanto al tamaño de la base. Se mantiene a lo largo de todas las inserciones por debajo de 10 milisegundos. Con respecto a las consultas hay un crecimiento lineal.

	Inserción	Búsqueda Extensión	Búsqueda Nombre	Búsqueda Tamaño
1 GB	3	964	204	630
2.5 GB	8	5714	6201	6390
5 GB	2	38421	40065	38122
10 GB	1	85074	84579	84344
20 GB	1	186981	187302	187142

Tabla 6.1: Tabla con los resultados del enfoque centralizado sin réplicas.

Existe una diferencia considerable de tiempos entre los inicios con cantidades de datos que no superan los 5 GB y los resultados posteriores. Por lo tanto, se deben ver los resultados contemplando dos subgrupos diferentes.

En el inicio de las pruebas, los tiempos de respuesta tienen mucha variabilidad. Las consultas realizadas sobre el primer GB se realizan en 0,6 segundos en promedio. Cuando se repiten las mismas consultas sobre 2,5 GB (1 vez y media más que el último tamaño evaluado) los tiempos de ejecución se ubican en promedio en 6 segundos. Lo cual marca que se incrementó en 10 veces el tiempo de respuesta. Si se hace el mismo contraste entre 2,5 GB y 5 GB la cantidad de datos se duplica pero el tiempo de ejecución se acerca a 38 segundos. Unas seis veces más que el último registro.

Cuando la base de datos tiene mayor cantidad de archivos guardados, los tiempos de respuesta son más predecibles observando la relación con el tamaño. El tiempo promedio sobre 5 GB de datos está en 38 segundos. Al duplicar la cantidad en 10 GB de datos llega a 84 segundos y finalmente sobre los 20 GB ronda los 187 segundos. Comparando los tiempos de ejecución desde los 5 GB en adelante, al duplicar la cantidad de datos, la relación del tiempo de respuesta se incrementa 2,2 veces más que la anterior. La relación es lineal ya que prácticamente ambas variables se duplican.

Por el lado de la integridad de los datos, todos los archivos pudieron ser recuperados exactamente como se los guardó anteriormente. La revisión por hash dió idénticos resultados en todos los casos.

En la [Figura 6.2] se pueden observar de manera gráfica los resultados previamente representados en la tabla. Por un lado es posible ver el crecimiento lineal que hay en relación al tiempo de respuesta en concordancia con el tamaño de la base. Además se puede observar el quiebre de la línea entre los primeros GB de datos y los siguientes resultados.

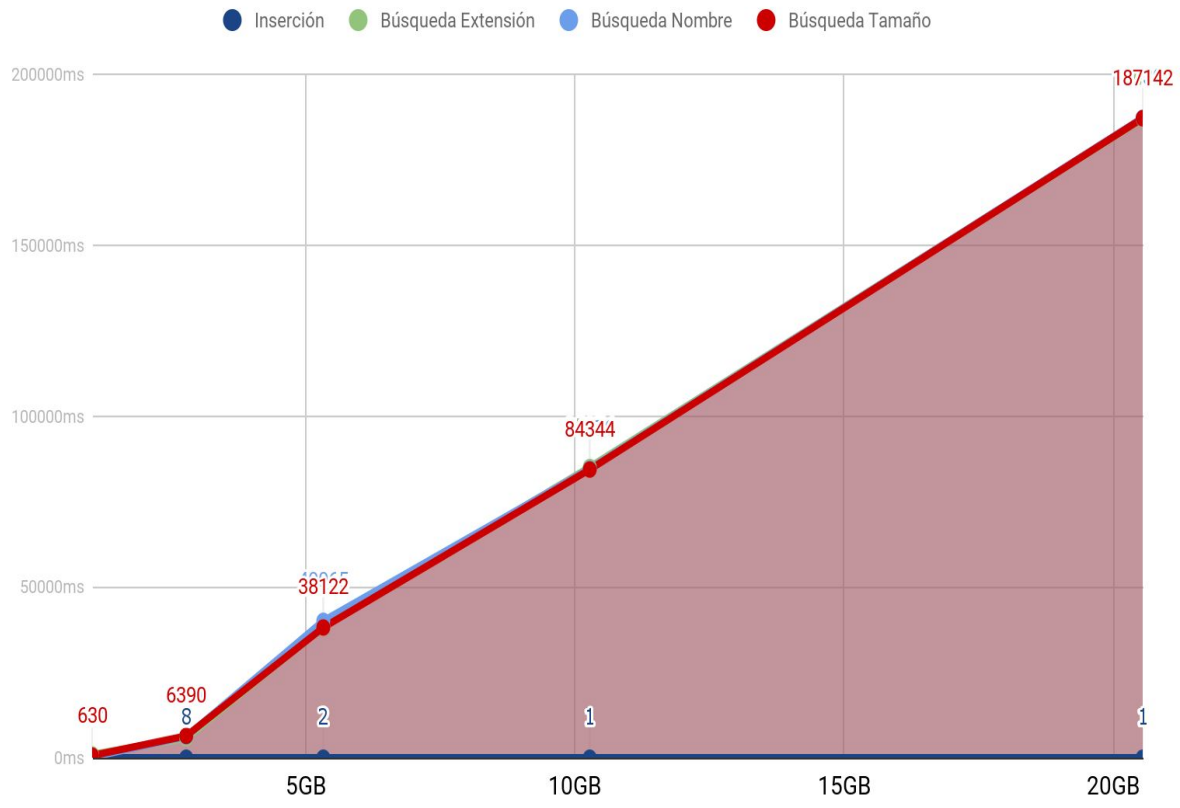


Figura 6.2: Representación gráfica de los resultados del enfoque centralizado sin réplicas.

6.2 Estructura centralizada con réplicas

La estructura centralizada con réplica, además de responder a las consultas de clientes externos, se encarga de sincronizar entre todos los nodos la última versión de los datos [Figura 6.3]

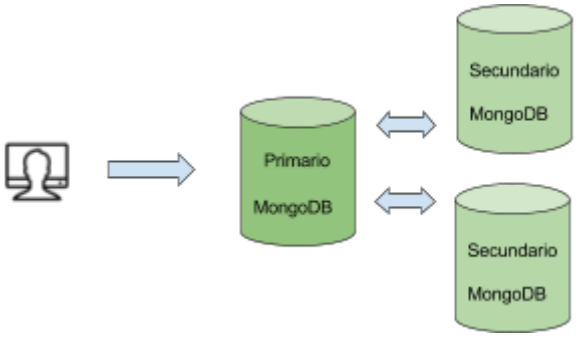


Figura 6.3: Representación de esquema centralizado con réplicas.

Como se explicó en la sección [5.6 Sobre arquitectura del cluster] en todos los casos de estudio se asigna una máquina externa al cluster para cumplir con el rol de cliente y/o servidor donde corre la aplicación web. Decimos que la máquina es cliente y/o servidor ya que además de ser el espacio físico donde se ejecuta la aplicación web cumple con el rol del cliente que interactúa con la base MongoDB.
La [Tabla 6.2] presenta los resultados obtenidos de la estructura centralizada con réplicas. Por el lado de la inserción de un documento, no existen cambios en el tiempo de ejecución y la relación con el tamaño de la base de datos. Se mantuvo muchas veces en 1 milisegundo e incluso en punto de mayor cantidad de datos la escritura del archivo se hizo en menos de 1 milisegundo. Comparando con lo anterior, en ambos enfoques centralizados no hubo variación en los resultados sobre la escritura.
Se puede concluir entonces que hasta los 20 GB en estructuras centralizadas la escritura tiene tiempos constantes por dato independientemente de la cantidad de información que almacene la base. Además es muy probable que mientras haya espacio libre en disco la tendencia se mantenga más allá de la cota superior de 20 GB.
En cuanto a los resultados de las consultas, nuevamente se registra una diferencia de tiempos en proporción del crecimiento de los datos. Cuando la base tiene poca información se hace impredecible el próximo resultado. Pero cuando supera los primeros GBs se puede ver una tendencia o una relación entre el crecimiento del tiempo y los datos almacenados.

	Inserción	Búsqueda Extensión	Búsqueda Nombre	Búsqueda Tamaño
1 GB	4	2708	2563	2565
2.5 GB	8	24324	6798	6798
5 GB	1	54524	57109	52761
10 GB	1	139613	136807	137869
20 GB	<1	261171	260934	260424

Tabla 6.2: Tabla con los resultados del enfoque centralizado con réplicas.

La primer porción de resultados es la menor a 5 GB. Sobre el primer GB de datos las consultas tardan en promedio 2.5 segundos. Mientras que en la siguiente consulta sobre 2.5 GB tarda en promedio 6.8 segundos. Lo cual demuestra que a pesar de que la diferencia en tamaño de datos es 1.5 veces mayor, el tiempo de resolución se incrementó casi tres veces con respecto a la primer consulta.

Por otro lado, se observa que en la búsqueda por extensión (coloreada en verde) sobre los 2.5 GB la respuesta fue de 24 segundos mientras que las otras consultas sobre la misma porción de datos promedian los 7 segundos. Así mismo, se observa que sobre todas las demás evaluaciones el tiempo de respuesta fue similar entre todas las consultas. Con lo cual, si bien no es posible llegar a una conclusión sobre el motivo que causó dicha diferencia, se puede inferir que fue algo excepcional dentro de los resultados recopilados.

La segunda porción de resultados a analizar abarca desde los 5 GB en adelante. En las consultas de este subgrupo, en promedio el tiempo se duplica al duplicarse el tamaño de la base de datos. El crecimiento sobre el tamaño y el tiempo es lineal.

Con respecto a la integridad de los datos, todos los archivos recuperados de base fueron validados por hash y confirmaron ser exactamente el mismo archivo al momento de guardado.

En la [Figura 6.4] se puede ver la representación gráfica de los datos presentados en la [Tabla 6.2]. Al igual que en el enfoque sin réplicas se observa el crecimiento lineal entre el tiempo de respuesta y el tamaño de la base. A su vez, se ve representado el punto de inflexión entre las líneas de búsqueda previas y posteriores a 5 GB.

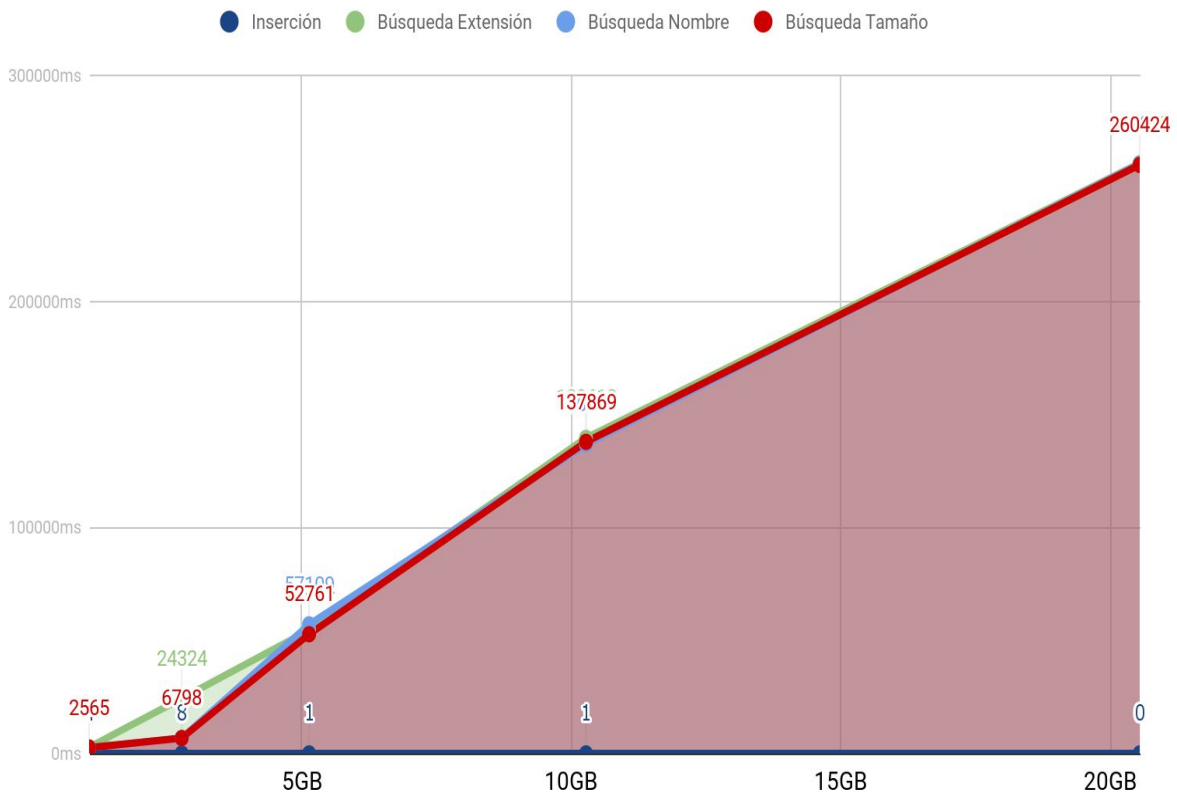


Figura 6.4: Representación gráfica de los resultados del enfoque centralizado con réplicas.

6.2.1 Comparación entre estructuras centralizadas

Si se contrastan los resultados de la estructura sin réplicas sobre la estructura con réplicas se puede observar un incremento en los tiempos de respuesta [Figura 6.5].

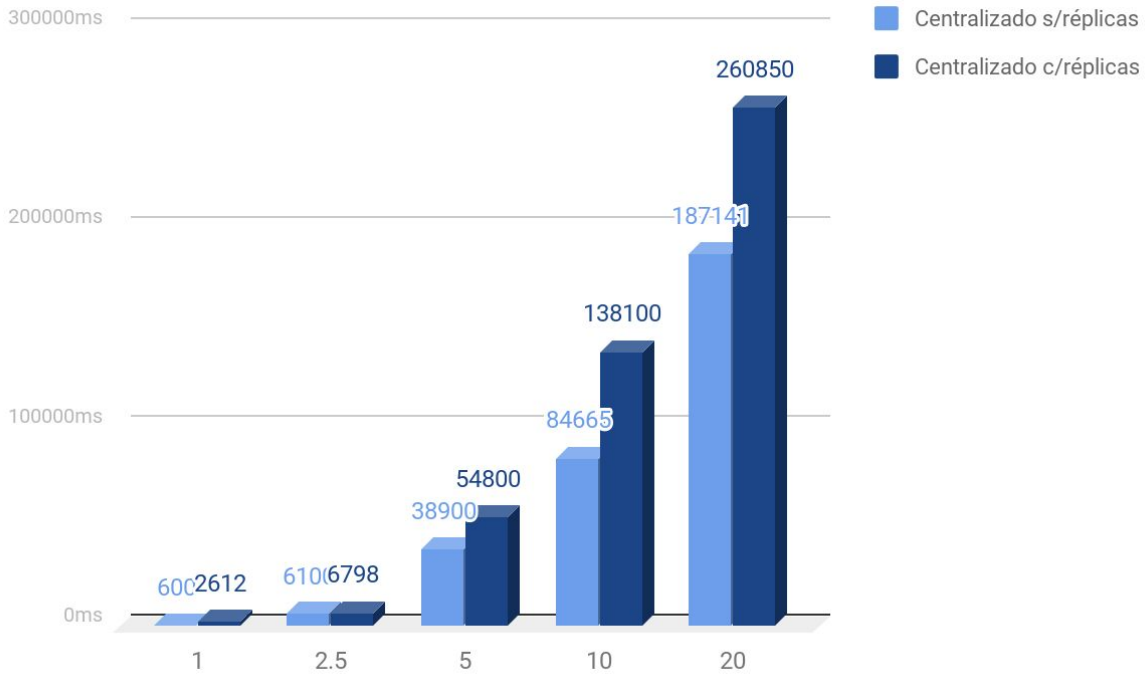


Figura 6.5: Comparación de los tiempos promedio de búsqueda entre estructuras centralizadas sin y con réplicas.

Previamente los resultados demostraron que hay una marcada diferencia de rendimiento entre el primer y el segundo intervalo de datos.
En la comparación de los primeros intervalos con 1 y 2.5 GB se observa que el incremento es unas 4 veces superior en promedio entre la estructura que posee réplicas por sobre la que no posee.
Al superar la barrera de los 5 GB las disparidades entre las arquitecturas empiezan a ser menores. La diferencia de tiempos es en general 1,5 veces mayor en la arquitectura con réplicas que la arquitectura simplemente centralizada.
Se puede concluir entonces que mantener actualizadas otras copias o réplicas de la base de datos penaliza los tiempos de respuesta.

6.3 Estructura distribuida con dos shards

Con la estructura distribuida, el contenido total de la base de datos se divide sobre los shards disponibles. Desde el funcionamiento interno de cada shard se mantienen actualizadas las réplicas y se responden las instrucciones analizando

únicamente la porción de datos que contiene dicho shard [Figura 6.6]. Por otro lado, el procesamiento sobre el direccionamiento de las consultas y la actualización de la metadata que se utiliza en este proceso, es responsabilidad del servicio de Router y el servicio de Config de MongoDB. Ambos servicios corren sobre un único nodo dedicado en de la arquitectura desarrollada.

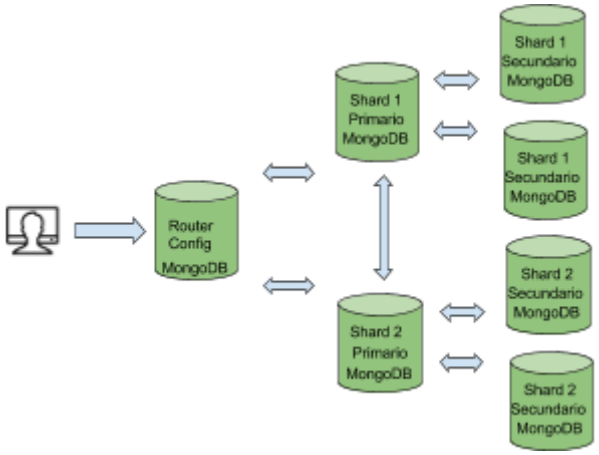


Figura 6.6: Representación de esquema distribuido con réplicas.

En la [Tabla 6.3] se detallan los resultados recopilados de la estructura distribuida sobre dos shards. Al igual que en las estructuras centralizadas, la inserción no se ve alterada en cuanto al tiempo de ejecución y la cantidad de datos almacenados. En general ninguna de las inserciones superan los 10 milisegundos. Analizando los tiempos de las consultas, se ve que los resultados no fueron uniformes como en los enfoques centralizados. Sobre todo teniendo en cuenta los resultados de la primera porción de datos insertados, en el intervalo que comprende hasta los 5 GB. A medida que la base empieza a tener mayor cantidad de información, las respuestas van convergiendo hacia cierta regularidad. El crecimiento entre los tiempos de respuesta y la cantidad de datos que conforman a la base crecen prácticamente de la misma forma, tal como sucedía en el enfoque centralizado.

	Inserción	Búsqueda Extensión	Búsqueda Nombre	Búsqueda Tamaño	Búsqueda Shard
1 GB	6	1019	930	897	2
2.5 GB	4	3657	704	3542	2
5 GB	1	18304	7001	6986	3
10 GB	2	82028	101901	74459	89
20 GB	2	190842	715	189937	68

Tabla 6.3: Tabla con los resultados del enfoque distribuido con dos shards.

Viendo el detalle de los resultados, se observa que la búsqueda por nombre con 20 GB cargados se ejecutó en menos de un segundo. A su vez, consultas similares y sobre la misma cantidad de información superan los tres minutos de ejecución. Por otro lado, analizando la búsqueda por nombre sobre 20 GB en las estructuras centralizadas o la otra estructura distribuida con tres shards, el tiempo de ejecución siempre es similar a las otras consultas. Se concluye entonces que esta búsqueda fue en verdad una anomalía en el experimento.
Una posible justificación de esta anomalía se puede encontrar en el funcionamiento de la consulta. En las bases de datos, buscar dos veces seguidas por el mismo parámetro difícilmente necesite del mismo tiempo de ejecución. Esto es porque los manejadores suelen almacenar durante cierto intervalo una copia del objeto recién consultado. Así mismo, esta misma lógica puede ocurrir para la escritura.
Para elegir el nombre a buscar, el script remueve de manera aleatoria un elemento de la colección de nombres insertados. Sin embargo, si bien sabemos que ese nombre nunca fue consultado anteriormente, nada nos asegura que haya sido insertado hace poco. En ese caso se podría explicar el tiempo menor a un segundo ya que la base de datos respondió con un dato alojado en memoria.
Una crítica que surge en base a este dato y la manera en que se hicieron las pruebas, consiste en la reiteración de las consultas. Si bien las consultas se hacen sobre distintos tamaños de la base de datos, se ejecutan una única vez sobre una cantidad de datos determinada. Por poner un ejemplo, si la misma consulta se ejecutara diez veces en cada punto de consulta habría diez tiempos distintos para sacar una conclusión más acertada sobre cuánto tiempo lleva recorrer la colección en busca del archivo con el nombre “x”. Se menciona esta observación en trabajos futuros.

Retomando el análisis de los intervalos de búsqueda, en el primero el promedio de respuestas se encuentra en 1 segundo. Sobre la siguiente evaluación donde la base crece 1.5 veces su tamaño, las mismas consultas tardan 3.5 segundos en promedio, lo que es similar a decir que los tiempos crecen 3.5 veces. El salto de 2.5 a 5 GB, donde se duplica la base de datos, tiene como promedio 7 segundos, lo que demuestra un crecimiento estrictamente lineal en relación tiempo-tamaño sobre la estructura.
Sobre los 10 GB el tiempo promedio de respuesta está en 85 segundos. Se duplicó el tamaño de la base de datos, pero el tiempo de búsqueda se incrementó un poco más de 12 veces.
Por último las consultas sobre 20 GB promediaron los 190 segundos. Lo que significa que al duplicarse el tamaño de la base de datos, los resultados crecieron 2.2 veces con respecto a la marca anterior. En este punto se puede ver que el crecimiento en relación tiempo y tamaño es prácticamente lineal.

La búsqueda por shard es un tipo consulta agregado únicamente para evaluar el caso en que la respuesta deba ser procesada en su totalidad por uno y solo uno de los shards. En base a los resultados, se observa que los tiempos están por debajo de los 100 milisegundos en todos los casos. El motivo de esta diferencia en tiempos de búsqueda con respecto a las consultas anteriores se debe a cómo está desarrollada y una particularidad que tiene Mongo sobre las estructuras distribuidas.
Para poder simular una consulta que sea respondida por un único shard, se desarrolló un proceso encargado de conectarse de manera directa a un shard y guardar en un archivo externo la colección de ids que contiene. Terminado este paso, se arma una consulta pidiendo todos los elementos cuyo id pertenezca a la colección previamente creada. La particularidad que tiene Mongo sobre los entornos distribuidos, es que al momento de elegir la clave para distribuir los documentos, el campo “id” en este caso, crea un índice sobre dicha clave para tener mayor velocidad sobre la colección distribuida. Por lo tanto, los tiempos de respuesta en este caso no están mostrando nada más allá de la velocidad que tiene Mongo sobre los campos indexados. Se puede analizar de los resultados que sobre los índices de Mongo no hay grandes cambios de rendimiento en base a la cantidad de datos que contiene la colección. Los tiempos de ejecución son muy similares a los tiempos de inserción. Por debajo de los 100 milisegundos y regulares a pesar de la diferencia de tamaños considerada en cada búsqueda.
Con respecto a la integridad de los datos, todos los archivos recuperados de base fueron validados por hash. Se confirmó en todos los casos que se trataba de exactamente el mismo archivo al momento de ser guardado.
La [Figura 6.7] representa de manera gráfica los resultados presentados previamente en la [Tabla 6.3]. Como se mencionó, las consultas muestran un crecimiento lineal desde los 5 GB de datos. Un crecimiento que no ocurre sobre el primer intervalo de consultas con una base menor a los 5 GB de tamaño.

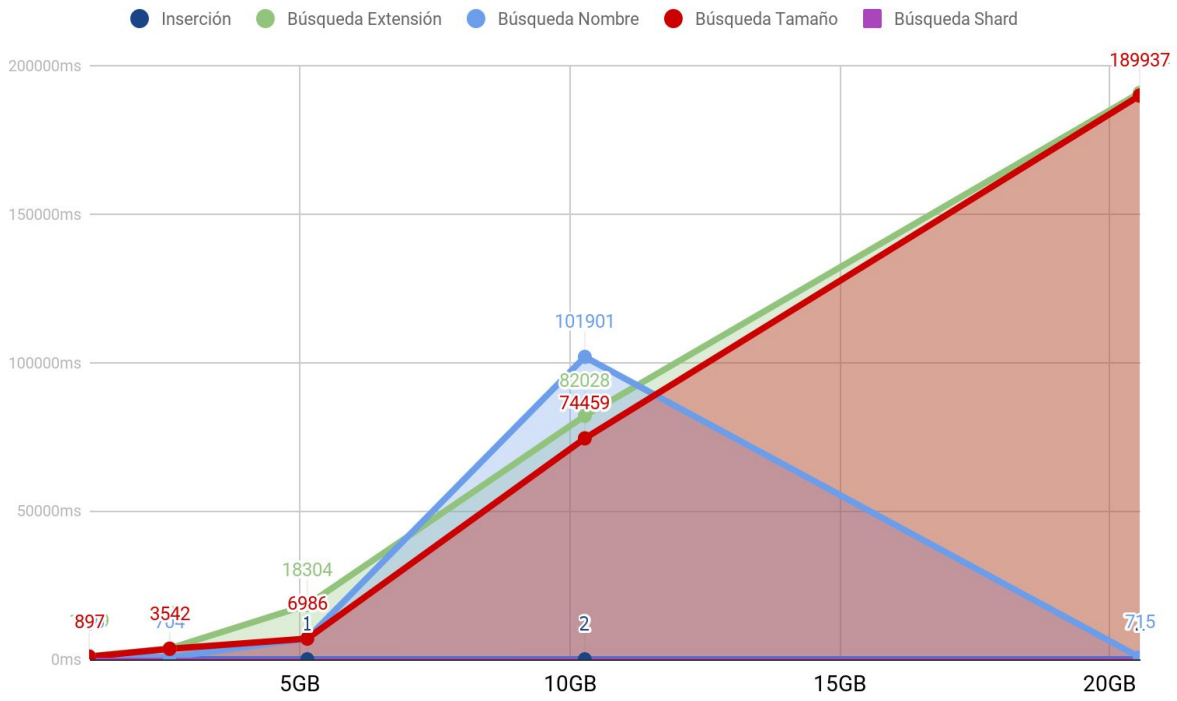


Figura 6.7: Representación gráfica de los resultados del enfoque distribuido con dos shards.

6.4 Estructura distribuida con tres shards

La estructura distribuida sobre tres shards posee las mismas cualidades de la estructura anterior. Cada fracción de datos administra internamente el estado de las dos réplicas. También existe el rol del Router y el Config. La única particularidad es que al haber un nuevo shard disponible, la información se fragmenta en tres porciones. Con lo cual para poder montar la estructura se utiliza una red compuesta de 10 máquinas. [Figura 6.8]

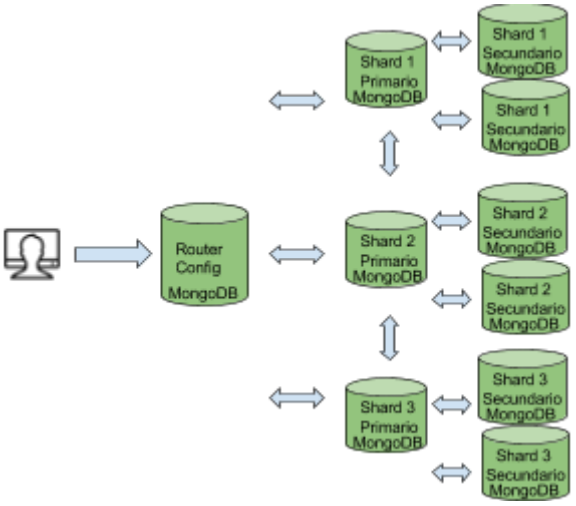


Figura 6.8: Representación de esquema distribuido de tres shards con réplicas.

Sobre la [Tabla 6.4] se detallan los resultados recopilados de las distintas consultas sobre la estructura distribuida en tres shards. Se observa una vez más que la escritura de un documento se mantiene a lo largo de las inserciones por debajo de 10 milisegundos. La cantidad de datos que componen a la base no modifica el rendimiento sobre las instrucciones de escritura para un documento determinado.
Analizando las búsquedas de orden N se observa una diferencia en cuanto a proporciones entre tamaños y tiempos de respuesta. Al igual que en los experimentos anteriores el análisis se lo puede dividir sobre dos subgrupos. La particularidad de esta estructura es que el resultado que tendía a regularizarse luego de los primeros 5 GB, ahora lo hace después de los 10 GB. El corrimiento de esta marca se debe indefectiblemente al incremento del tercer shard.

	Inserción	Búsqueda Extensión	Búsqueda Nombre	Búsqueda Tamaño	Búsqueda Shard
1 GB	5	180	18	14	2
2.5 GB	4	2354	2259	2252	1
5 GB	2	4610	4517	4512	2
10 GB	2	38298	18724	9563	32
20 GB	3	112304	113118	112901	2

Tabla 6.4: Tabla con los resultados del enfoque distribuido con tres shards

El primer conjunto a analizar está compuesto por los intervalos de 1, 2.5, 5 y 10 GB. Entre ellos el primer tiempo de respuesta relevado sobre 1 GB de datos está promediando el orden de los 100 milisegundos. Al incrementar 1.5 veces el tamaño de la base, los tiempos de respuesta se encuentra alrededor de 2.2 segundos. En tiempos de respuesta significa un incremento de 22 veces el valor previo. Luego, al duplicar la base a 5 GB, el tiempo de búsqueda es de 4.5 segundos en promedio. Un crecimiento exactamente lineal, ya que se duplicó la cantidad de datos y con ellos los tiempos de respuesta. Los resultados recopilados sobre los 10 GB, son quizá el caso más dispar si se comparan las tres búsquedas sobre la misma cantidad de información. Tomando como promedio unos 20 segundos de procesamiento para ejecutar las búsquedas, el incremento fue de 4 veces su valor.
El último subgrupo de consultas es sobre los 20 GB. En este punto ya se puede ver un resultado muy homogéneo. Sobre las tres consultas el tiempo de respuesta fue prácticamente el mismo, con centésimas de diferencia. En promedio el tiempo fue de 112 segundos, lo que representa un incremento de 5.6 veces el tiempo de búsqueda por sobre la duplicidad en la cantidad de datos.

La búsqueda por un shard específico tuvo un pico de 32 milisegundos, pero luego se encontró siempre entre 1 y 2 milisegundos. Al igual que la estructura con dos shards los tiempos son prácticamente inalterables sobre los 20GB de datos. Se puede ver la diferencia de rendimiento que implican los índices de Mongo sobre las colecciones.
Por el lado de la integridad de los datos, los archivos pudieron ser recuperados exactamente como se los guardó anteriormente. Todas las operaciones de lectura validaron correctamente los archivos recuperados.
La [Figura 6.9] representa gráficamente los resultados de la [Tabla 6.4]. Al hacer una comparación visual con la [Figura 6.8] se pueden ver dibujos muy parecidos a nivel gráfico entre los subgrupos de consultas descritos. Si bien cambia la escala sobre los milisegundos, es notable la similitud de proporciones. Tanto en la inclinación de las líneas del primer conjunto como el último conjunto se pueden ver las semejanzas.

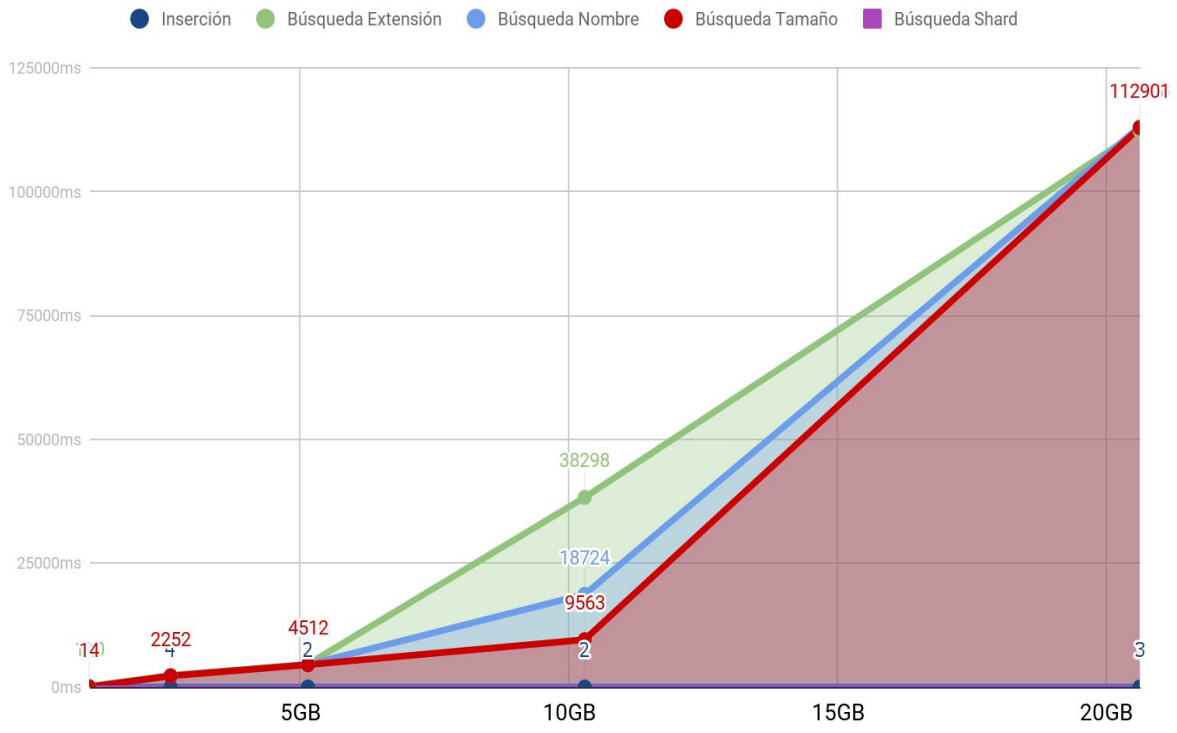


Figura 6.9: Representación gráfica de los resultados del enfoque distribuido con tres shards.

6.4.1 Comparación entre estructuras distribuidas

Si se comparan las estructuras distribuidas se puede notar una mejora en los tiempos de respuesta [Figura 6.10]. Al agregar un shard la estructura cuenta con un nuevo nodo para distribuir las cargas de procesamiento.

Lo que antes se procesaba sobre dos máquinas, a niveles de hardware se mejora en un 50% agregando un nuevo nodo. Sin embargo la mejora del tiempo final de las consultas tiene distintos niveles en base a la cantidad de datos almacenados.
Manteniendo cierta relación entre tiempos promedios, los intervalos a analizar se los puede dividir en los datos menores y mayores a 10GB.
En el primer subconjunto, sobre el primer GB de datos la diferencia entre estructuras representa unas trece veces el valor promedio de los tres shards con respecto a dos. Con 2.5GB se mejoró 2.4 veces el tiempo al agregar un nuevo shard, mientras que con 5GB el tiempo mejora un poco más del 50%.
En el segundo grupo, la comparación sobre los 10GB demuestra una mejora de casi 4 veces el tiempo de respuesta de los tres shards por sobre los dos. Finalmente en los 20GB la mejora es de aproximadamente el 42% comparando tres sobre dos shards.

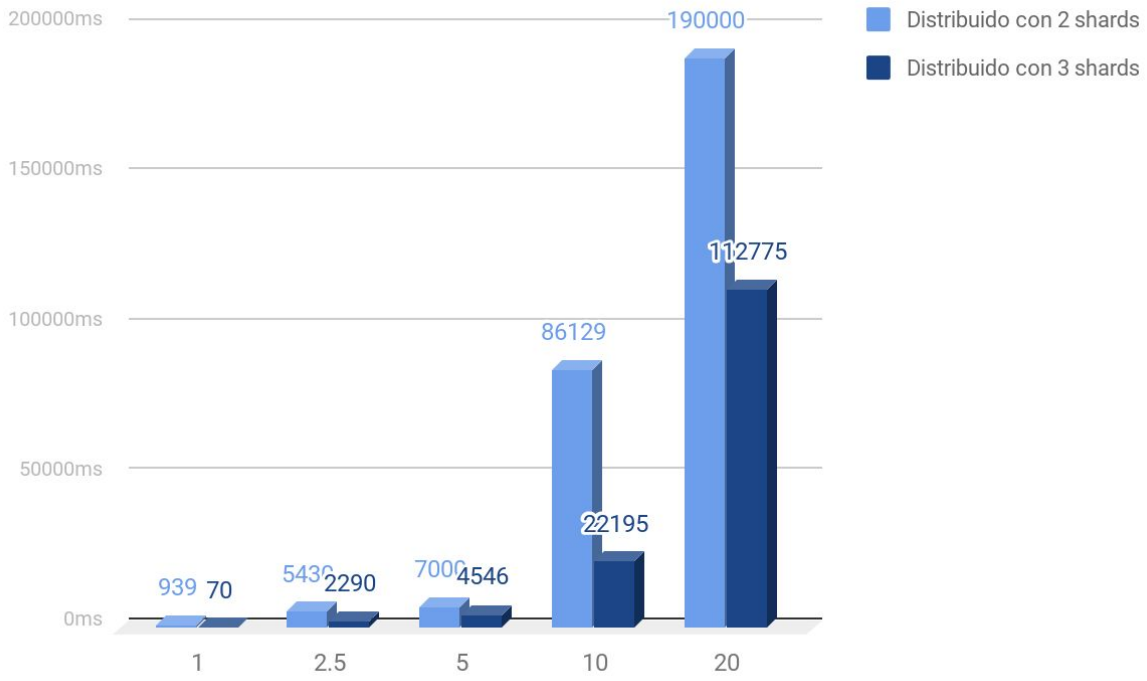


Figura 6.10: Comparación de los tiempos promedio de búsqueda entre estructuras distribuidas con dos shards y tres shards.

6.5 Comparación entre estructuras centralizadas y distribuidas

Como ocurrió a lo largo de todas las pruebas, la inserción se mantuvo inmutable. Respecto de todas las estructuras analizadas y la cantidad de datos almacenados no hubo variaciones sobre el tiempo de escritura. El promedio general estuvo por debajo de los diez milisegundos en todos los casos.
Las búsquedas, en cambio tuvieron marcadas diferencias con las diferentes estructuras [Figura 6.11].

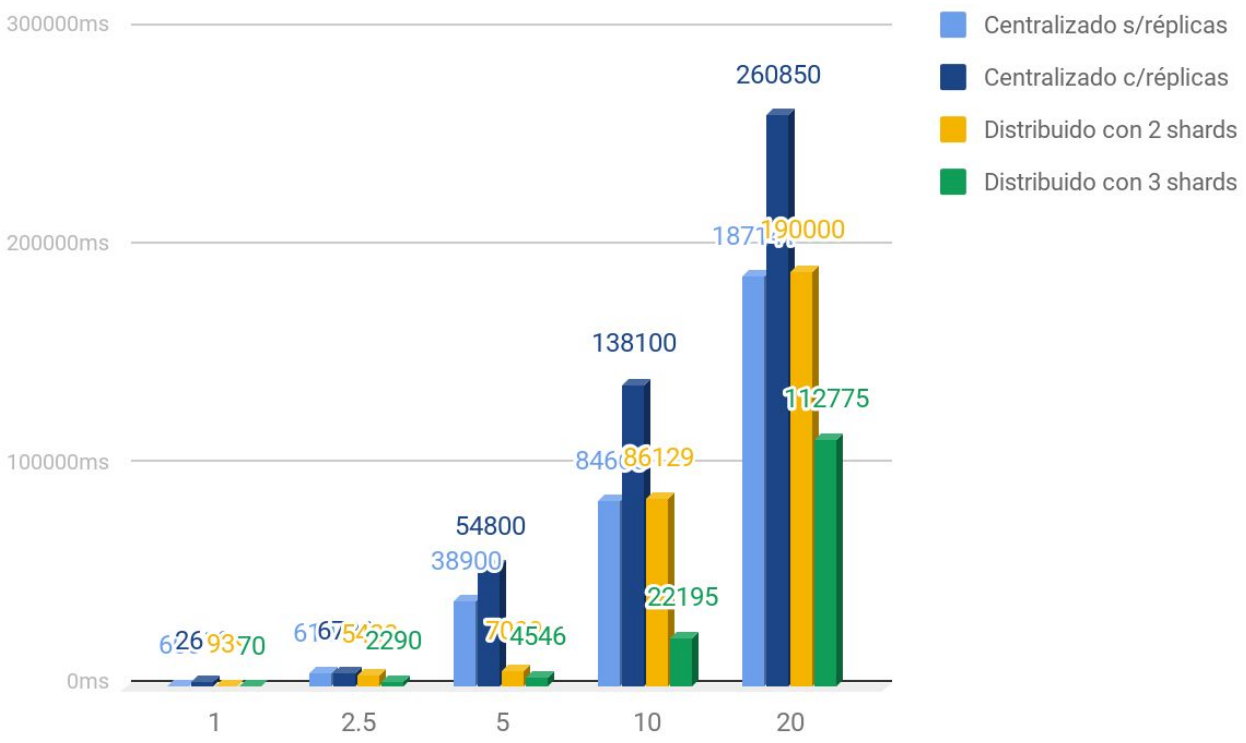


Figura 6.11: Comparación de los tiempos promedio de búsqueda entre las cuatro estructuras analizadas.

Como ocurrió a lo largo de todas las observaciones. Hay dos subgrupos de datos recolectados. En todos los casos, sobre la primera porción de información las búsquedas no tienen un patrón definido en base a su tiempo y la cantidad de datos almacenados. La segunda porción, por el contrario, demuestra un crecimiento constante al analizar ambas variables.

Por el lado de las estructuras centralizadas, se puede ver el quiebre hacia tiempos constantes a partir de los 5GB de datos. Con dos shards empieza a partir de los 10GB y con tres shards recién en los 20GB.

Una posible explicación a esta diferencia se puede encontrar en el almacenamiento de la información. Sobre las primeras consultas la base de datos tiene un tamaño tal que podría permitir tener casi toda la información en memoria. Lo cual explicaría la velocidad en los tiempos de respuesta y su posterior normalización con el crecimiento. A su vez, se puede observar que al agregar shards la potencia de almacenamiento en memoria también se incrementa. Lo que explicaría la diferencia sobre los 5GB entre enfoques centralizados y distribuidos y sobre los 10GB la diferencia con la estructura de tres shards.

6.5.1 Tiempos de consultas agrupados en subconjuntos

A lo largo de los cuatro estudios de rendimiento hubo un comportamiento en los tiempos de respuesta muy similar. Las consultas ejecutadas en los primeros GBs de datos no tenían ninguna relación en base al tiempo de ejecución y a la cantidad de datos almacenados. A su vez, en la segunda mitad ocurre exactamente lo contrario. Al multiplicarse el tamaño de la base prácticamente que se multiplica el tiempo de respuesta en las consultas. Si se analizan los gráficos se puede observar que en aquellas líneas que reflejan el tiempo de búsqueda las pendientes son poco pronunciadas. Por el contrario luego de los primeros datos tienen un crecimiento lineal.
Si analizamos los casos centralizados la pendiente de crecimiento es menor hasta los 2.5 GB y luego a partir de los 5 GB empieza a ser lineal. En la estructura distribuida con dos shards el comportamiento es similar pero el punto de quiebre hacia el crecimiento lineal es luego de los 5 GB. Por último la estructura distribuida con tres shards crece de manera lineal recién después de los 10 GB de datos.
Si bien no se puede afirmar con total seguridad, una posible explicación a este comportamiento puede estar entre MongoDB y la administración de la memoria. Es posible que MongoDB intente mantener todos los documentos posibles en memoria para agilizar los tiempos de respuesta. Esta hipótesis se refuerza si se tiene en cuenta la memoria disponible de cada estructura analizada y los distintos momentos en que se realiza el salto hacia el crecimiento lineal.

6.5.2 Peor rendimiento de búsqueda

De la observación de los resultados se concluye que la estructura con menor rendimiento es la centralizada con réplicas. En todas las pruebas tuvo el peor tiempo de respuesta. Administrar la base de datos y orquestar el estado de las réplicas en un mismo nodo implicó tener los tiempos de respuesta menos eficientes de las estructuras analizadas.
El enfoque centralizado con réplicas implica una mejora en cuanto a la disponibilidad de datos ya que no existe un único punto de entrada a la información. En caso de caer uno de los servicios, la base de datos sigue siendo accesible a través de otro nodo. Sin embargo, se penaliza el rendimiento sobre las búsquedas.

6.5.3 Enfoque centralizado sin réplicas y distribuido sobre dos shards

Analizando el segundo subgrupo de datos, en donde las estructuras toman resultados constantes en relación con el tamaño, se puede observar una igualdad

en los tiempos de búsqueda sobre ambas estructuras. Una única implementación centralizada de datos tiene tiempos de respuesta prácticamente idénticos a una distribuida en dos fragmentos. Si bien los tiempos de ejecución entre ambas arquitecturas fueron similares, sería un error interpretar que no hay diferencias entre estas dos estructuras.
Por un lado se debe tener en cuenta que, en la estructura distribuida, en cada uno de los dos fragmentos o shards recae la responsabilidad de actualizar el estado de otras dos réplicas. Se permite de esa forma aumentar la disponibilidad de la información. Otra forma de verlo es que se incrementa la tolerancia a fallas de la base de datos.
Por otro lado, en los enfoques centralizados no es posible crecer en datos más allá del límite de almacenamiento individual de una máquina. En una arquitectura distribuida, por el contrario, bastaría con agregar un shard a la estructura que se encuentre administrando la base de datos. Como se propone en la sección de trabajos futuros [7.2.1] MongoDB permite incluir un nuevo fragmento e ir redistribuyendo los datos en segundo plano hasta tener un equilibrio entre la información y la cantidad de shards.

6.5.4 Mejor rendimiento de búsqueda

El enfoque distribuido con tres shards tuvo el mejor tiempo de respuesta en todos los análisis efectuados. El primer subconjunto de datos, donde hay una variación significativa entre los resultados de tiempos obtenidos se extiende hasta los 10GB. Recién en el último experimento con 20GB, perteneciente al segundo subconjunto, se puede ver la primer consulta con tiempos coherentes en comparación con las otras estructuras.
Comparando con el peor caso sobre igual cantidad de datos, el enfoque distribuido en tres shards responde 2.3 veces más rápido las consultas que el centralizado con réplicas.
En contraste con la estructura con dos shards y la centralizada sin réplicas, la mejora de rendimiento implica un poco más del 42% en tiempos de búsqueda. Visto de otra forma, si se tienen los datos distribuidos sobre dos fragmentos, agregar un shard nos permite responder un 42% más rápido las consultas. Agregando un componente a la estructura subyacente los tiempos pueden mejorar sustancialmente, sin que se haya ahondado en manejo de índices o estructuras alternativas con datos preprocesados. Se propone en trabajos futuros seguir sobre este tema y derivados.

6.5.5 Resultados sobre un único shard

El desarrollo y la planificación de los estudios de rendimiento hechos a lo largo de este trabajo fue madurando sobre un marco teórico inicial de MongoDB. Marco teórico que empezó a nutrirse y reforzarse con la experimentación.
El objetivo de este experimento, como se explicó en la sección [5.2.3], apuntaba a evaluar un escenario en que todos los documentos necesarios para responder la consulta se encuentren sobre un único shard. La solución planteada para poder simular dicho caso, consistió en tener de antemano la colección de ids que había sobre un shard particular. Luego sobre el cluster se consultaba por todos los ids que pertenecieran a la colección. Lo que en principio pareció una solución simple, en la práctica resultó en tiempos excesivamente bajos en comparación con las demás consultas. De hecho, si se analizan los datos, en la mayoría de los casos el tiempo estuvo por debajo de los tres milisegundos.
La explicación a estas notorias diferencias se basa en el propio funcionamiento de mongo. Como se indicó en la sección [3.2.1.1] MongoDB crea un índice para mejorar la velocidad de respuesta, sobre el o los campos elegidos como Shard Key. Campo que en nuestro estudio fue el campo “id”. En base a esta experiencia se concluye que el experimento demostró de forma práctica la mejora del rendimiento que puede representar el definir índices acordes a las búsquedas que se deban realizar.

7. Conclusiones y Trabajos futuros

7.1 Conclusiones

El desarrollo del estudio implicó varios desafíos. En un primer paso teórico, se investigó sobre el almacenamiento y las distintas posibilidades de plantear una estructura distribuida. Se analizó cómo el enfoque relacional y el NoSQL encaran distintas soluciones a un mismo problema en base a sus características y restricciones. Una vez que se decidió utilizar el enfoque NoSQL, siendo MongoDB la herramienta para experimentar, se continuó con el estudio particular del manejador de base de datos. Se analizó cómo funcionan y cómo se configuran las distintas arquitecturas. Se analizaron las soluciones centralizadas, las réplicas y las distribuidas con réplicas con las particularidades de cada caso. Luego de estudiar estos conceptos, se comenzó a trabajar en la parte práctica de la tesina.

Se inició con pruebas locales configurando los entornos a estudiar sobre máquinas virtuales. Se trabajó en la automatización de las funcionalidades necesarias para el estudio de rendimiento. Se desarrolló en el entorno local un programa para la carga de los archivos, otro para la recuperación y el chequeo de la consistencia y otro para registrar los tiempos de ejecución. A su vez, se hizo el sistema web que unifica todos los procesos bajo una misma interfaz. Finalmente, el último paso consistió en ejecutar todo lo producido de manera local a un cluster real para recopilar los resultados.

La parte final del estudio consistió en analizar y darle una interpretación a los resultados que arrojaron los distintos estudios. Por otro lado permitió analizar mejoras para ser tenidas en cuenta en trabajos futuros y a su vez generó nuevos conocimientos en el funcionamiento de MongoDB.

En cuanto a los resultados de los estudios se puede concluir que en MongoDB las estructuras distribuidas de bases de datos permiten mejorar los tiempos de respuesta por sobre las centralizadas. Comparando los resultados dentro de las arquitecturas distribuidas agregar un tercer shard mejora en más de un 42% el rendimiento sobre dos shards. A su vez quedó demostrada la consistencia del manejador de la base de datos en la escritura como en la lectura. Las validaciones fueron satisfactorias en todos los casos evaluados.

Haciendo un balance en el desarrollo de la tesina es valorable obtener resultados que permiten comparar y analizar las estructuras. Cuando se quiere buscar información sobre este tipo de estudios, resulta común encontrar muchos artículos explicando los beneficios de la distribución en una forma teórica incluso en algunos casos publicitaria. Pero no es tan común el respaldo de la experimentación y los números resultantes.

Como crítica constructiva se destaca que hubiera sido significante la reiteración de las consultas sobre una misma cantidad de datos. Como se analizó en el caso con dos shards y un tiempo de ejecución atípico como lo fue la búsqueda por nombre. Sobre 20 GB de datos, un tiempo tan distinto a la media podría haber sido anecdótico si existían otras nueve consultas por nombre que respalden el resultado de las demás búsquedas realizadas sobre la misma cantidad de datos.

7.2 Trabajos futuros

En base al trabajo realizado y a los experimentos que se presentaron, quedan muchas líneas futuras de trabajo. Muchas de estas líneas de trabajo futuro son mayormente experimentales, con el objetivo de mejorar la caracterización del funcionamiento y rendimiento de mongo.

Experimentar sobre estructuras distribuidas. Teniendo como punto de partida esta investigación y viendo los resultados de las estructuras distribuidas sobre las centralizadas, resulta interesante poder comparar mejoras en tiempos de respuesta entre estructuras distribuidas. Poder concluir si siempre o hasta qué punto agregar shards mejora los tiempos de respuesta.
También se podría analizar si esa mejora es constante en términos proporcionales sobre las estructuras anteriores. Por ejemplo: en este estudio el tiempo de respuesta mejoró en un 42% agregando un shard. Se puede evaluar si la mejora mantiene la proporción si se agrega un cuarto y un quinto shard.

Implementar un File Server. En base a lo concluido de este estudio se pudo constatar que la herramienta Mongo fue consistente en la escritura y la lectura de los datos. Tomando como una primera etapa los archivos menores a 16 MB es posible analizar la creación de un File Server customizable sobre MongoDB.
Teniendo los elementos básicos de creación y lectura sobre una colección de documentos y bajo en análisis de tiempos de respuesta se podría armar una herramienta capaz de gestionar en base a ciertos parámetros una configuración de MongoDB lista para usar como un servicio de archivos a través de una API.

Aplicar Mongo sobre un controlador de versiones de código. Teniendo en cuenta la alta disponibilidad de los datos y las estructuras distribuidas para escalar y mejorar los tiempos de respuesta agregando fragmentos. Se podría trabajar en el desarrollo de alguna herramienta que persista el versionado de código sobre un cluster en MongoDB como por ejemplo Git [7.1]. Para una primer versión se podría hacer un sistema simplificado. Una herramienta que permita sumar sobre cada registro de un archivo, el número de versión, los cambios en base a la versión anterior y el nuevo archivo.

Virtualización y Mongo para estructuras distribuidas. Herramientas como docker permiten virtualizar los componentes que necesita un servicio como Mongo para correr de manera independiente de la arquitectura y el sistema operativo. Podría resultar de interés automatizar y detallar la implementación de estructuras distribuidas de MongoDB sobre un entorno virtualizado que además de ofrecer el software subyacente para su ejecución resuelve la lógica de ruteo de los servicios que se quieran utilizar.

Comparación entre distintos manejadores de base de datos NoSQL. Resulta interesante investigar qué otras opciones además de Mongo se pueden utilizar para almacenamiento distribuido de documentos y analizar entre ellas similitudes y diferencias.

8. Bibliografía

[1.1] Principles of Distributed Database Systems. Özsu, M. Tamer, Valduriez, Patrick, 3era Edición, 2011, Springer-Verlag New York. (pp. 3-5).

[1.2] The Zettabyte Era: Trends and Analysis. Cisco. 7 de junio de 2017. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[1.3] El tráfico de Internet se duplicó en 2016. Cámara Argentina de Internet. 27 de diciembre de 2017.
http://www.cabase.org.ar/el-trafico-de-internet-se-duplico-en-2016/

[1.4] NOSQL Databases. Dr.Stefan Edlich. Noviembre de 2017.
http://nosql-database.org/

[1.5] Key-Value stores: a practical overview. Marc Seeger. 21 de septiembre de 2009. Stuttgart, Germany.

[1.6] Redis. Salvatore Sanfilippo. Noviembre de 2017
https://redis.io/

[1.7] Wide Column Stores - DB-Engines Encyclopedia. Solid IT. Noviembre de 2017.
https://db-engines.com/en/article/Wide+Column+Stores

[1.8] Wide Column Stores - DB-Engines Encyclopedia. Solid IT. Noviembre de 2017.
https://db-engines.com/en/article/Document+Stores

[1.9] Survey of Graph Database Models. Angles, Renzo, Gutierrez, Claudio.1 de febrero de 2008. Chile.

[1.10] The Neo4j Graph Platform – The #1 Platform for Connected Data. Neo4j, Inc.
https://neo4j.com/

[1.11] Wide Column Stores - DB-Engines Encyclopedia. Solid IT. Noviembre de 2017.
https://db-engines.com/en/ranking/graph+dbms

[1.12] Cloud Computing, Principles and Paradigms. Rajkumar Buyya, James Broberg, Andrzej Goscinski. 29 de marzo de 2011. New Jersey y Canada.

[1.13] Principles of Distributed Database Systems. Özsu, M. Tamer, Valduriez, Patrick, 3era Edición, 2011, Springer-Verlag New York. (pp. 459-495).

[1.14] Tutorial MongoDB. Explicando el sharding con una baraja de cartas. Rubén Fernández. 30 de enero de 2014.
http://charlascylon.com/2014-01-30-tutorial-mongodb-explicando-el-sharding-con-una

[1.15] Wide Column Stores - DB-Engines Encyclopedia. Solid IT. Noviembre de 2017.
https://db-engines.com/en/ranking

[1.16] Oracle | Integrated Cloud Applications and Platform Services. Oracle Corporation.
https://www.oracle.com

[1.17] MySQL. Oracle Corporation. 2017.
https://www.mysql.com/

[1.18] SQL Server 2017 on Windows and Linux | Microsoft. Microsoft. 2017
https://www.microsoft.com/en-us/sql-server/sql-server-2017

[1.19] PostgreSQL: The world's most advanced open source database. The PostgreSQL Global Development Group. 2017.
https://www.postgresql.org/

[1.20] Query Documents - MongoDB Manual 3.6. MongoDB. 2017.
https://docs.mongodb.com/manual/tutorial/query-documents/

[2.1] Principles of Distributed Database Systems. Özsu, M. Tamer, Valduriez, Patrick, 3era Edición, 2011, Springer-Verlag New York. (pp. 171-204).

[2.2] Principles of Distributed Database Systems. Özsu, M. Tamer, Valduriez, Patrick, 3era Edición, 2011, Springer-Verlag New York. (pp. 428-433).

[2.3] Comparative Models of the File Assignment Problem. Lawrence W. Dowdy, Derrell V. Foster. 2 de junio de 1982.

[2.4] Apache Cassandra. The Apache Software Foundation. 2017

http://cassandra.apache.org/

[2.5] Benchmarking Cassandra Scalability on AWS – Netflix TechBlog – Medium. Netflix Technology Blog. 1 de noviembre de 2011.
https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

[2.6] Cassandra – A structured storage system on a P2P Network. Avinash Lakshman. 25 de agosto de 2008.
https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/

[2.7] 1.2. Why CouchDB? — Apache CouchDB 2.0 Documentation. Apache Software Foundation. 2017.
http://docs.couchdb.org/en/2.0.0/intro/why.html

[2.8] 8.3. Erlang — Apache CouchDB 2.0 Documentation. Apache Software Foundation. 2017.
http://docs.couchdb.org/en/2.0.0/query-server/erlang.html

[2.9] 1.1. Technical Overview — Apache CouchDB 2.0 Documentation. Apache Software Foundation. 2017.
http://docs.couchdb.org/en/2.0.0/intro/overview.html#acid-properties

[2.10] 1.1. Technical Overview — Apache CouchDB 2.0 Documentation. Apache Software Foundation. 2017.
http://docs.couchdb.org/en/2.0.0/intro/overview.html#distributed-updates-and-replication

[3.1] MongoDB Licensing | MongoDB. MongoDB, Inc. 2017.
https://www.mongodb.com/community/licensing

[3.2] GNU Affero General Public License - GNU Project - Free Software Foundation. The Free Software Foundation. 18 de noviembre de 2016.
https://www.gnu.org/licenses/agpl-3.0.html

[3.3] 10gen Announces Company Name Change to MongoDB, Inc. | MongoDB. MongoDB, Inc. 2017.
https://www.mongodb.com/press/10gen-announces-company-name-change-mongodb-inc

[3.4] Wide Column Stores - DB-Engines Encyclopedia. Solid IT. Noviembre de 2017.
https://db-engines.com/en/ranking

[3.5] ObjectId - MongoDB Manual 3.6. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/reference/method/ObjectId/

[3.6] Convertir Epoch - Tiempo Unix. Misjao.
https://espanol.epochconverter.com/

[3.7] MongoDB Index Internals · Lyons Blog. Lyon Zhang. 19 de febrero de 2014.
http://zhangliyong.github.io/posts/2014/02/19/mongodb-index-internals.html

[3.8] B - Trees :: Data Structures. Rajinikanth B.
http://btechsmartclass.com/DS/U5_T3.html

[3.9] Shard Keys - MongoDB Manual 3.6. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/sharding-shard-key/

[3.10] Data Partitioning with Chunks - MongoDB Manual 3.6. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/sharding-data-partitioning/

[3.11] Sharded Cluster Components - MongoDB Manual 3.6. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/sharded-cluster-components/

[3.12] Replication - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/replication/

[4.1] Gridfs - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/gridfs/

[4.2] RFC 1867 - Form-based File Upload in HTML. E. Nebel, L. Masinter. Noviembre 1995.
https://tools.ietf.org/html/rfc1867

[4.3] Encoding vs. Encryption vs. Hashing vs. Obfuscation. Daniel Miessler. 28 de mayo de 2011.
https://danielmiessler.com/study/encoding-encryption-hashing-obfuscation/#hashing

[4.4] BSON Types - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/reference/bson-types/

[4.5] Shard Keys - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/sharding-shard-key/index.html

[4.6] Shard Keys - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/core/sharding-shard-key/index.html#unique-indexes

[5.1] lubuntu | lightweight, fast, easier. OSUOSL, FOSSASIA. 2017
http://lubuntu.net/

[5.2] Lenguaje de Programación Ruby. Yukihiro Matsumoto. 2017.
https://www.ruby-lang.org/es/

[5.3] psoldier/mongo-sharded-cluster. Pablo Soldi. 30 de mayo de 2017.
https://github.com/psoldier/mongo-sharded-cluster

[5.4] xxHash - Extremely fast non-cryptographic hash algorithm. Yann Collet. 8 de septiembre de 2017.
http://cyan4973.github.io/xxHash/

[5.5] aappleby/smhasher: Automatically exported from code.google.com/p/smhasher. Aappleby. 9 de enero de 2016.
https://github.com/aappleby/smhasher

[5.6] Cursor Explain - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/reference/method/cursor.explain/#cursor.explain

[5.7] Storage Size - MongoDB Manual 3.6.. MongoDB, Inc. 2017.
https://docs.mongodb.com/manual/reference/method/db.collection.storageSize/#db.collection.storageSize

[5.8] soveran/cuba: Rum based microframework for web development. Michel Martens. 2017.
https://github.com/soveran/cuba

[5.9] Bootstrap - The most popular HTML, CSS, and JS library in the world. Mark Otto, Jacob Thornton, and Bootstrap contributors. 2017.
http://getbootstrap.com/

[5.10] Chart.js · GitBook. ChartJs Contributors.
http://www.chartjs.org/docs/latest/

[5.11] Wikipedia - Wikipedia, la enciclopedia libre. Fundación Wikimedia. 9 de diciembre de 2017.
https://es.wikipedia.org/wiki/Wikipedia#Licencia_de_contenido

[5.12] Wget - GNU Project - Free Software Foundation. Free Software Foundation. 15 de septiembre de 2017.
https://www.gnu.org/software/wget/

[5.13] Explainshell.com - wget --random-wait -r -p -e robots=off -U mozilla www.example.com. Idan Kamara. 2017.
https://explainshell.com/explain?cmd=wget+--random-wait+-r+-p+-e+robots%3Doff+-U+mozilla+www.example.com

[7.1] Pro Git. Scott Chacon, Ben Straub, 2da Edición. 26 de agosto de 2009. Appres. (pp. 4-51)

Apéndice I

AI.1 Propagación eager

La propagación activa implica aplicar los cambios sobre todas las réplicas en el contexto de una transacción. Lo que significa que al momento de terminar el commit todas las réplicas tienen exactamente el mismo valor. El commit de dos fases o 2PC [2.2] es un protocolo que se aplica para tal fin. A groso modo, consiste en un nodo coordinador de las réplicas que al momento de actualizar le consulta a todas las demás réplicas si están listas para commitear un cambio.
Si algún nodo responde de manera negativa, cualquiera sea, se cancela la transacción.
En caso de que todos le indican al coordinador que pueden, el coordinador escribe en un log el commit a ejecutar, indica a todas las réplicas que ejecuten el commit y espera por la respuesta de cada réplica.
Por último el coordinador registra cuál fue el resultado final de la transacción en el log [Figura 2.1].

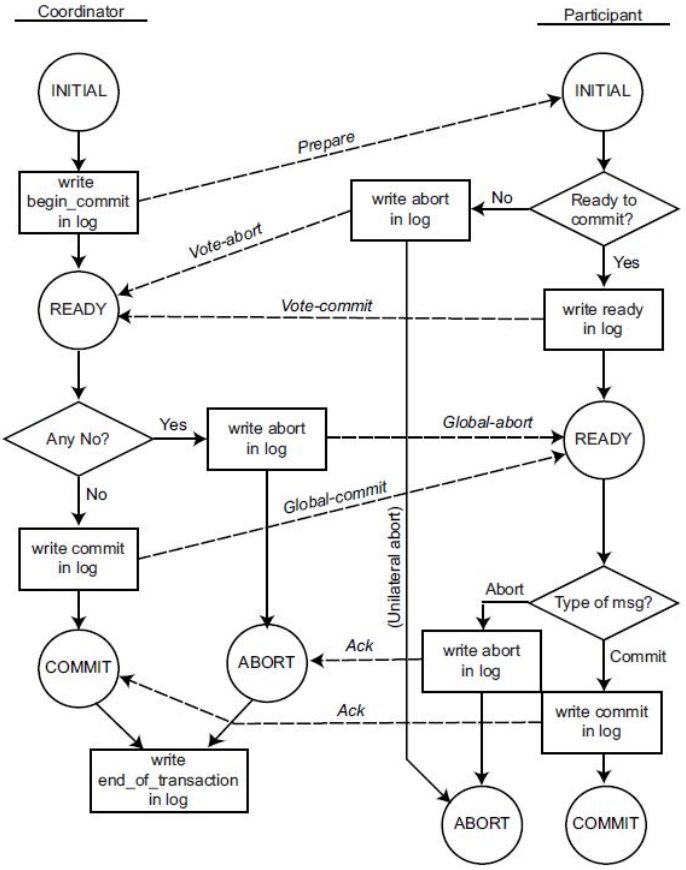


Figura 2.1: Esquema de acciones en un protocolo de dos fases.

Como ventajas se puede considerar que todo se ejecuta en un simple commit, lo cual no genera estado inconsistente. Al ser transaccional cualquier lectura de una copia es una lectura actualizada.
Como desventaja está el rendimiento, ya que el tiempo de la actualización va a ser igual a la respuesta más lenta de todas las réplicas. Por otro lado si una de las copias no se encuentra accesible la transacción no se podrá llevar a cabo ya que todas las copias debe actualizarse.

AI.2 Propagación lazy

La actualizaciones de réplicas con la propagación lazy no ejecuta todas en el contexto de una transacción. No se espera que la transacción actualize a todas las copias. Intenta ejecutar el commit tan pronto como una de todas las réplicas sea actualizada. Luego la propagación a las otras copias se hace de manera asincrónica desde la transacción original. Las réplicas son actualizadas un tiempo después de que se ejecute el commit transaccional.
La propagación Lazy puede usarse en los sistemas donde se pueda tolerar cierto grado de inconsistencia en pos de mejorar el rendimiento. En este caso la posible desventaja puede convertirse en ventaja y viceversa, dependiendo de la lógica de negocios del sistema que está detrás de la base de datos.

AI.3 Técnica centralizada

La actualización centralizada le asigna roles a las réplicas. Una será la réplica maestra encargada de recibir las actualizaciones y las otras serán las réplicas esclavas encargadas de repetir lo que ejecuten las copias maestras.
La ventaja está en que en la actualización se reduce a un único nodo, la réplica maestra no necesita encargarse de sincronizar al resto de las réplicas.
Como desventaja existe la posibilidad de que si hay un sitio central que maneja todas las réplicas maestras haya una sobrecarga que se transforme en un cuello de botella.

AI.4 Técnica distribuida

Las técnicas distribuidas aplican la actualización sobre la copia local del sitio donde se genera la transacción de actualización y luego se propaga a las otras réplicas. La técnica es distribuida ya que distintas transacciones puede actualizar diferentes copias de la misma información alocada en distintos sitios. La técnica puede ser útil en aplicaciones colaborativas donde la toma de decisiones también sea distribuida. La carga de procesamiento puede distribuirse de manera más uniforme y puede proveer mayor disponibilidad de sistema combinado con propagación lazy por ejemplo.

La mayor complicación surge con los casos en que dos sistemas actualizan concurrentemente los mismos datos y la propagación es de tipo lazy. En ese posible escenario la solución de conflictos tiene que contemplar hacer y deshacer operaciones para buscar una “reconciliación” de los datos. Una tarea que generalmente va a poder solucionarse desde la lógica de negocios de la aplicación y no desde un manejador de base de datos.

AI.5 Fragmentación

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

Figura 2.2: Ejemplo de una tabla con todos los datos que definen a un proyecto.

AI.5.1 Fragmentación horizontal

Consiste en dividir una relación a lo largo de sus tuplas. Así cada fragmento tiene un subconjunto de tuplas relacionadas bajo algún discriminador.

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Figura 2.3: Ejemplo de fragmentación horizontal.

La [Figura 2.3] muestra cómo sería la fragmentación horizontal de la [Figura 2.2]. La tabla PROJ1 contiene todos aquellos proyectos cuyo presupuesto no superan los $200000 y PROJ2 guarda la información de los proyectos con mayores presupuesto.

AI.5.2 Fragmentación vertical

Consiste en dividir una relación a lo largo de sus atributos. Cada subconjunto se compone de algunos atributos (columnas) de la tabla original.

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Figura 2.4: Ejemplo de fragmentación vertical.

La [Figura 2.4] muestra cómo sería la fragmentación vertical de la [Figura 2.2]. La tabla PROJ1 contiene solo la información acerca del presupuesto de los proyectos. Mientras que PROJ2 contiene los nombres y la localizaciones.
Cualquiera sea el tipo de división que se decida implementar se debe tener en cuenta que el grado de la fragmentación afectará el rendimiento de las consultas. El grado de fragmentación puede ir del extremo monolítico sin fragmentación, hasta el otro punto en donde hay un fragmento por tupla o por atributo (dependiendo del tipo implementado).
Tener pocos fragmentos de datos muy grandes nos estaría llevando a una situación muy similar al enfoque centralizado en donde el poder de procesamiento se reduce al poder de un único nodo. Por otro lado, muchos fragmentos pequeños frente a la unión de resultados, que es la operación más costosa a nivel relacional, tendría efectos adversos también.
Por lo tanto el punto está en encontrar el grado adecuado de fragmentación que pueda estar equilibrado en comparación con los extremos y siempre teniendo en cuenta las características finales de la o las aplicaciones que van a interactuar con la base de datos.

Apéndice II

AII.1 Topología de red y monitoreo

Los conjuntos de réplicas tienen distintos estados desde el momento que se inicia la red en que se descubren y sincronizan los nodos alcanzando su estado de equilibrio, los estados de fallos que puedan surgir y su posterior resolución.
En cada uno de estos estados a su vez, existen distintos enfoques para implementar las soluciones. Según las prestaciones de cada lenguaje hay distintos tipos de drivers sobre los cuales corren los clientes mongo: los single-thread (por ej. en Perl), los multi-thread (por ej. en PyMongo) o las soluciones híbridas (por ej. en C)[11].

AII.1.1 Descubrimiento de nodos que componen la red

Al momento de invocar el método que inicia un cliente Mongo se envía por parámetro la seed list. Un listado de direcciones ip's o urls que compone a cada nodo de la red. El cliente se encargará entonces de recorrer y mantener una conexión con este conjunto de nodos, pero dependiendo del tipo de driver su comportamiento será distinto.

AII.1.1.1 Multi-thread

En los drivers multithread se crea un thread por cada host, que cumplirá la tarea de monitorear periódicamente un nodo asignado. Al momento que se crea el cliente, el driver inicia en background cada thread para el descubrimiento del set. Cada uno se conecta entonces con su nodo asignado y ejecuta "isMaster()". Genera así el handshake entre cliente-servidor.
La información que recibe el cliente entre otras cosas es: la confirmación de pertenencia al conjunto de nodos que conforman el replica-set. El rol que cumple dicho nodo en esta red (primario, secundario) y por último la lista de todos los miembros que pertenecen al replica-set. Es posible que en esta lista de nodos, exista alguno que el cliente no tenía en su listado local (envíado al momento de creación). Por lo tanto al detectarlo genera un nuevo thread para que monitoree este nuevo nodo aplicando nuevamente este handshake. De esta manera se va realizando el conocimiento de todos los elementos que componen el replica-set.
Mientras se ejecuta este paso inicial de conocimiento de la topología pueden aparecer peticiones de ejecución de distintas operaciones. En principio, su ejecución está bloqueada pero dependiendo del avance de este proceso inicial de búsqueda puede ocurrir que empiece una ejecución concurrente de tareas.

Si las operaciones son de lectura y a su vez permiten ser respondidas por un nodo secundario, el cliente puede ejecutarla si ya estableció conexión con un nodo secundario.
Del mismo modo si ya localizó al nodo primario, puede ejecutar operaciones de lectura o escritura (sacando provecho del multi-threading) mientras se continúa la búsqueda total de la red.

AII.1.1.2 Single-thread

Al momento de iniciar un cliente, simplemente se crea una instancia del driver y se espera a la primer operación (cualquiera sea el tipo), para dar inicio al escaneo de los nodos que componen la red. Como no existe la posibilidad de una concurrencia, se ejecuta de forma serializada, en un orden aleatorio (por lo menos al inicio) el proceso de chequeo de estados de todos los servidores.
En el conocimiento de la topología, con el enfoque single-thread existe una variante en el handshake entre el cliente y los nodos. En la respuesta al saludo inicial “isMaster()”, de no ser el primario, se adjunta en la respuesta quién es/ o quién podría ser el nodo primario para los registro del nodo con quien se está conectando. El driver utiliza esta información para conectarse con el posible primario inmediatamente después, ya que poder encontrar el primario es la tarea más prioritaria. Luego de descubrir el nodo primario, el driver continúa hasta finalizar el reconocimiento de todos los nodos de la red y luego comienza a ejecutar la operación inicial (desencadenante de todo lo anterior).

AII.1.1.3 Híbrido

Los drivers híbridos implementan dos enfoques para el armado y el monitoreo de la topología. Single-thread(ST) y Pooled.

AII.1.1.4 Single-thread híbrido

Es una optimización de la opción anterior. El proceso de descubrimiento de la topología sigue siendo bloqueante, pero el escaneo ya no es serializado. Se crean N procesos ST que se conectan a la base de datos en paralelo a través de sockets no-bloqueantes que ejecutan un evento al momento de llegada de una respuesta para ser atendida por el proceso. Se chequea entonces el estado de cada miembro de manera concurrente y asincrónica. Se ejecuta el descubrimiento de red hasta que todos los proceso reciben su respuesta u ocurre un timeout del pedido asincrónico. El driver termina de actualizar su topología y permite entonces la ejecución de operaciones a la aplicación.
El beneficio de esta opción radica en el tiempo de espera que existe en el descubrimiento de la topología. Con el enfoque Single-thread el tiempo total está dado por la sumatoria del tiempo lineal del chequeo para cada nodo de la red, en cambio con el enfoque híbrido se reduce a la duración del chequeo que tuvo el

tiempo máximo de todos los nodos. Es ventajoso este enfoque en las redes compuestas por una gran cantidad de replica-sets o en las que exista un alto tiempo de latencia entre miembros.

AII.1.1.5 Pooled

Consiste en un proceso que se ejecuta en background para el monitoreo de la red. Al momento de empezar, el thread se conecta a todos los servidores de nuestra red. De manera asincrónica desde un socket no-bloqueante ejecuta en loop el handshake con los nodos (“isMaster()”). A medida que van obteniendo la respuesta, los sockets disparan eventos en el proceso que se encarga de actualizar la topología. A su vez al igual que en la técnica de Multi-threading, si encuentra un nuevo nodo genera una nueva conexión y agrega el evento para atender la respuesta en su listado de eventos de sockets a atender. Por otro lado, ni bien se encuentra un nodo que pueda ejecutar alguna de las operaciones que solicita la aplicación (secundario-primario) es posible hacerlo mientras el proceso continúa actualizando la topología en background. No es necesario que se termine de tener todos los nodos chequeados.

AII.1.1.6 Estado estable

Una vez que el driver termina el descubrimiento de toda la topología, pasa a la fase de estado estable. Consiste en chequeos periódicos a cada servidor de la red cada determinada cantidad de tiempo. Con este mecanismo se mantienen estadísticas como el tiempo de latencia entre nodos (utilizado más adelante para la delegación de operaciones) o para detectar un nuevo nodo que se sume a la red. También sirve para solucionar errores de manera proactiva ya que se pueden detectar fallas, como la baja de un nodo, y trabajar en consecuencia sin tener que esperar la ejecución de una operación por parte de la aplicación para empezar a actuar. Si bien se puede ver a esta lógica como un sobre-procesamiento, el enfoque se basa en la idea de estar siempre listo para responder a futuras demandas, reduciendo al mínimo los problemas innatos de la arquitectura que puedan surgir.
El tiempo de refresco cambia al igual que la manera de hacerlo ya que obviamente tenemos arquitecturas de drivers diferentes. En las soluciones multi-thread, dado el bajo costo que implica el escaneo de un thread ejecutando en background, se actualiza el estado cada diez segundos. En cambio con los drivers single-thread se bloquea la aplicación para hacer un nuevo escaneo (menos costoso que el de la etapa de descubrimiento) cada un minuto.

AII.1.1.7 Solución de fallas

En el caso de replica-set el foco está puesto sobre el nodo primario. Si no hay acceso a un nodo secundario se pierde un nodo de réplica, pero dependiendo de su

configuración, el mayor problema que implicaría para la estructura sería una posibilidad menos para responder operaciones de lectura. La base de datos seguiría pudiendo responder a todas las operaciones que nuestra aplicación requiera. En cambio el nodo primario es el único que puede recibir las operaciones de escritura. Con lo cual no poder accederlo implicaría estar imposibilitado de escritura e incluso de lectura (dependiendo la configuración).
Uno de los mayores beneficios que nos da la posibilidad de tener réplicas, es la alta disponibilidad de los datos. Si por cualquier motivo fuera inaccesible el nodo primario, rápidamente un nodo secundario ocupa ese lugar. Una configuración con muchos nodos secundarios puede ofrecer un estándar muy alto desde lo preventivo.

AII.2 Actualización de secundarios

AII.2.1 Idempotencia aplicada al Oplog para actualización de secundarios

El log es el elemento utilizado para mantener el estado actual de la base de datos. Por lo tanto ya sea que se aplique una o varias veces una operación sobre la estructura, cada operación en el log debe producir siempre el mismo resultado. Se dice entonces que cada operación debe ser idempotente. Es decir que debe tener la cualidad de producir siempre el mismo resultado recibiendo la misma entrada una o múltiples veces.
Al plantear una estructura distribuida, una parte de la funcionalidad está delegada en la capa de red. Los errores de red pueden ocurrir y tienen que ser contemplados por la base de datos. Por ejemplo: una operación podría ser efectivamente ejecutada en un nodo y a causa de errores de red, no llegaría la respuesta confirmando que se ejecutó la operación. La idea de tener un conjunto de operaciones que permitan obviar estos problemas se hace necesaria.
Para lograr siempre el mismo resultado aplicando la misma operación se deben analizar las distintas operaciones que provee MongoDB. Buscar, Insertar, Actualizar y Eliminar.

AII.2.1.1 Buscar

Son naturalmente idempotentes, recuperar información una vez causa el mismo efecto que hacerlo dos veces. Nunca se modifica la información.

AII.2.1.2 Insertar

Partiendo de un id único se puede asegurar que una vez que la operación se ejecuta correctamente, se generará una excepción por id duplicado en los intentos posteriores. La inserción es idempotente, pero se debe tener especial atención

sobre la condición de unicidad sobre el/los indexs que tenga la estructura de cada documento.
Otro dato no menor es que las operaciones de escritura simples en Mongo, siempre son atómicas. En las escrituras que modifican varios documentos es atómica al nivel de cada una de las escrituras particulares. Para estos casos se puede aislar una instrucción simple que afecta varios documentos a través del operador $isolated.

AII.2.1.3 Eliminar

Al igual que en la inserción partiendo de un id único, no se puede eliminar el mismo documento dos veces. Ejecutar más de una vez la operación devolverá el mismo resultado. Por otro lado, cuando se eliminan elementos en base a una condición, surge el problema de la “carrera de condiciones”. Podría ocurrir que al reintentar ejecutar la operación de eliminado, exista otro proceso que inserta datos que correspondan con la instrucción de eliminado.

AII.2.1.4 Actualizar

Setear un campo con un valor predeterminado es idempotente naturalmente. No va a cambiar el valor en base a la cantidad de ejecuciones.
El caso especial se da en las instrucciones de incremento ($inc). Si se ejecutan varias veces y no se valida que ya no se haya ejecutado la instrucción el valor cambia y deja de ser una instrucción idempotente. Para resolver estos casos, se divide la instrucción en dos pasos.
El primero es setear en un atributo “pendiente” un valor único (objectId), a través del operador $addToSet (es idempotente ya que sólo se ejecuta la primera vez). El segundo paso sería hacer un find por la instancia a editar con el _id y el token, y por último incrementar el contador y borrar el token como parte de la actualización del registro. Ambos pasos son atómicos, con lo cual se realizan con éxito o no se realizan. De esta manera se asegura la idempotencia en actualizaciones.

AII.2.1.5 Oplog

La estructura del documento no está completamente detallada y hoy en día se están haciendo algunos cambios, sin embargo se pueden analizar algunos puntos en común del formato, que se usa para ir haciendo un log de cada instrucción.

```
{
      "ts" : Timestamp(1395663575, 1),
      "h" : NumberLong("-5872498803080442915"),
      "v" : 2,
      "op" : "i",
      "ns" : "wiktory.items",
```

```
      "o" : {
        "_id" : ObjectId("533022d70d7e2c31d4490d22"),
        "author" : "JRR Hartley",
        "title" : "Flyfishing"
      }
}
```

Estructura básica de entrada en el oplog.

'TS': la fecha y hora (timestamp) muestra cuándo fue ejecutada la operación. Permite saber en un determinado momento cómo es la estructura y tener una secuencia clara de cómo se fueron ejecutando las operaciones. Es utilizado por otras réplicas a la hora de sincronizar datos.
'H': es un id único de operación. Asegura que cada operación pueda ser identificada unívocamente.
'V': es la versión del oplog, a través de la cual se puede saber el formato. El formato del oplog está cambiando actualmente, con lo cual MongoDB utiliza este número para saber cómo leer cada instrucción.
'OP': indica el tipo de operación a ejecutar. Los valores más usados son "i" para insertar, "u" para actualizar y "d" para eliminar. También existen "c" para los comandos que afectan la estructura de la base de datos en un alto nivel y "n" para cambios en la base de datos o colecciones que no impliquen un cambio en la información guardada.
'NS': especifíca el namespace donde se va a ejecutar la instrucción. En este ejemplo ("wiktory.items") en la base de datos wiktory, sobre la colección items.

Los siguientes campos van a depender del tipo de instrucción indicada previamente. En la inserción, se tiene el documento completo a agregar.

```
{
      "o" : {
        "_id" : ObjectId("533022d70d7e2c31d4490d22"),
        "author" : "JRR Hartley",
        "title" : "Flyfishing"
      }
}
```

Estructura básica de inserción en el oplog.

Mientras que en la actualización se tienen dos partes

```
{
      "o2" : {
        "_id" : ObjectId("533022d70d7e2c31d4490d22")
      },
      "o" : {
        "$set" : {
          "outofprint" : true
        }
      }
}
```

Estructura básica de actualización en el oplog.

“O2”: contiene la búsqueda del elemento sobre el que vamos a aplicar los cambios.
“O”: contiene los datos concretos con los valores que fueron actualizados.
Para el borrado solo está el _id del documento que fue borrado y el atributo “b” siempre seteado en true.

AII.3 Bibliografía

- https://emptysqua.re/blog/how-to-write-resilient-mongodb-applications/
- *https://emptysqua.re/blog/server-discovery-and-monitoring-in-pymongo-perl-and-c/*
- *https://www.mongodb.com/presentations/mongodb-drivers-and-high-availability-deep-dive*
- *https://www.compose.com/articles/the-mongodb-oplog-and-node-js/*

Apéndice III

AIII. Instructivo para la configuración de Sharded Cluster en MongoDB.

AIII.1 Máquinas virtuales

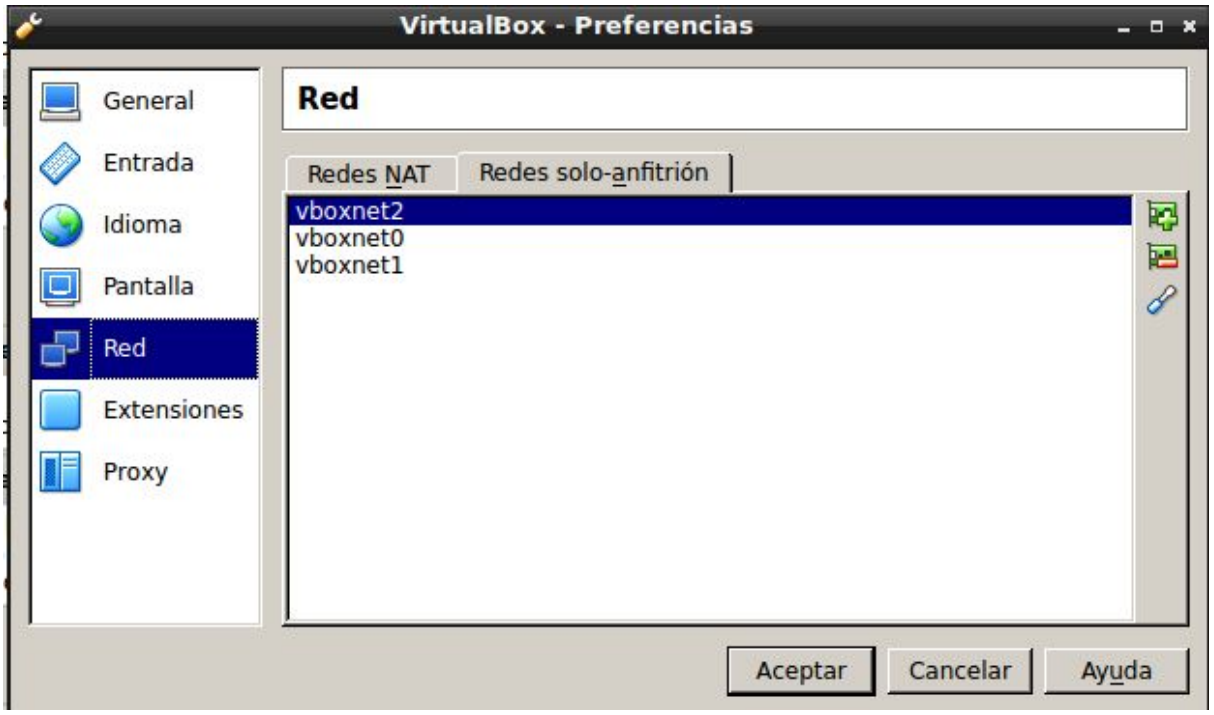
Shard	https://mega.nz/#!FxBAgDwY
Clave	!iq5nTtzx7SpDqVeI0tPcE1xvLyzVTzhXVxkfgZOQ8Cc
Router-Config	https://mega.nz/#!pkRSmbzQ
Clave	!J2nxPq7CiYKu5zb6GMkOzF90wMzxWKc1py5LS2vJEdA

Todas las máquinas virtuales tienen la misma configuración. El SO es una versión ligera de Ubuntu (Lubuntu 64bits) con mongoDB instalado. La única diferencia entre estas máquinas virtuales es el usuario root de cada sistema. En un Sharded Cluster existen 3 tipos de actores: Shard, Router y Config. Para lograr mayor legibilidad el usuario y contraseña por default de cada máquina es shard o routerconfig, según corresponda. A su vez, tienen por defecto la configuración correspondiente del archivo /etc/mongod.conf (archivo utilizado por Mongo para conocer qué actor se está ejecutando en un sharded cluster) en base a su rol específico.

AIII.2 Configuración

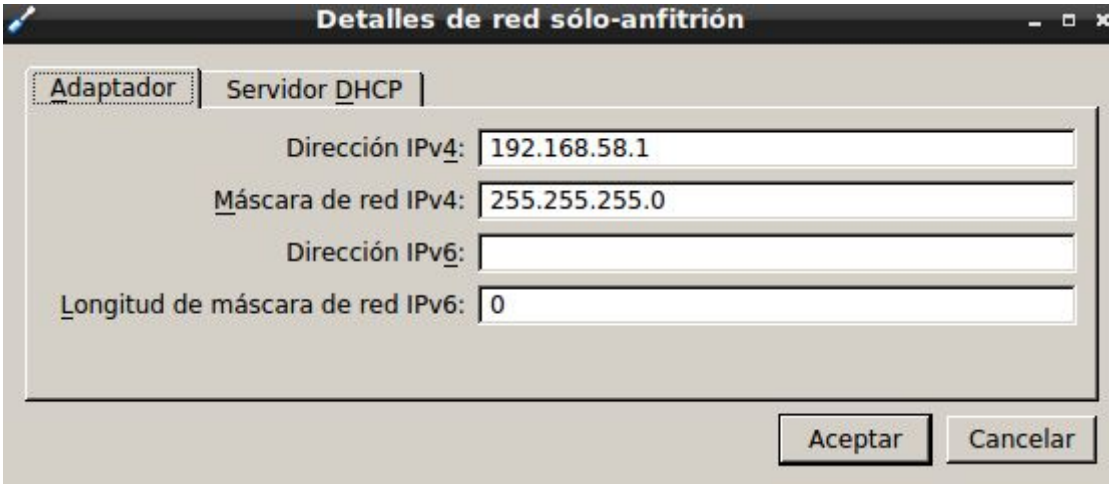
Acceder a guests desde IPs fijas

Es necesario configurar la red sobre la que va a correr el cluster.
En VirtualBox se debe seleccionar:
Archivo -> Preferencias -> Red -> Redes solo-anfitrión -> Agregar Red solo-afitrión



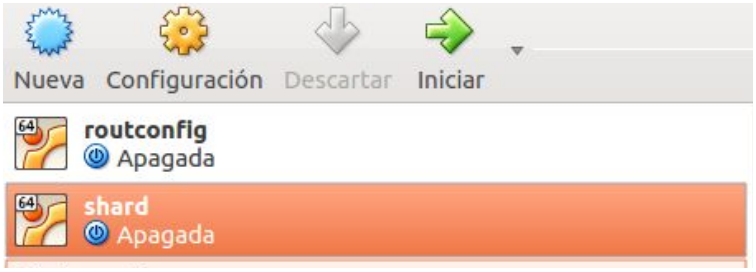
Luego editar la dirección IPv4 por 192.168.58.1

(La IP del adaptador podría ser otra. A fines prácticos se recomienda mantener esta IP)



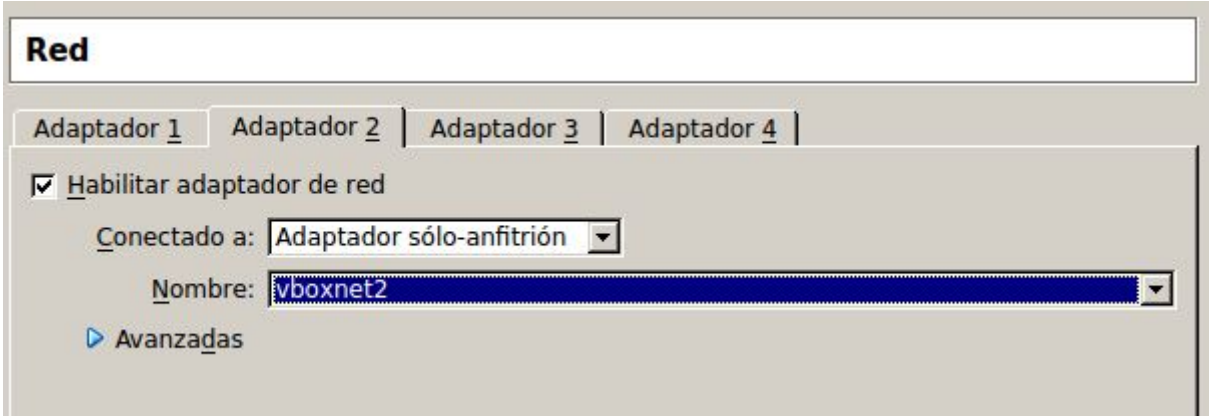
Luego de configurar el adaptador pasamos a la configuración de cada máquina virtual

(la máquina debe estar apagada). Se selecciona y luego a “Configuración”



Configuración->Red.
El adaptador 1, por defecto debería ser NAT. No se aplican cambios sobre esta pestaña, se selecciona la pestaña del 2do adaptador.

Seleccionar la opción Habilitar adaptador de red.
Conecto a: Adaptador sólo-anfitrión.
Nombre: Seleccionar el adaptador creado previamente en las preferencias de VirtualBox (en este caso vboxnet2)



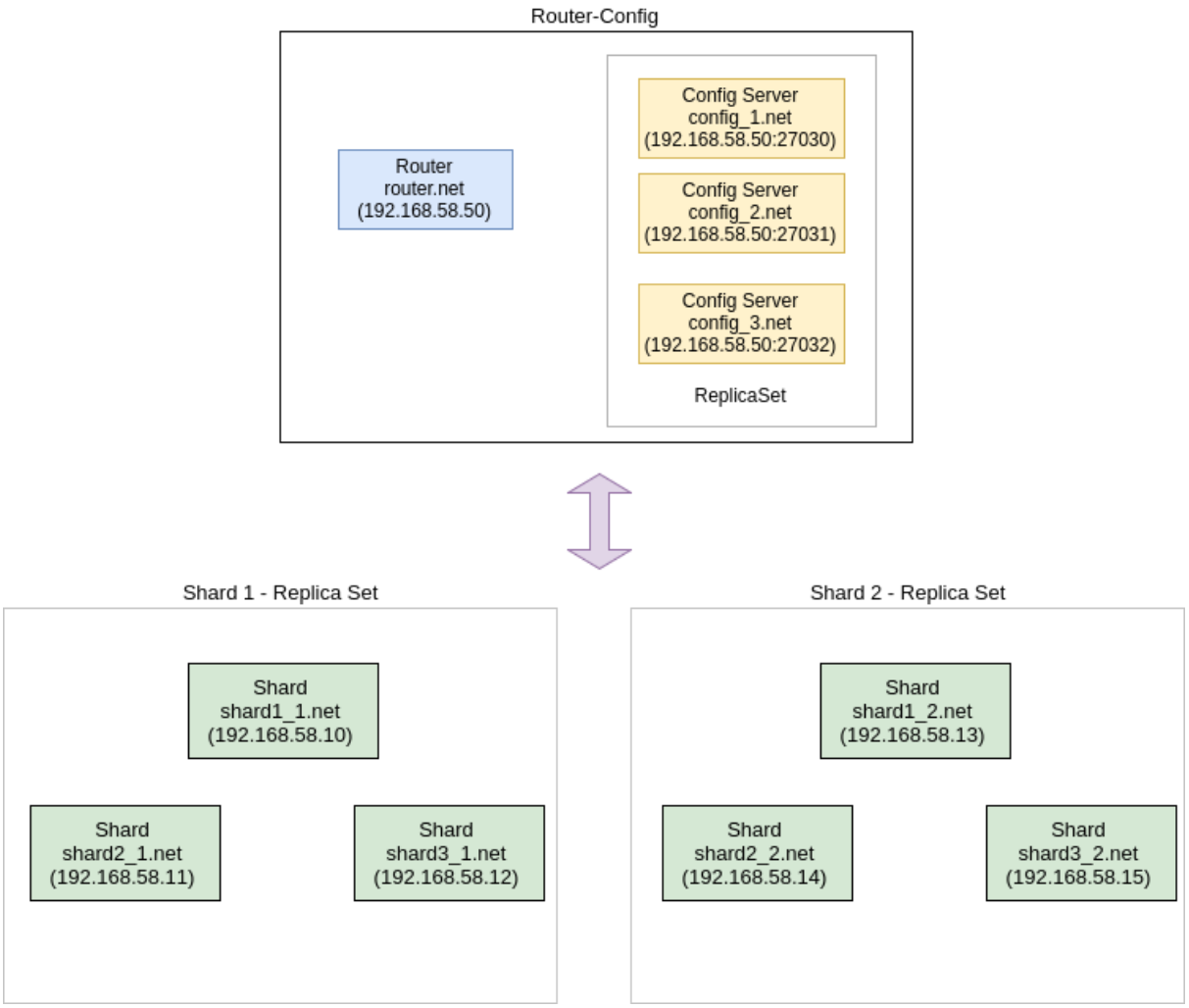
Esta configuración se debe repetir para todas las máquinas virtuales que se utilicen.

AIII.3 Sharded Cluster

El sharding en Mongo es implementado por tres componentes distintos. Cada parte cumple una función específica.

- Config Servers: El servidor de configuración es usado para guardar los metadatos que vinculan los datos solicitados con el fragmento o shard específico del cluster. En un entorno de producción debe haber replicados tres servidores de configuración, para garantizar la redundancia y la alta disponibilidad.
- Query Routers o Routers: Los enrutadores de consultas son los nodos a los que se conectan las aplicaciones externas. Son responsables de comunicarse con los Config Servers para averiguar dónde se almacenan los datos solicitados y luego devolver la información extraída de los shards apropiados.
- Shards: Los Shards son responsables de las operaciones reales de almacenamiento de datos. En entornos de producción, un único fragmento se compone generalmente de un conjunto de réplicas en lugar de una sola máquina. Esto es para garantizar que los datos seguirán siendo accesibles en caso de que un servidor de fragmentos primario se desconecte.

AIII.4 Estructura del proyecto



- 1 Config Server con Replica Set
- 1 Query Routers
- 2 Shard Servers con replica set c/u

El siguiente paso es configurar IPs y hostnames para cada nodo de la estructura. *Podría ser suficiente con las IPs, pero se recomienda referenciar a todos los nodos a través de un nombre dns, de manera tal que si cambian las IPs en algún momento, no haya que cambiar la configuración de la base de datos.*
Cada miembro del cluster debe poder conectarse a su vez a cualquier miembro del cluster. Esto incluye todos los shards y los servidores de configuración. Se debe asegurar la posibilidad de conexión entre todos los actores del cluster.

Para este propósito se debe tener la siguiente lista de subdominios:

- Config Servers
    - config_1.net (192.168.58.50:27030)
    - config_2.net (192.168.58.50:27031)
    - config_3.net (192.168.58.50:27032)
- Query Routers
    - router.net (192.168.58.50)
- Shard Servers
    - shard1_1.net (192.168.58.10)
    - shard1_2.net (192.168.58.11)
    - shard1_3.net (192.168.58.12)
    - shard2_1.net (192.168.58.13)
    - shard2_2.net (192.168.58.14)
    - shard2_3.net (192.168.58.15)

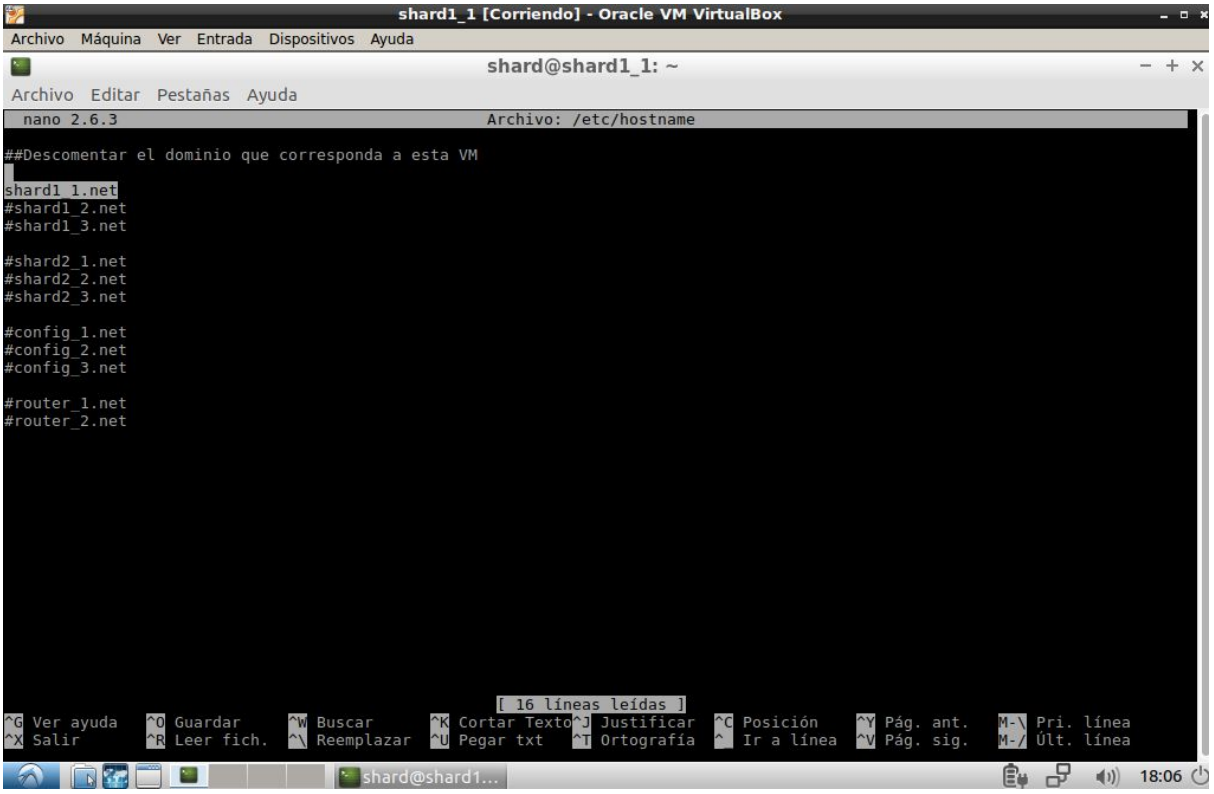
Teniendo en cuenta esta configuración se deben ejecutar algunos pasos previos antes de crear el sharded cluster.

1- Clonar las máquinas Shard.
Luego de bajar los dos tipos de máquinas virtuales del link de Mega. Se deben crear desde virtualbox tantos clones como son necesarios para la estructura. En el primer link no es necesario clonar la máquina ya que nos provee dos roles del sharded -Router y Config- en una sola VM. Sólo es necesario clonar las máquinas shard para representar la estructura planteada.
Para mayor legibilidad lo ideal es respetar la nomenclatura utilizada en los dominios previamente establecidos.
Por ej: La VM con dominio **shard2_1.net** se debe llamar **shard2_1 .**

2- Configurar Hostname e IP.
Una vez encendida la VM, se ejecuta por consola (Ctrl+Shift+t) el siguiente comando para editar el archivo hostname:

```
~$ sudo nano /etc/hostname
```

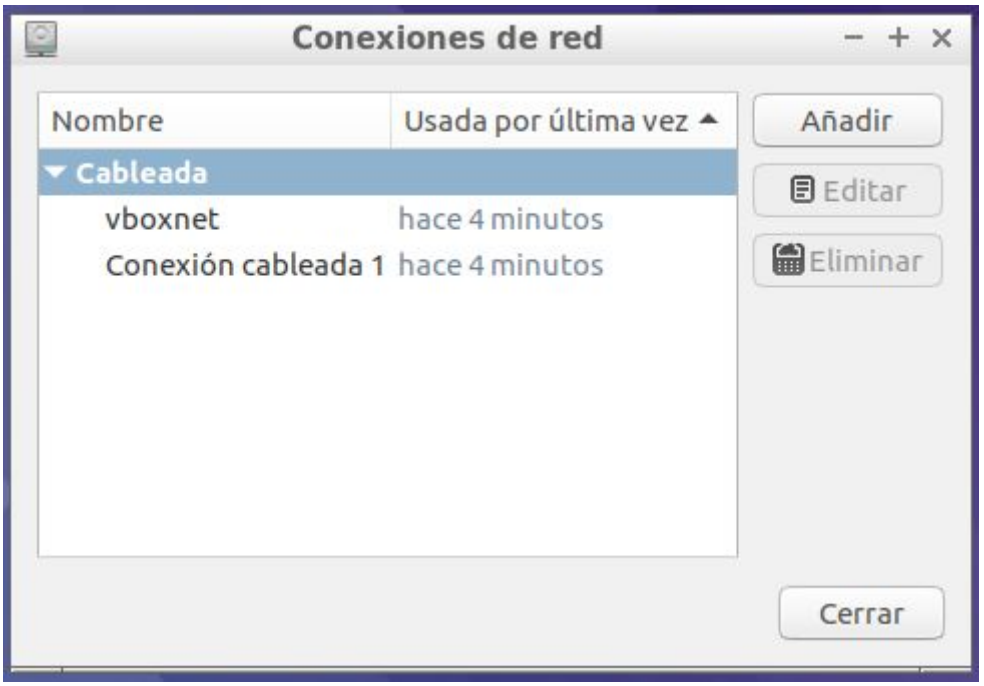
Se debe descomentar del archivo la línea cuyo nombre coincida con la VM



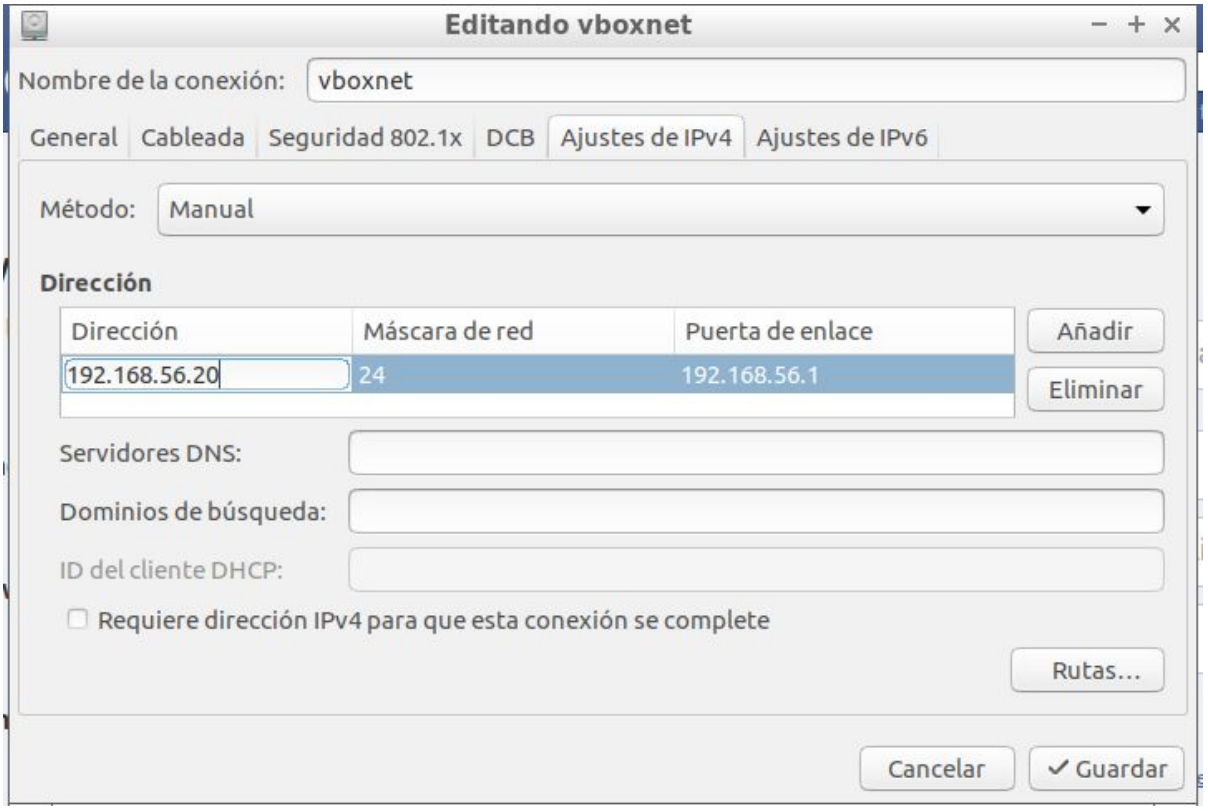
Guardamos (Ctrl+o, Enter) y cerramos (Ctrl+x).

Por último, desde la sección de “Edición de Conexiones de Red”

En caso de estar conectado. Hay que desconectar el adaptador “vboxnet”, previamente creado al comienzo del tutorial, para que tomen efecto los cambios.



Edición de conector, Ajustes de IPv4. Setear la IP que se corresponda para la VM. Usar como referencia el archivo /etc/hosts para ver qué IP corresponde en cada caso.



Se pueden setear otras IP, pero hay que tener en cuenta que la IP elegida para una VM determinada, cualquiera sea su rol, debe ser única dentro del cluster y que debe estar debidamente correspondida en el archivo /etc/hosts de cada una de las VMs.

Finalmente queda apagar la VM.

Repetir estos pasos por cada una de las VMs creadas en el cluster.

AIII.5 Crear el Sharded Cluster

Una vez montada la estructura y hecha la interconexión por dns entre todos los nodos, se procede a configurar el cluster. En este punto es recomendable encender las VM sin el modo gráfico, ya que se puede ejecutar todo perfectamente por la consola del SO host accediendo por ssh. Ya que se utilizan menos recursos, se puede asignar menos memoria RAM para que no sea tan costosa la virtualización en un entorno local. Es recomendable ir encendiendo las máquinas virtuales por tipo de actor que representan ya que al comienzo no es necesario que estén todas las máquinas encendidas.

AIII.5.1 Config y Router Server

La VM routconfig ya tiene preparados los roles de Config y Router. Incluso tiene preconfigurado los daemons que ejecutan MongoDB en modo background. Se utilizará esta VM más adelante.

AIII.5.2 Shards

Iniciamos las 3 máquinas de alguno de los dos conjuntos shard que pertenecen al mismo ReplicaSet -en modo sin pantalla- y accedemos por ssh

```
~$ ssh shard@192.168.58.1X
```

Una vez dentro de la consola se deben ejecutar 3 pasos:
1- Editar el archivo /etc/mongod.conf
Se deben editar el hostname de la opción bindIp, según corresponda en todas las VMs:

```
# network interfaces
net:
  port: 27017
  bindIp: shard1_1.net
```

Tener en cuenta que este documento posteriormente será parseado como yml, con lo cual hay que identar con especial atención los 2 espacios en el salto de línea.

Para el primer shard, no va a ser necesario editar nada más que el bindIp en sus 3 Replica Sets, pero con el segundo conjunto shard se debe modificar el replSetName por **shard2rs**.

```
replication:
  replSetName: shard2rs
```

Ejecutar

```
~$ sudo rm /var/lib/mongodb/mongod.lock
~$ sudo systemctl restart mongod.service
```

Limpiamos los logs antiguos y reiniciamos el daemon de MongoDB.

Se debe ejecutar el paso 1 en todos los shards que pertenecen al mismo ReplicaSet, antes de pasar a la siguiente instrucción.

2- Desde la consola de la primer VM (shard1_1) ejecutar:

```
~$ mongo --host shard1_1.net
```

Una vez dentro del shell de mongo ejecutar

```
> rs.initiate()
> rs.add( { _id: 1, host: "shard1_2.net" } )
> rs.add( { _id: 2, host: "shard1_3.net" } )
```

Cabe destacar que los dominios también deben cambiar cuando se trabaja sobre el segundo grupo de shards.

3- Chequeo del funcionamiento

Para chequear que el replicaSet de mongo funcione correctamente, basta con loguearse por consola en algún nodo slave (esto se puede ejecutar desde cualquier VM que pertenezca al ReplicaSet. Por ejemplo, en este caso, no es necesario estar en la consola de comandos de la VM shard1_2. Puede ser desde la de shard1_1)

```
~$ mongo --host shard1_2.net
```

Luego ejecutar

```
> rs.status()
```

En la respuesta se detalla el estado actual del cluster. Se listan los miembros que participan, el rol que cumplen en el replicaSet, su hostname y su estado.
Si todos se encuentran en estado 1 para el primary y 2 para los secondary la configuración fue exitosa.

Este paso debe ser repetido tantas veces como actores shard haya en la estructura.

AIII.6 Inicialización de Sharding

Una vez configurada toda la estructura sobre la que correrá el sharded cluster. Solo resta inicializar el funcionamiento del sharding. Para esto hay que encender todas las VMs.
Las consultas de lectura/escritura/configuración a base de datos se hacen a través de las consolas de los routers. Desde la perspectiva del cliente que consulta la base de datos, la conexión sigue siendo la misma. Es indistinto si la consulta es a una estructura sharded o a un único servidor Mongo, ya que en el caso sharded los routers en combinación con los nodos config, tienen la funcionalidad necesaria para simular una única base de datos.
Por lo tanto, para interactuar con el sharded hay que tener conexión con una instancia mongos (Router)

A través de ssh o desde el entorno visual, se debe acceder a la VM de routconfig y ejecutar en consola para conectarse al shell de mongos:

```
~$ mongo --host ip_routconfig
```

Lo primero que hay que hacer es agregar los distintos conjuntos de shards que tiene nuestra estructura.

```
mongos> sh.addShard('shard1rs/shard1_1.net')
mongos> sh.addShard('shard2rs/shard2_1.net')
```

Una vez en el shell de Mongo hay que indicar sobre qué base de datos se va a permitir hacer el sharding.

```
mongos> sh.enableSharding('tesina')
```

Se puede chequear si se habilitó el sharding correctamente mediante

```
mongos> use config
mongos> db.databases.find()
```

Para habilitar la opción de shard sobre una colección, es obligatorio que previamente la base de datos esté habilitada para sharding.
Es necesario, también, elegir una clave sobre la colección, que será utilizada en el futuro para direccionar los nodos a través del sharding.

```
mongos> use tesina
mongos> db.archivos.ensureIndex( { _id : "hashed" } )
mongos> sh.shardCollection('tesina.archivos',{_id:"hashed"})
```

AIII.7 Prueba de Sharding en un caso concreto

En el repositorio https://github.com/psoldier/mongo-sharded-cluster se encuentra una simple aplicación web que permitirá cargar, buscar y registrar los tiempos de respuesta en el manejo de archivos con esta estructura.

AIII.8 Bibliografía

Lubuntu (64bit para vm)
https://help.ubuntu.com/community/Lubuntu/GetLubuntu

Production Configuration
https://docs.mongodb.com/manual/core/sharded-cluster-components/#production-configuration

Development Configuration
https://docs.mongodb.com/manual/core/sharded-cluster-components/#development-configuration

Eficiencia
https://docs.mongodb.com/manual/administration/production-notes/#performance-monitoring