# UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

*La Universidad Católica de Loja*

## ÁREA TÉCNICA

TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Análisis e implementación de soluciones con base de datos NoSQL para el manejo de Big Data, aplicando técnicas de escalabilidad y tolerancia a fallos

TRABAJO DE TITULACIÓN.

**AUTORA**: Sigcho González Fabiola Jacqueline

**DIRECTOR**: Elizalde Solano, René Ronaldo, Mgs.

LOJA – ECUADOR

2017



*Septiembre, 2017*

## APROBACIÓN DEL DIRECTOR DEL TRABAJO DE TITULACIÓN

Magister.
René Rolando Elizalde Solano
**DOCENTE DE LA TITULACIÓN**

De mi consideración:

El presente trabajo de titulación: Análisis e implementación de soluciones con base de datos NoSQL para el manejo de Big Data, aplicando técnicas de escalabilidad y tolerancia a fallos realizado por Sigcho González Fabiola Jacqueline, ha sido orientado y revisado durante su ejecución, por cuanto se aprueba la presentación del mismo.

Loja, septiembre del 2017

f)…….....

## DECLARACIÓN DE AUTORÍA Y CESIÓN DE DERECHOS

"Yo Sigcho González Fabiola Jacqueline declaro ser autora del presente trabajo de titulación: Análisis e implementación de soluciones con base de datos NoSQL para el manejo de BigData, aplicando técnicas de escalabilidad y tolerancia a fallos, de la Titulación Sistemas Informáticos y Computación, siendo René Rolando Elizalde Solano director del presente trabajo; y eximo expresamente a la Universidad Técnica Particular de Loja y a sus representantes legales de posibles reclamos o acciones legales. Además certifico que las ideas, conceptos, procedimientos y resultados vertidos en el presente trabajo investigativo, son de mi exclusiva responsabilidad.

Adicionalmente declaro conocer y aceptar la disposición del Art. 67 del Estatuto Orgánico de la Universidad Técnica Particular de Loja que en su parte pertinente textualmente dice: “Forman parte del patrimonio de la Universidad la propiedad intelectual de investigaciones, trabajos científicos o técnicos y tesis de grado que se realicen a través, o con el apoyo financiero, académico o institucional (operativo) de la Universidad”

f)…….....
Autor Fabiola Jacqueline Sigcho González
Cédula 1104975022

## DEDICATORIA

Dedico este trabajo principalmente a Dios, por haberme dado la fortaleza en los momentos difíciles y haberme permitido llegar hasta este momento tan importante de mi formación profesional.

A mis padres Lorena y José Antonio por ser las personas que me han acompañado durante todo mi trayecto estudiantil y de vida, por sus consejos y apoyo incondicional.

A mis hermanos Dennis y Rosemary, y a mi sobrino Lucas, les dedicó este trabajo como símbolo de superación.

A mi abuelita María por su cariño infinito y mi abuelito Alfredo que aunque ya no está entre nosotros, siento que me cuida y me protege siempre.

A toda mi familia, por haberme brindado su apoyo y por compartir conmigo buenos y malos momentos.

***Jacqueline***

## AGRADECIMIENTO

A Dios por guiar mi camino y haberme dado fuerzas para superar obstáculos y dificultades a lo largo de mi vida.

Agradezco la confianza y el apoyo incondicional de mis padres para alcanzar esta meta. A mis hermanos, sobrinos y mis mascotas por haberme brindado muchos momentos de alegría. Agradezco especialmente a todos los docentes de la Titulación de Ingeniería en Sistemas, que fueron parte de mi formación académica, por compartir sus conocimientos y experiencias que permitieron formarme como profesional.

Finalmente, agradezco al Ing. René Elizalde y al Ing. Juan Carlos Morocho por su confianza y asesoramiento para desarrollar el presente proyecto, y por brindarme las facilidades para poder cumplir con el trabajo en el laboratorio de Tecnologías avanzadas de la Web.

***Jacqueline***

## ÍNDICE DE CONTENIDOS

## ÍNDICE DE FIGURAS

## ÍNDICE DE TABLAS

## RESUMEN

Hoy en día el término Big Data ha adquirido una especial importancia, y actualmente el gran volumen de datos exige diseñar nuevas infraestructuras escalables y tolerantes a fallos, que aseguren un alto rendimiento y disponibilidad. El presente trabajo se orienta en analizar y medir el rendimiento de soluciones con bases de datos NoSQL para la gestión de Big Data e identificar aquellas características más relevantes y sus beneficios. Las bases de datos elegidas MongoDB, CouchDB y Couchbase apoyan la filosofía de escalabilidad, proporcionan tolerancia a fallos y permiten procesar datos con formatos no estructurados. Se definió el diseño metodológico para realizar la implementación y configuración de las bases de datos en un sistema distribuido en la infraestructura disponible del laboratorio de Tecnologías Avanzadas de la Web de la Universidad Técnica Particular de Loja. Se ha realizado una comparación entre las bases de datos a través de la evaluación del rendimiento del sistema. Y finalmente se presentan los resultados concluyentes sobre las bases de datos y su desempeño en distintos escenarios.



**PALABRAS CLAVES:** NoSQL, Couchbase, MongoDB, CouchDB, fragmentación, replicación, big data, escalabilidad, análisis.

**ABSTRACT**

Nowadays the term Big Data has acquired a special importance, and at present the large volume of information volume requires designing new scalable and fault tolerant infrastructures to ensure high performance and availability. This paper focuses on analyzing and measuring the performance of solutions with databases NoSQL for the management of Big Data and identifying the most relevant characteristics and their benefits. The databases chosen MongoDB, CouchDB and Couchbase support the philosophy of scalability, provide fault tolerance and allow processing data with unstructured formats. It defined the methodological design for performing the implementation and configuration of databases in a distributed system in the infrastructure available of the laboratory of Advanced Technologies of the Web of the Technical University of Loja. A comparison has been realized between the databases across the evaluation of the performance of the system. And finally we presents the conclusive results on the databases and their performance in different scenarios.

.

**KEYWORDS:** NoSQL, Couchbase, MongoDB, CouchDB, sharding, replication, big data, scalability, analysis.

## INTRODUCCIÓN

En la actualidad las bases de datos relacionales (RDBMS[1]) son utilizadas por la mayoría de sistemas que realizan transacciones que requieren gran precisión, en tales sistemas los datos se organizan y almacenan en tablas, y usan el lenguaje de consulta estructurado SQL (por las siglas en inglés *Structured Query Language*) para el manejo y gestión de la información. El modelo relacional es actualmente el más utilizado para implementar bases de datos ya planificadas, y por más de veinte años ha dominado el escenario del almacenamiento de la información (Vazques & Sotolongo, 2013). Sin embargo, en la última década una nueva estructura de almacenamiento de datos ha ganado interés entre la comunidad tecnológica, las denominadas bases de datos no relacionales o NoSQL, debido a la escalabilidad y velocidad de sus tiempos de respuestas (Leavitt, 2010) y el almacenamiento de datos tanto estructurados como no estructurados, proporcionando una mayor versatilidad que los sistemas relacionales pues no es necesario definir previamente esquemas o estructuras.

Con el surgimiento de la web, los datos crecen diariamente y con ello se generan enormes cantidades de información no estructurada que se origina de diversas fuentes tales como la web, redes sociales, imágenes, vídeos, correos, foros, entre otros., en este sentido los sistemas tradicionales o relacionales se convierten en insuficientes para el procesamiento y análisis de enormes cantidades de información y no soportan el almacenamiento de datos no estructurados. Las cantidades masivas de información hacen referencia al término conocido como Big Data que se define como “grandes volúmenes de información que se acumulan con el tiempo y resultan imposible de manipular y gestionar utilizando herramientas tradicionales de gestión de bases de datos” (Zdnet, 2010). Por consiguiente, de las limitaciones que presentaban las bases de datos relacionales aparecieron las bases de datos conocidas como NoSQL, las cuales ofrecen una mayor capacidad de almacenamiento, alta escalabilidad y tolerancia a fallos.

Las bases de datos normalmente están implementadas en un solo servidor, lo cual es una desventaja en términos de escalabilidad y alta disponibilidad, ante esta necesidad y las necesidades actuales nace la iniciativa de desarrollar el presente Trabajo de Titulación que consiste en el análisis e implementación de soluciones con bases de datos NoSQL para el manejo de Big Data, aplicando técnicas de escalabilidad y tolerancia a fallos, cuyo objetivo principal es la implementación de bases de datos NoSQL en un ambiente distribuido bajo una

[1] RBMS (por sus siglas en inglés Relational Database Management System) consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. La colección de datos, normalmente denominada base de datos (Silberschatz, Korth, & Sudarshan, S. (Instituto Indio de Tecnología, 2002).

arquitectura escalable y tolerante a fallos, para el manejo de grandes volúmenes de información. En un sistema distribuido, la escalabilidad hace referencia a la capacidad de incluir varios servidores para que la información sea procesada eficientemente y la tolerancia a fallos se refiere a que incluso al producirse un fallo en uno de los servidores, las aplicaciones clientes puedan tener acceso a los datos proporcionando de tal manera una alta disponibilidad de la información.

La metodología que se aplica en el proyecto, adopta un enfoque iterativo e incremental. Para lo cual se establecen hitos y entregables que son revisados y supervisados por el director del Trabajo de Titulación

El presente trabajo está organizado en cuatro capítulos: En el Capítulo I uno se presenta una introducción a las bases de datos NoSQL sus orígenes y propósitos, con la finalidad de entender claramente el problema y así plantear una solución. Se define el alcance, los objetivos, las limitaciones del trabajo y asimismo se indican algunas abreviaciones para una mejor comprensión del proyecto.

El Capítulo II se enfoca en realizar un análisis de los diferentes productos de bases de datos NoSQL que están en auge y se plantea una propuesta de solución con las bases de datos que permitan su implementación sobre un entorno distribuido.

El Capítulo III contempla la implementación de las bases de datos NoSQL seleccionadas, luego del análisis realizado en el Capítulo II. Para obtener una comparación equilibrada de rendimiento de las bases de datos las pruebas se han realizado sobre el mismo conjunto de datos. Este capítulo además describe el ciclo de desarrollo de la implementación de las bases de datos, que incluye la definición de la arquitectura y el proceso de configuración en los servidores.

El Capítulo IV presenta la evaluación y análisis de resultados obtenidos a partir de la definición de determinados escenarios de prueba y con los parámetros de evaluación: Tiempo de respuesta y consumo de recursos. Finalmente, se concluye con el desarrollo de las conclusiones y recomendaciones en base a los objetivos propuestos en el trabajo.

# CAPÍTULO I

# VISIONAMIENTO

En este capítulo se presenta una introducción a las bases de datos NoSQL, se explica el Teorema de CAP; creado con la idea de que en un entorno distribuido no es posible garantizar las características de consistencia, disponibilidad y tolerancia a la partición simultáneamente, además se describe los diferentes tipos de almacenamiento y características que ofrecen los sistemas NoSQL. Se abordan estas temáticas con el fin de describir aspectos importantes que permitan comprender con claridad el problema que plantea el presente Trabajo de Titulación.

En el apartado 1.1 se indican las generalidades que incluye la problemática, donde se describe cómo se llevan a cabo las implementaciones de las bases de datos relacionales y la habilidad que proporcionan las bases de datos NoSQL para implementarse en un entorno distribuido; se plantea el alcance que define y delimita el desarrollo del presente trabajo; y se propone los objetivos que se pretende lograr. En el apartado 1.2 se presenta una introducción a las bases de datos NoSQL; en el apartado 1.3 se indican algunos de los beneficios que proveen las bases de datos NoSQL; en el apartado 1.4 se indican las limitaciones, que es aquello que no se llevará a cabo en el presenta trabajo; y por último en el apartado 1.5 se presenta una tabla que incluye las abreviaciones utilizadas en el presente proyecto.

## 1.1. Generalidades

### 1.1.1. Problemática.

Las bases de datos NoSQL se diseñaron para almacenar y gestionar grandes volúmenes de información de forma escalable y proporcionando alto desempeño para responder con rapidez a las necesidades de sus usuarios. Las nuevas necesidades de la época y el crecimiento acelerado de la información estructurada, semiestructurada y no estructurada que provienen de la web u otras fuentes exigen diseñar nuevas infraestructuras que proporcionen escalabilidad y tolerancia a fallos. Uno de los principios fundamentales de las bases de datos no relacionales es la distribución, la cual hace énfasis en la habilidad de replicar y distribuir datos sobre múltiples servidores. A diferencia de las bases de datos relacionales que se diseñaron para ser centralizadas en un solo servidor, las bases de datos NoSQL se crearon con el fin de distribuir la información en varios servidores para obtener mayor velocidad de procesamiento. Por esta razón, el presente trabajo pretende utilizar esta característica; para definir un procedimiento de instalación de bases de datos NoSQL en un ambiente distribuido que asegure alta disponibilidad y tolerancia a fallos.

### 1.1.2. Alcance.

El presente Trabajo de Titulación pretende analizar diferentes bases de datos NoSQL e identificar las que proporcionen la posibilidad de almacenar información en un ambiente distribuido, así como su implementación sujeta a las propiedades de escalabilidad y tolerancia

a fallos. Y definir un proceso de instalación y configuración reutilizable que refleje las características y ventajas de utilizar productos NoSQL.

### 1.1.3. Objetivos.

#### 1.1.3.1. *General.*

Definir un procedimiento para la implementación de una base de datos NoSQL sobre un ambiente distribuido, que asegure escalabilidad y tolerancia a fallos para el manejo de Big Data.

#### 1.1.3.2. *Específicos.*

- Aplicar técnicas de escalabilidad en un ambiente distribuido de bases de datos NoSQL para gestionar grandes volúmenes de información.

- Asegurar la disponibilidad de la información contenida en bases de datos NoSQL, aplicando mecanismos de tolerancia a fallos.

- Construir una aplicación web que haga uso de una base de datos NoSQL bajo una arquitectura escalable y tolerante a fallos.

## 1.2. Introducción a las bases de datos NoSQL

El término NoSQL significa *not only SQL* y se usa para agrupar sistemas de bases de datos que no siguen el modelo tradicional de RDBMS (Bender, Deco, González Sanabria, Hallo, & Ponce Gallegos, 2014). En los últimos años, el desarrollo de las tecnologías NoSQL ha sido fuertemente motivado por el surgimiento de un nuevo concepto denominado Big Data. El término BigData se asocia a la generación de cantidades masivas de información que provienen de la web, las redes sociales, páginas web, vídeos, entre otras. Dans (2011) citado por Camargo-Vega, Camargo-Ortega, y Joyanes-Aguilar (2015) define Big Data como “el tratamiento y análisis de enormes repositorios de datos, tan des-proporcionadamente grandes que resulta imposible tratarlos con las herramientas de bases de datos y analíticas convencionales”.

Las bases de datos NoSQL surgieron con el fin de dar solución a los problemas de escalabilidad y disponibilidad de los sistemas relacionales; y debido a las nuevas necesidades de la época.

> Los nuevos requerimientos en la época actual incluyen: disponibilidad total, tolerancia a fallos, almacenamiento de penta bytes de información distribuida en miles de servidores, buen desempeño sin límite de nodos con escalabilidad horizontal, manejo de grandes cantidades de datos estructurados y no estructurados, nuevos métodos de consulta, libertad para manejar nuevos esquemas (Bender et al., 2014, p.103).

Google, Facebook y Twitter son algunas de las compañías que se enfrentan al crecimiento exponencial de los datos no estructurados, así pues descubrieron las limitaciones de los sistemas relacionales al gestionar enormes volúmenes de datos, por lo que crearon sus propios sistemas. Por ejemplo, Facebook y Twitter usan Apache Cassandra un proyecto Open Source de Google Code, y Amazon desarrolló DynamoDB.

### 1.2.1. Teorema de CAP.

El Teorema de CAP es también llamado el Teorema de Brewer, en reconocimiento a su creador el científico Eric Brewer, quien en el año 2000 durante el Simposio de Principios de Computación Distribuida propuso la idea de que en un entorno distribuido no es posible garantizar de forma simultáneamente las propiedades de consistencia, alta disponibilidad y tolerancia a partición. Un entorno distribuido se define como “una colección de computadores autónomos que aparecen a sus usuarios como un único sistema coherente” (Tanenbaum & Van Steen, 2007). A continuación se describe cada una de las propiedades de Teorema.

**Consistencia:** En un sistema distribuido, todos los nodos ven la misma versión de datos, es decir, que si existen actualizaciones en los datos estos deben reflejarse en el resto de nodos. Esta propiedad puede conseguirse replicando la información en cada nodo que conforma el sistema distribuido.

**Disponibilidad:** Hace referencia a la garantía de que cada solicitud realizada por los usuarios recibe una respuesta satisfactoria, aunque un nodo dentro del sistema distribuido ha fallado.

**Tolerancia a particiones:** El sistema seguirá funcionando aunque un servidor este fuera de servicio, esto significa que el particionamiento de un nodo no debe afectar al resto de nodos y las solicitudes deben ser respondidas sin que el usuario pueda notar que un servidor ha fallado. Esta propiedad se relaciona con la propiedad de disponibilidad.

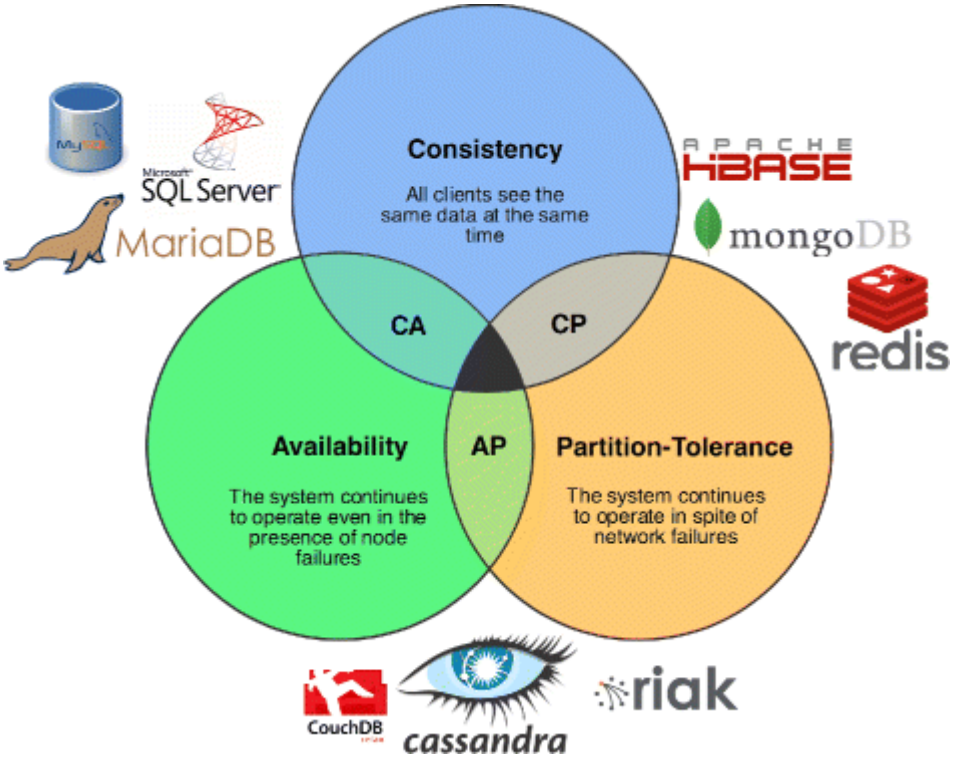



Figura 1. Propiedades del Teorema de CAP
Fuente: journalofbigdata.springeropen.com/articles/10.1186/s40537-015-0025-0
Elaboración: Lourenço, Cabral, Carreiro, Vieira, y Bernardino (2015)

El Teorema de CAP afirma que para ganar velocidad en un entorno distribuido se debe sacrificar al menos una de las propiedades antes mencionadas. La Figura 1 muestra las propiedades del Teorema de CAP y las posibles combinaciones que pueden relacionarse. A continuación se describe las combinaciones de las propiedades CAP:

**CP:** El sistema es consistente y tolerante a fallos aun si se pierde la comunicación entre nodos, pero no se asegura que haya disponibilidad total de la información.

**AP:** Garantiza alta disponibilidad de los datos y tolerancia a particiones, es decir, el sistema siempre responderá a las peticiones de los usuarios aunque existan problemas en las comunicaciones, pero existirá inconsistencias pues todos los nodos podrán no tener la misma versión de datos.

**CA:** Las peticiones de los usuarios siempre serán respondidas por el sistema y la información será consistente. Sin embargo, no se permite ningún fallo en las comunicaciones de los nodos del sistema.

### 1.2.2. Tipos de almacenamiento NoSQL.

En las bases de datos NoSQL, el almacenamiento de los datos es distinto a los sistemas relacionales pues no hay una estructura fija, así pues existe una categorización propia de las

diferentes formas de almacenamiento de información. A continuación se describe los tipos de almacenamiento de los sistemas NoSQL.

#### *1.2.2.1. Almacenamiento clave-valor.*

Son las bases de datos NoSQL más sencillas, se conforman por una clave única y un valor que permiten el acceso a los datos de forma más rápida. Este tipo de almacenamiento asegura velocidad en el procesamiento de datos, buena escalabilidad, tolerancia a fallo y proporcionan la posibilidad de tener los datos en un sistema distribuido. Romero, Sanabria y Cuervo (2012) recomiendan usar este tipo de almacenamiento "en casos donde se necesita velocidad en las consultas o se tienen muchos datos con estructura simple que requieren ser procesados una y otra vez y tienen valores cambiantes, por ejemplo, listas de los más vendidos" (p.27).

| CLAVE | VALOR |
|---|---|
| 1 | {"nombre": "Jhon", "areas": "Programación, Matemáticas"} |
| 2 | {"nombre": "Joe", "areas": "Alemán, Ingles, Mandarín, Francés"} |
| 3 | {"nombre": "Alicia", "areas": "Literatura española, Literatura inglesa"} |

Figura 2. Almacenamiento de base de datos clave-valor
Fuente: Elaboración propia

La Figura 2 es la representación gráfica del formato clave-valor, en donde hay una clave "1" que tiene asociado los valores nombre y áreas; y un valor que corresponde a las áreas {"Alemán, Inglés, Mandarín, Francés"} en las que tiene conocimiento el estudiante "Joe". Algunas bases de datos que usan este tipo de almacenamiento son Amazon Dynamo, Riak y Redis.

#### *1.2.2.2. Almacenamiento en documentos.*

Este tipo de almacenamiento es libre de esquemas y fue diseñado para manejar y almacenar la información en documentos. A diferencia del almacenamiento clave-valor este se conforma por una clave única para cada registro y un documento, donde el objeto documento generalmente utiliza formatos específicos como XML[2], JSON[3] o BSON[4] que contienen colecciones clave-valor con valores anidados vinculados con cada clave. En la Figura 3 se muestra la representación gráfica de las bases de datos documentales. El documento está especificado con formato JSON, en donde la clave es el nombre del campo por ejemplo

[2] XML (por sus siglas en ingles Extensible Markup Language) es un formato simple y muy extensible diseñado para cumplir con los retos de la publicación electrónica a gran escala e intercambio de datos en la web (w3.org, 2016).
[3] JSON (JavaScript Object Notation) es una sintaxis para almacenar e intercambiar datos (w3schools.com, 2012).
[4] BSON es una serialización binaria codificada de documentos JSON (Bsonspec.org, 2013).

“name” y el valor son los datos “Ale C”; y la clave única es el número entero identificado por el campo _id.

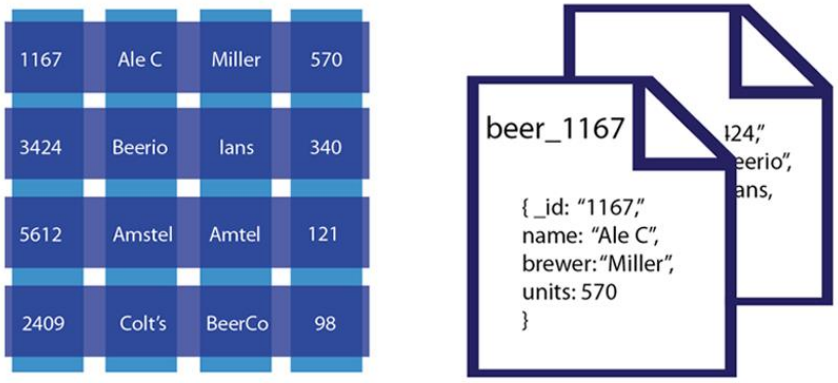



Figura 3. Almacenamiento del tipo de base de datos documentales
Fuente: docs.Couchbase.com/developer/dev-guide-3.0/compare-docs-vs-relational.html
Elaboración: Couchbase (2017)

Entre los principales representantes de este tipo de almacenes de información se destacan MongoDB y CouchDB.

#### *1.2.2.3. Orientado a columnas.*

El almacenamiento de datos orientado a columnas tiene una estructura similar a las clásicas bases de datos relacionales, la diferencia es que almacena su información en columnas, y no en filas. Según Romero *et al*. (2012) una desventaja de este tipo de almacenamiento es que “los datos de una entidad se esparcen entre varias columnas; por lo tanto, la inserción y la actualización o lectura del contenido completo de una entidad puede ser más lenta y compleja que en una base de datos relacional”. Por tanto, se recomienda usar el almacenamiento en columnas para grandes volúmenes de datos donde la lectura es más frecuente que la escritura. La Figura 4 muestra el esquema de la familia de las bases de datos de columna.

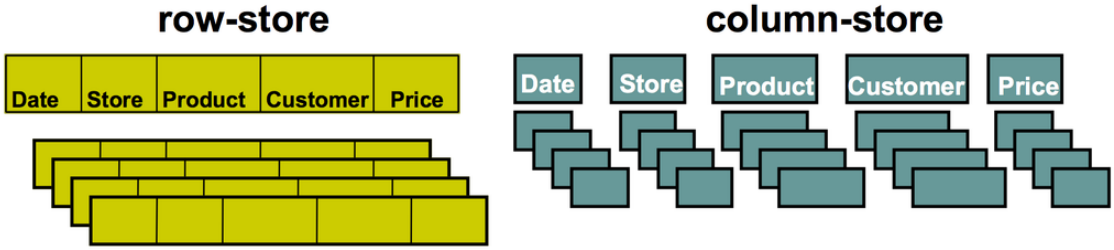



Figura 4. Almacenamiento de base de datos *orientados* a columnas
Fuente: jeuazarru.com/wp-content/uploads/2014/10/dbco.pdf
Elaboración: Adrián Garcete (2012)

Entre las bases de datos orientadas a columnas se encuentran BigTable, Cassandra y Hbase.

#### *1.2.2.4. Grafos.*

Las bases de datos de grafos representan una categoría especial de las bases de datos NoSQL, en donde las relaciones están representadas por grafos. Una base de datos de grafos está formada de nodos (*objetos*), aristas (*representan las relaciones entre los objetos*) y propiedades (*atributos del objeto expresado como pares clave-valor*). Moniruzzaman y Hossain (2013) afirman que "las bases de datos de grafos son útiles cuando se está más interesado en las relaciones entre los datos que los propios datos: por ejemplo en la representación de estructuras dinámicas como redes sociales o la realización de investigaciones forenses".

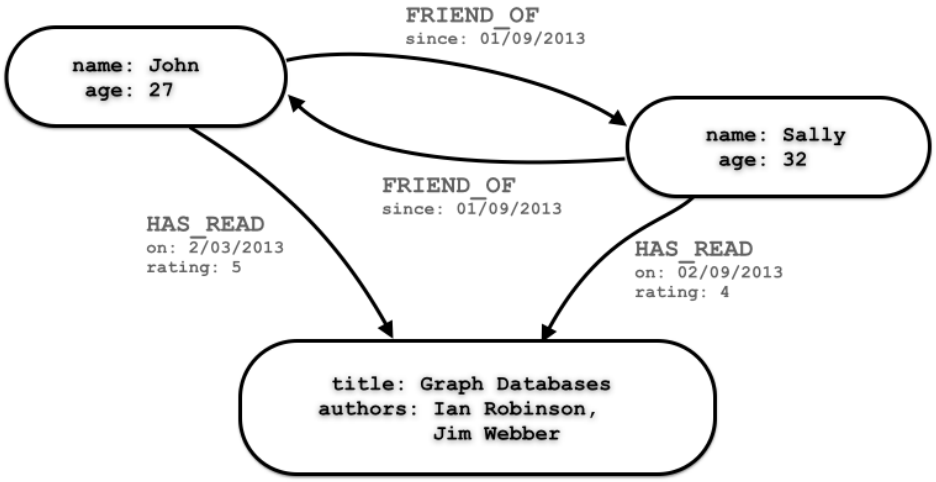



Figura 5. Bases de datos orientadas grafos
Fuente: neo4j.com/developer/guide-data-modeling/
Elaboración: Neo4j (2016)

Ejemplos de las principales bases de datos de grafos son: Neo4j, InfoGrid, Sones GraphDB, AllegroGraph y InfiniteGraph.

### 1.2.3. Características de las bases de datos NoSQL.

Aunque es difícil determinar propiedades comunes para un conjunto de tecnologías, se proponen las siguientes características como las más importantes de las bases de datos NoSQL en el desarrollo del presente trabajo.

#### *1.2.3.1. Escalabilidad horizontal y vertical.*

La escalabilidad indica la habilidad de un sistema para hacerse más grande y/o cambiar su configuración para adaptarse a nuevas situaciones como el crecimiento de información, sin afectar su funcionamiento y sin perder calidad.

**Escalabilidad horizontal**

Se refiere a la facilidad de añadir o eliminar nuevos componentes de hardware al sistema sin afectar el rendimiento (Romero *et al*., 2012). Es decir, el escalamiento horizontal se logra utilizando muchos servidores que forman una colección de servidores que trabaja en conjunto como uno solo. En un entorno distribuido es posible realizar una buena distribución de sobre carga conectando varios servidores para conseguir una mayor capacidad de procesamiento de datos, en vez de utilizar grandes "*mainframes"* que son demasiados costosos.

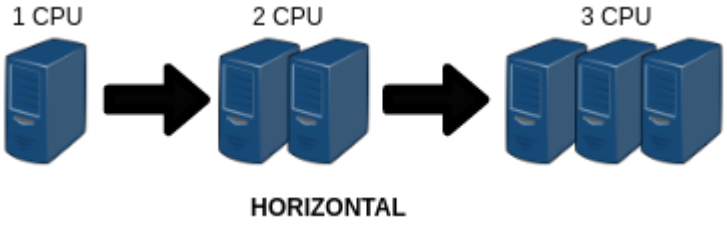



Figura 6. Escalabilidad horizontal
Fuente: Elaboración propia

En la Figura 6, se muestra una representación de la escalabilidad horizontal, mientras se empleen más servidores el trabajo se distribuye en más de un servidor obteniendo una mayor capacidad de procesamiento con un coste proporcionalmente bajo. Es una solución muy buena para aplicaciones que proporcionan servicios web, correo o supercomputación[5].

Una de las ventajas del escalado en horizontal es el equilibrio de carga que se logra cuando al conjunto de servidores o clúster se adiciona nuevos servidores duplicados. La Figura 7 muestra un ejemplo de la escalabilidad horizontal de un sistema con alta proporción de lectura y escritura que escala por replicación de datos. Cada servidor utiliza la base de datos "maestro" para escritura y una base de datos de réplica para lecturas. La función del balanceador de carga es enviar solicitudes sin estado a las bases de datos. En caso que el sistema se sobrecargara de solicitudes, únicamente es necesario agregar otro servidor.

Una ventaja importante del escalado en horizontal es que no se ve limitado por hardware; y cada servidor que se agrega proporciona una mejora al sistema.

[5] Supercomputación es un término utilizado para referirse al uso de supercomputadores de alta gama en cuanto al rendimiento, capacidad de almacenamiento y eficiencia para conseguir potencia de cálculo (García, 2004).

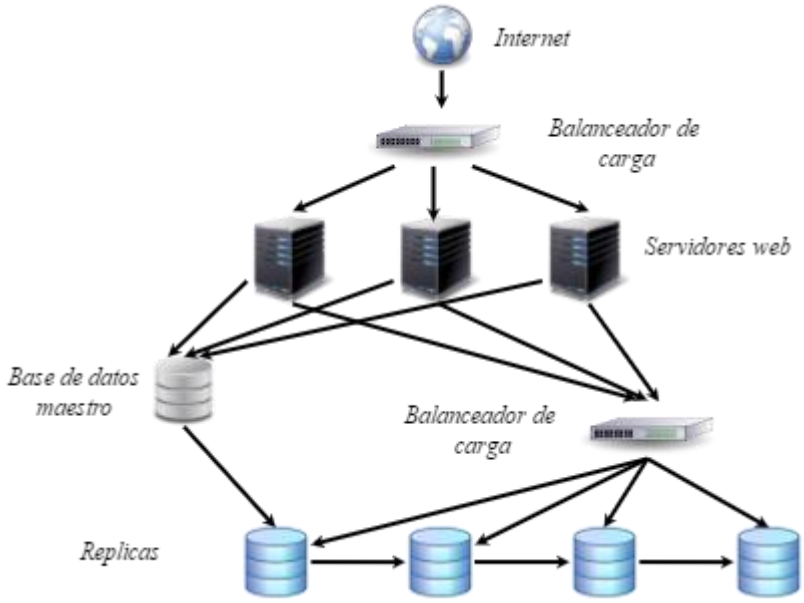



Figura 7. Ejemplo de sistema de lectura-escritura
Fuente: di-side.com/blog/software-scalability
Elaboración: Di-Side (2014)

**Escalabilidad vertical**

El escalado en vertical involucra el uso de máquinas grandes, poderosas y más rápidas. El escalar verticalmente significa pasar de un equipo a otro más potente (con más memoria, más procesadores), es decir, un único supercomputador con varios núcleos de CPU capaz de almacenar y procesar grandes cantidades de información. Este tipo de escalamiento es el más caro.

En el escalamiento vertical se requiere la migración total de un sistema a uno nuevo con un hardware más potente y eficiente. La desventaja de este tipo de escalabilidad es que a medida que se emplee hardware de altas prestaciones, los costos económicos serán mucho más altos. Pero a nivel estructural no supone ninguna modificación de relevancia, lo que lo hace una buena alternativa si se puede asumir los costos.

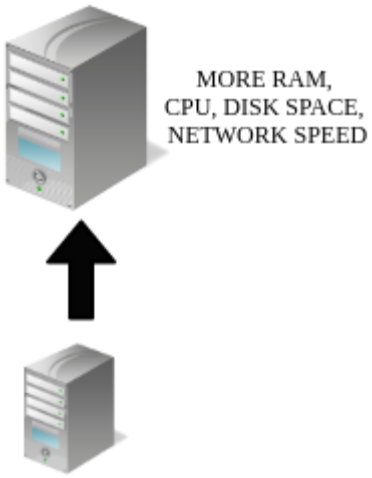



Figura 8. Escalabilidad vertical
Fuente: Elaboración propia

#### *1.2.3.2. Mecanismos de tolerancia a fallos.*

Es la capacidad de los sistemas de seguir funcionando a pesar de la pérdida arbitraria de información. El objetivo de los mecanismos de tolerancia a fallos es que el sistema pueda ser capaz de recuperarse de fallas parciales sin que se afecte el rendimiento global.

**Fragmentación o Sharding**

Es una técnica que permite dividir un conjunto de datos muy grandes en fragmentos de datos más pequeños. Los fragmentos se asignan normalmente en múltiples servidores, a fin de distribuir la carga y así asegurar un óptimo rendimiento del sistema.

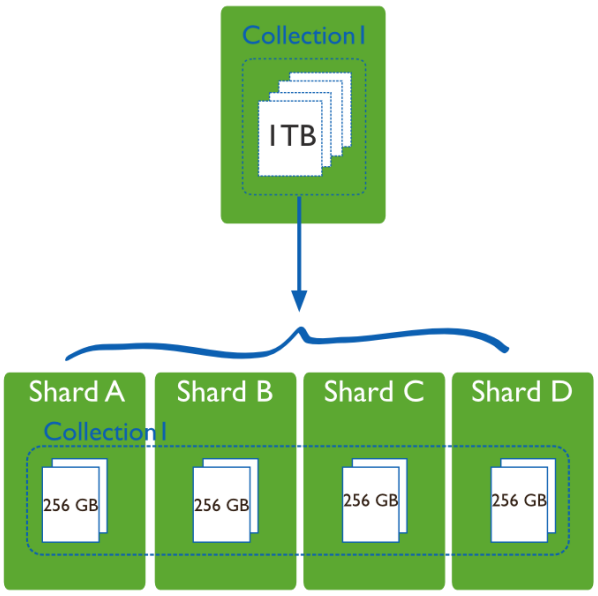



Figura 9. Sharding
Fuente: docs.MongoDB.com/v3.0/core/sharding-introduction/
Elaboración: MongoDB (2016)

La  Figura 9 muestra un ejemplo de base de datos que almacena un conjunto de datos de 1 Terabyte, dividido en 4 fragmentos a través de varios servidores, en donde cada fragmento es una base de datos independiente que puede contener al menos 256 GB de datos y colectivamente los fragmentos forman una sola base de datos lógica. Si existieran 40 fragmentos cada fragmento podría sostener solamente 25 GB de datos.

Entre los beneficios que brinda la fragmentación de datos están:

- Uso más eficiente de recursos.
- La carga de datos se equilibra en múltiples servidores.
- Recuperación rápida y transparente, cuando un servidor falle.
- Capacidad para trabajar simultáneamente en varias bases de datos permite realizar diferentes tareas en paralelo, lo que conlleva mayor rendimiento.

**Replicación**

Según Barragan y Forero (2013) la replicación “es un mecanismo utilizado para propagar y separar datos en un ambiente distribuido con el objetivo de tener mejor performance y confiabilidad, mediante la reducción de dependencia”. En otras palabras, es un mecanismo que proporcionan las bases de datos para asegurar una alta disponibilidad mediante copias de datos almacenadas en varios servidores locales o remotos. De manera que si un servidor falla sea posible acceder a los datos a partir de las copias de seguridad almacenadas en otros servidores. Las bases de datos NoSQL soporta diferentes mecanismos de replicación que pueden ser: Maestro-Esclavo o Maestro-Maestro.

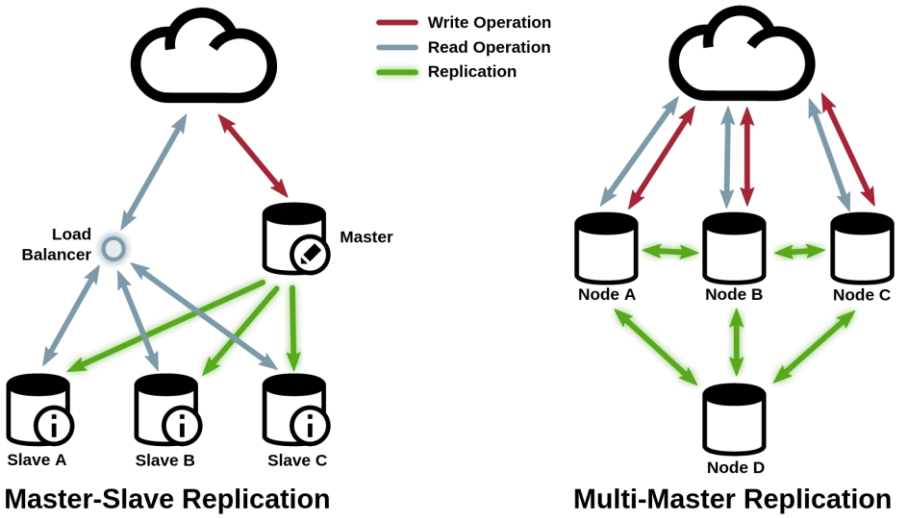



Figura 10. Tipos de replicación
Fuente: http://berb.github.io/diploma-thesis/original/062_internals.html
Elaboración: Benjamín Erb

- **Maestro-Esclavo.** El nodo maestro se utiliza para escrituras, mientras que los N esclavos para lecturas. Es decir, que todas las actualizaciones que se realicen sobre el maestro son replicadas en el esclavo.

- **Maestro-Maestro.** Todos los nodos que tengan la base de datos replicada actúan como maestros y como esclavos al mismo tiempo.

En la Figura 10 se observa dos configuraciones de replicación para un servidor de aplicación web. En el lado izquierdo se muestra una configuración *maestro-esclavo,* donde el maestro maneja escribe y actualiza asincrónicamente en los esclavos. Las peticiones de lectura son equilibradas con la carga a los esclavos. En el lado derecho se muestra una configuración de replicación común para CouchDB *maestro-maestro* donde todos los nodos manejan las solicitudes y realizan la replicación asincrónica, pudiendo generar un esquema de replicación lo suficiente complejo y versátil.

### 1.3. Beneficios

A continuación se menciona algunos beneficios que proporcionan las bases de datos NoSQL:

- Facilidad para escalar si existe necesidad de añadir nuevos servidores para manejar la sobrecarga de datos.

- Mejora la disponibilidad y gestión de la información, de manera que si un servidor ha fallado los usuarios puedan tener acceso a los datos a través de réplicas almacenadas en otros servidores.

- Mejora los tiempos de respuestas de las peticiones de los usuarios que involucran el procesamiento de datos.

- Mejora el rendimiento de los sistemas NoSQL para procesar cantidades masivas de información.

- Utiliza diferentes tipos de datos de forma flexible, y evita los procesos de transformación como ocurre en el modelo relacional.

- Mayor capacidad de almacenamiento, es posible almacenar peta bytes de información distribuida en miles de servidores.

## 1.4. Limitaciones

Se excluye del presente trabajo los siguientes puntos:

- Análisis exhaustivo de cada una las características de los diferentes tipos de almacenamiento NoSQL.
- Demostración del Teorema de CAP.

## 1.5. Tabla de abreviaciones

| Abreviación | Significado |
|---|---|
| SQL | Lenguaje de consulta estructurado |
| RDBMS | Sistema de gestión de bases de datos relacionales |
| NoSQL | No Only SQL |
| CAP | Consistencia, disponibilidad y tolerancia a partición |
| SQL | Lenguaje de consulta estructurado |

# CAPÍTULO II

# ANÁLISIS DE SISTEMAS NOSQL EN ENTORNOS DISTRIBUIDOS

En el presente capítulo se indican y analizan varios estudios realizados sobre diferentes bases de datos NoSQL con la finalidad de obtener una comparativa general con las características más importantes que ayuden en el desarrollo del presente Trabajo de Titulación, en el transcurso del capítulo se explican más detalladamente.

Teniendo en cuenta las características indicadas en la comparativa general, se seleccionan las posibles bases de datos candidatas que cumplan con determinados criterios. Y por último, se analiza cuáles de estas bases de datos son más adecuadas para solucionar el problema, siendo necesario seleccionar los sistemas NoSQL que permitan la implementación sobre un entorno distribuido, que soporten escalabilidad, tolerancia a fallos y que además sean multi-plataforma.

## 2.1. Estudio sobre comparativas de las principales bases de datos NoSQL

En esta sección se indican algunos estudios comparativos realizados por diferentes autores, en cada uno se resaltan algunas características importantes de las bases de datos NoSQL. Como primera comparativa se presenta un fragmento del estudio realizado por Faraj, Rashid, y Shareef (2014) en el que se compara las bases de datos relaciones y no relacionales desde distintos parámetros como el modelo de datos, la representación de datos, el esquema y lenguaje de consulta que utilizan y el escalamiento que soportan.

Tabla 1. Comparación entre las bases de datos relacionales y no relacionales

| Parámetro | Base de datos relacional | Base de datos NoSQL |
|---|---|---|
| Modelo de base de datos | El modelo de base de datos se basa en el enfoque del modelo relacional conocido como bases de datos relacionales (RDBMS) | El modelo de base de datos se basa en el enfoque del modelo conocido como no-relacional o base de datos NoSQL |
| Representación de datos | Almacena datos estructurados en forma de tablas (filas, columnas), relaciones entre tablas o combinaciones | Almacena datos no estructurados basados en varios tipos de almacenamiento como clave y valor, familia de columna, base de datos de grafos o almacén de documentos |
| Esquema | Define tablas y estructuras | El esquema es altamente flexible y dinámico |
| Escalabilidad | Verticalmente escalable (aumentan la potencia de hardware como CPU, RAM, disco duro o, etc.) | Horizontalmente escalable (aumenta la capacidad agregando más máquinas o servidores de la base de datos) |

| | | |
|---|---|---|
| Lenguaje de consulta | Utiliza el lenguaje SQL para definir y manipular datos | NoSQL no tiene un lenguaje de consulta estándar |
| Característica | Una característica clave de RDBMS es la facilidad de mantenimiento que hace el sistema para reparar, prueba y copia de seguridad fácilmente, ofreciendo herramientas para el administrador de base de datos | Una característica clave de los sistemas NoSQL es "Shared Nothing", escala horizontalmente – replicación y partición de datos en varios servidores |

Fuente: ow.ly/Flgf3076iEh
Elaboración: Azhi Faraj, Bilal Rashid, Twana Shareef

El segundo estudio que se presenta fue realizado por Moniruzzaman y Akhter (2013), el cual efectúa una clasificación y comparación de las bases de datos NoSQL para el análisis de Big data sobre la base de los siguientes atributos:

**Diseño:** este atributo presenta las características sobre las que se ha diseñada la base de datos, entre las que se mencionan el tipo de almacenamiento, el lenguaje de consulta que soportan, el protocolo que usan y si soportan MapReduce para procesar datos en paralelo.

**Integridad:** este atributo describe las características que una base de datos NoSQL debe poseer para afirmar que la información que almacenan tiene calidad. Entre ellas están la atomicidad, consistencia, aislamiento y durabilidad que son características que garantizan que las transacciones en las bases de datos sean fiables.

**Indexación:** este atributo muestra las características de indexación que soporta cada base de datos como índices secundarios, claves compuestas, búsqueda de texto completo e índices geoespaciales. En general, una base de datos que soporta indexación tienden a mejorar la velocidad de procesamiento cuando se realizan búsquedas sobre miles de datos.

**Distribución:** este atributo describe el esquema de distribución que soporta la base de datos que puede ser replicación o fragmentación y compara entre las distintas bases de datos cuales soportan escalabilidad horizontal.

**Sistema:** este atributo presenta los lenguajes de programación que soporta cada base de datos y sobre que sistemas operativos puede implementarse.

| Attributes | | NoSQL Databases | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Database model | | Document-Stored | | Wide-Column Stored | | | | Key-Value Stored | | Graph-oriented |
| | Features | MongoDB | CouchDB | DynamoBD | HBase | Cassandra | Accumulo | Redis | Riak | Neo4j |
| Design & Features | Data storage | Volatile memory File System | Volatile memory File System | SSD | HDFS | | Hadoop | Volatile memory File System | Bitcask LevelDB Volatile memory | File System Volatile memory |
| | Query language | Volatile memory File System | JavaScript Memcached-protocol | API calls | API calls REST XML Thrift | API calls CQL Thrift | | API calls | HTTP JavaScript REST Erlang | API calls REST SparQL Cypher Tinkerpop Gremlin |
| | Protocol | Custom, binary (BSON) | HTTP, REST | - | HTTP/REST Thrift | Thrift & custom binary CQL3 | Thrift | Telnet-like | HTTP, REST | HTTP/RES Tembedding in Java |
| | Conditional entry updates | Yes | Yes | Yes | Yes | No | Yes | No | No | |
| | MapReduce | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | No |
| | Unicode | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| | TTL for Entries | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | |
| | Compression | Yes | Yes | - | Yes | Yes | Yes | Yes | Yes | |
| Integrity | Integrity model | BASE | MVCC | ASID | Log Replication | BASE | MVCC | - | BASE | ASID |
| | Atomicity | Conditional | Yes | Yes | Yes | Yes | Conditional | Yes | No | Yes |
| | Consistency | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes |
| | Isolation | No | Yes | Yes | No | No | - | Yes | Yes | Yes |
| | Durability (data storage) | Yes | Yes | Yes | Yes | Yes | Yes | Yes | - | Yes |
| | Transactions | No | No | No | Yes | No | Yes | Yes | No | Yes |
| | Referential integrity | No | No | No | No | No | No | Yes | No | Yes |
| | Revision control | No | Yes | Yes | Yes | No | Yes | No | Yes | No |
| Indexing | Secondary Indexes | Yes | Yes | No | Yes | Yes | Yes | - | Yes | - |
| | Composite keys | Yes | Yes | Yes | Yes | Yes | Yes | - | Yes | - |
| | Full text search | No | No | No | No | No | Yes | No | Yes | Yes |
| | Geospatial Indexes | Yes | No | No | No | No | Yes | - | - | Yes |
| | Graph support | No | No | No | No | No | Yes | No | Yes | Yes |
| Distribution | Horizontal scalable | Yes | Yes | Yes | Yes | Yes | Yes | | Yes | No |
| | Replication | Yes | Yes | Yes | Yes | Yes | Yes | | Yes | Yes |
| | Replication mode | Master-Slave-Replica Replication | Master-Slave Replication | - | Master-Slave Replication | Master-Slave Replication | - | Master-Slave Replication | Multi-master replication | - |
| | Sharding | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes | Yes |
| | Shared nothing architecture | Yes | Yes | Yes | Yes | Yes | - | - | Yes | - |
| System | Value size max. | 16MB | 20MB | 64KB | 2TB | 2GB | 1EB | - | 64MB | |
| | Operating system | Cross-platform | Ubuntu Red Hat Windows Mac OS X | Cross-platform | Cross-platform | Cross-platform | NIX 32 entries Operating system | Linux *NIX Mac OS X Windows | Cross-platform | Cross-platform |
| | Programming language | C++ | Erlang C++ C Python | Java | Java | Java | Java | C C++ | Erlang | Java |

Figura 11. Comparación de las bases de datos NoSQL con una matriz sobre la base de los atributos diseño e integridad, indexación, distribución y sistema
Fuente: https://arxiv.org/abs/1307.0191
Elaboración: A B M Moniruzzaman, Syed Akhter Hossain

A medida que un entorno distribuido trata con múltiples servidores, se deberá priorizar el uso de las dimensiones CAP pues se debe renunciar al menos una de las tres características. En la Tabla 2 se presenta comparativa de las bases de datos NoSQL en relación a los criterios de consistencia, características de CAP y licencia (Fernández et al., 2014).

Tabla 2. Comparación de CAP para bases de datos NoSQL

| Base de datos NoSQL | Modelo de datos | Consistencia | Clasificación CAP | Licencia |
|---|---|---|---|---|
| BigTable | En columnas | Eventualmente consistente | CP | Propiedad de Google |
| DynamoDB | Clave-valor | Eventualmente consistente | AP | Propiedad de Amazon |
| HBase | Columnas | Eventualmente consistente | CP | Apache – Open Source |
| Cassandra | Columnas | Eventualmente consistente | AP/CP | Apache – Open Source |
| MongoDB | Documentos | Eventualmente con optimismo | CP | GNU – Open Source |
| CouchDB | Documentos | Eventualmente con optimismo | CP | Apache – Open Source |

Fuente: ow.ly/Z0jD3076GzZ
Elaboración: Fernández *et al*

La comparativa anterior prácticamente revela que las características de diversas bases de datos tienen la misma finalidad tales como alta disponibilidad, consistencia, y tolerancia de partición. Con base a las comparativas antes mencionadas se puede decir que las bases de datos NoSQL son populares porque son fácilmente escalables sin interrumpir las operaciones en línea.

La escalabilidad y tolerancia a fallos son puntos críticos cuando se manejan millones de datos, en este caso, los sistemas NoSQL se convierten en una alternativa factible debido a que permiten el almacenamiento de grandes volúmenes de información y soportan distintos mecanismos de escalabilidad y tolerancia a fallos. DB-engines (2016) es una iniciativa encargada de reunir y presentar información sobre los diferentes sistemas de gestión de base de datos relacionales y no relacionales, su lista se basa en estudios mensuales de popularidad que se miden con base en 5 factores: número de referencias en los buscadores Google y Bing, número de búsquedas en Google Trends, Tweets y número de apariciones en perfiles de LinkedIn. En la Tabla 3 se presenta el estudio correspondiente al mes de Diciembre del 2016.

Tabla 3. Bases de datos NoSQL más populares (Diciembre 2016)

| Criterio | Base de datos NoSQL | | | | |
|---|---|---|---|---|---|
| | MongoDB | Cassandra | Redis | HBase | Neo4j |
| Ranking | #1 | #2 | #3 | #4 | #5 |
| Método de replicación | Maestro-esclavo | Selectable replication factor | Maestro-esclavo | Selectable replication factor | Maestro-esclavo |
| Método de partición | Sharding | Sharding | Sharding | Sharding | No |
| Modelo de datos | Document store | Wide column store | Key-value store | Wide column store | Graph |
| Licencia | Libre y comercial | Open Source | Libre y comercial | Open Source | Libre y comercial |
| Sistema operativo | Linux<br>OS X<br>Solaris<br>Windows | BSD<br>Linux<br>OS X<br>Windows | BSD<br>Linux<br>OS X<br>Windows | Linux<br>Unix<br>Windows | Linux OS X<br>Windows |
| Desarrollador | MongoDB, Inc | Apache Software Foundation | Salvatore Sanfilippo | Apache Software Foundation | Neo Technology |
| Sitio web | www.MongoDB.com | cassandra.apache.org | redis.io | hbase.apache.org | neo4j.com |

Fuente: db-engines.com/en/ranking
Elaboración: DB-engines

La última comparativa muestra un resumen de las características generales de las bases de datos NoSQL más populares orientadas a documentos. DB-engines considera como bases de datos representantes a MongoDB, Couchbase y CouchDB, tales productos son de código abierto, libres de esquemas, y almacenan diferentes tipos de información en las mismas estructuras. En la Tabla 4 se presenta una comparación de las bases de datos orientadas a documentos sobre la base de los siguientes atributos: Lenguaje de programación que soporta, esquema de datos, método de acceso, tolerancia a fallos, tipo de licencia y última versión estable.

Tabla 4. Comparación de bases de datos documentales

| Base de datos | Lenguaje | Esquema de datos | Método de acceso | Tolerancia a fallas | Licencia | Última versión |
|---|---|---|---|---|---|---|
| #1 MongoDB | C, C#, Java, PHP Python, Perl, Ruby Haskell | Libre de esquemas | Proprietary protocol using JSON | Si | AGPL versión 3 | 3.4.1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| #3 Couchbase | C, Java PHP, Python, Perl, Ruby | Libre de esquemas | Memcached protocol RESTful HTTP API | Si | Apache versión 2 | 4.5.1 |
| #4 CouchDB | C, C#, Java, PHP Python, Perl, Ruby | Libre de esquemas | RESTful HTTP/JSON API | Si | Apache versión 2 | 2.0.0 |

Fuente: db-engines.com/en/system/CouchDB;Couchbase;MongoDB
Elaboración: DB-engines

## 2.2. Propuesta de solución

A partir de los estudios comparativos analizados anteriormente se expone una propuesta para la implementación de bases de datos NoSQL hacia un enfoque basado en escalabilidad y tolerancia a fallos dentro de un entorno distribuido. Para lo cual, antes de realizar el procedimiento de la implementación, se seleccionan las posibles bases de datos candidatas considerando determinados criterios y de estas finalmente se seleccionan las bases de datos que se consideren las más adecuadas. En las secciones 2.2.1 y 2.2.2 se describen aquellos criterios seleccionados y los que no son relevantes para la elección de las bases de datos que permitan alcanzar el objetivo del Trabajo de Titulación.

### 2.2.1. Criterios considerados.

A continuación se indican las características que son la base para la elección de las bases de datos, se describe su significado y el motivo por que fue tomado en cuenta para que sea elegido como objeto de selección para las bases de datos. Los criterios considerados se resumen en la comparativa general (ANEXO A) realizada con base en los estudios anteriores, que además es el fundamento para la elección de los posibles sistemas NoSQL candidatos.

- **Licencia**

Hace referencia a las condiciones que establece un producto NoSQL para poder usarlo. La licencia es un parámetro que influye para la elección de la base de datos debido a que de preferencia para el desarrollo del presente trabajo se requiere sistemas con un licenciamiento libre, es decir que no tenga algún costo en la descarga o uso.

- **Ranking**

Indica la posición en que actualmente se ubica el producto según una categoría dada. Este criterio es considerado debido a la preferencia de los usuarios en la utilización de dicho

producto, las categorías consideradas para el ranking son los modelos: Document Store, Key Value Store y Wide Column Store.

- **Sitio web y soporte**

Lista el sitio oficial de la base de datos donde se puede acceder a su documentación y descarga de su última versión estable, lo que ayuda como guía para el desarrollo del Trabajo de Titulación.

- **Modelo de la base de datos**

Señala el tipo de modelo de almacenamiento que soporta la base de datos. Este es un aspecto importante dado que el modelo elegido debe ser flexible a la hora de almacenar los datos de los que se dispone.

- **Sistema operativo**

Describe el sistema operativo en los que puede implementarse y poner en funcionamiento la base de datos.

- **Métodos de acceso y APIs (Application Programming Interface)**

Se refiere al conjunto de métodos y funciones que ofrece la base de datos y que sirven para el desarrollo de aplicaciones.

- **Lenguajes de programación que soportan**

Este criterio lista los diferentes lenguajes de programación que soporta cada base de datos para el desarrollo de aplicaciones.

- **Escalabilidad horizontal**

Se refiere a la capacidad del sistema para aumentar el número de servidores, de tal manera que cuando exista un crecimiento de carga únicamente se agreguen más servidores sin tener que ampliar la capacidad de estos.

- **Método de replicación**

Es uno de los criterios más importantes dentro de esta comparativa en vista que es esencial para cumplir con el objetivo principal del Trabajo de Titulación. La replicación implica guardar copias de datos en varios servidores a través de los diferentes mecanismos de replicación

que soportan los sistemas de almacenamiento, los cuales pueden ser: maestro-maestro y maestro-esclavo.

- **Método de partición**

Junto con la replicación este criterio es de gran importancia debido a que es una de las características fundamentales que deben cumplir las bases de datos que se implementarán en este trabajo. La fragmentación o partición de datos se trata de un mecanismo que las bases de datos NoSQL ofrecen para distribuir los datos en distintos servidores y así asegurar la escalabilidad y rendimiento en los sistemas de almacenamiento.

Los criterios antes mencionados se muestran en la comparativa general del ANEXO A y es el punto de partida para el desarrollo del Trabajo de Titulación.

### 2.2.2. Criterios no considerados.

Las siguientes características fueron descartadas debido a que se consideran criterios básicos de todo sistema NoSQL y por lo tanto no son relevantes para el desarrollo del trabajo.

- Desarrollador
- Esquema de datos
- Protocolo de acceso
- Lenguaje de consulta

### 2.2.3. Bases de datos candidatas.

Partiendo del análisis de la comparativa general, se seleccionan las posibles bases de datos que sean más adecuadas para la implementación en un ambiente distribuido, por consiguiente se toma en cuenta aquellos sistemas que cumplan con todos los criterios propuestos en la sección 2.2.1. Las bases de datos candidatas son las siguientes:

- MongoDB
- CouchDB
- Riak KV
- Redis
- Cassandra
- Hbase

### 2.2.4. Posibles soluciones de implementación.

Antes de definir las posibles soluciones de implementación, se realiza el análisis de cada una de las bases de datos seleccionadas como candidatas con el fin de definir los criterios más relevantes que permitan realizar una elección final.

#### *2.2.4.1. Análisis de criterios para elección de base de datos.*

La elección final de los sistemas de almacenamiento NoSQL a implementar se basa en los criterios que se describen a continuación, los cuales son considerados como los más importantes para cumplir con el objetivo general del proyecto y son aquellos que se acoplan a las condiciones e infraestructura con la que se dispone para la implementación del entorno distribuido. Los datos que se utilizan para su almacenamiento e implementación se encuentran ya en formato NoSQL por lo que no es necesario realizar ninguna conversión

Teniendo en cuenta estas consideraciones, los criterios más importantes para la toma de decisión final son:

- La licencia debe ser libre: se dispone de una gran cantidad de datos que demanda gran capacidad de almacenamiento, si bien algunos sistemas relacionales son capaces de almacenarlos bajo una arquitectura distribuida esto demanda precios muy altos para la adquisición de licencias y hardware. En consecuencia, la utilización de bases de datos NoSQL que brindan licencias gratuitas suprime tales costos.

- Sistema operativo: considerando la infraestructura de hardware que dispone el Laboratorio de Tecnologías Avanzadas de la Web (S.O: Ubuntu y Mac OS), se requiere que la o las bases de datos seleccionadas soporten tales sistemas operativos.

- Modelo de base de datos orientado a documentos: las bases de datos elegidas deben ser capaces de soportar en modelo de documentos por las siguientes razones: no requiere definir una estructura para el almacenamiento de datos que está formado por una clave y un valor, permite cargar archivos JSON y csv, y existe la documentación necesaria que describe su proceso de implementación.

- Escalabilidad: es una de las características más importantes que debe tener la base de datos pues indica la capacidad del sistema para adaptarse al crecimiento de carga sin afectar el rendimiento. Normalmente la escalabilidad se realiza verticalmente, es decir, aumentando el tamaño y la potencia del sistema, sin embargo para este trabajo

se requiere de una base de datos capaz de escalar horizontalmente, es decir, que permita incrementar el número de servidores del sistema.

- Capacidad de replicación o fragmentación: la base de datos debe ser capaz de soportar ya sea la capacidad de replicación o de fragmentación que son necesarios para asegurar una alta disponibilidad de los datos, junto al criterio de escalabilidad son los que más influyen para la toma de decisión final.

### 2.2.5. Bases de datos seleccionadas.

Tomando como referencia el listado de las bases de datos candidatas presentado en el apartado 2.2.3 y considerando los criterios finales que se mencionan en la sección 2.2.4.1, se determinó que las bases de datos NoSQL que cumplen con tales criterios son las siguientes: MongoDB, CouchDB y Couchbase.

#### *2.2.5.1. MongoDB.*

Es una base de datos orientada a documentos y es la líder de los sistemas NoSQL. MongoDB usa un formato propio llamado BSON para guardar datos en documentos con lo cual no es necesario definir una estructura, esto significa que cada registro puede tener un distinto esquema de datos. BSON es una representación binaria de estructuras de datos y mapas, diseñado para tener un almacenamiento y velocidad más eficiente que JSON.

MongoDB fue creada para brindar escalabilidad, alto rendimiento y disponibilidad, escalando desde una simple implantación de un único servidor a arquitecturas más robustas y complejas. Los registros o conjunto de datos en MongoDB se denominan documentos. Un documento es un registro o conjunto de datos formado por una estructura simple y compuesta por pares: clave y valor, donde la clave es el nombre del campo y el valor contiene los datos; ambos se separan mediante el uso de ":" como se muestra en la Figura 12.


Figura 12. Ejemplo de un documento en MongoDB
Fuente: docs.MongoDB.com/manual/core/document/
Elaboración: MongoDB

En cada documento existe el campo _id que permite identificar a un documento de manera única. Los documentos también pueden ser agrupados en colecciones y una colección puede

anidar un número indeterminado de documentos con muy diferentes formatos. En la siguiente figura se muestra un ejemplo de una colección de documentos en MongoDB.

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Figura 13. Colección de documentos en MongoDB
Fuente: docs.MongoDB.com/v3.2/core/data-modeling-introduction/
Elaboración: MongoDB

Para situaciones donde la lectura y escritura es más frecuente MongoDB brinda un elevado rendimiento, potenciando la computación en memoria (MongoDB, 2016). Además ofrece escalado en horizontal para adaptarse al aumento de información con los que trabajan, asimismo proporciona un método de replicación nativa y apoya la tolerancia a fallos a través de un método para dividir los datos entre los múltiples servidores, conocido como Sharding.

#### *2.2.5.2. CouchDB.*

Es una base de datos orientada a documentos que almacena sus datos en formato JSON formado por pares de clave-valor y al igual que MongoDB no requiere definir una estructura para el almacenamiento de datos y las consultas, combinaciones y transformaciones de los datos se realizan con JavaScript por medio de MapReduce y HTTP como API.

En CouchDB cada base de datos es una colección de documentos JSON independientes que se organizan por medio de vistas. En la Figura 14 se muestra un ejemplo típico de un documento en CouchDB, en cada documento siempre existirán los campos: _id y _rev, el primero ayuda a identificar como único un documento y el segundo sirve para controlar la versión de cada documento.

```
{
  "_id":"biking",
  "_rev":"AE19EBC7654",

  "title":"Biking",
  "body":"My biggest hobby is mountainbiking. The other day...",
  "date":"2009/01/30 18:04:11"
}
```

Figura 14. Documento en CouchDB
Fuente: guide.CouchDB.org/editions/1/es/
Elaboración: CouchDB

Las colecciones de documentos de CouchDB contienen datos en formato JSON así es posible lograr casi cualquier tipo de estructura para poder consultar y obtener datos. Las vistas es el mecanismo que provee Couchbase para la combinación de documentos que devuelven como resultado valores de varios documentos. Según Anderson, Lehnardt, y Staler (2010) las vistas son útiles para:

- Filtrar los documentos en su base de datos para encontrar aquellos más relevantes para un proceso en particular.
- Extraer datos de los documentos y presentar en un orden específico.
- La construcción eficiente de índices para encontrar documentos por cualquier valor o estructura que reside en ellos.
- Uso de índices para representar las relaciones entre los documentos.
- Por último, con vistas puede hacer todo tipo de cálculos con los datos de sus documentos.

#### *2.2.5.3. Couchbase.*

Al igual que MongoDB y CouchDB, Couchbase es una base de datos no relacional orientada a documentos capaz de almacenar Terabytes de datos, de alto rendimiento y fácilmente escalable. Utiliza JSON para almacenar datos como formato de documentos, lo cual permite una gran flexibilidad en los datos al no forzar un esquema fijo.

Couchbase almacenan la mayor cantidad de sus datos en RAM para ello usa una capa de caché que permite el acceso rápido a sus datos. Brown (2012) afirma que el “sistema utiliza múltiples servidores y capas de caché con particionamiento automático,… Esto permite que sea posible agrandar y reducir el clúster para aprovechar más RAM o disco de E/S para ayudar a mejorar el rendimiento”. Couchbase almacena sus datos en el disco pero inicialmente la escritura y actualización se realiza mediante la capa caché que es la que proporciona un alto rendimiento.

La arquitectura básica de la base de datos consiste en uno o más servidores Couchbase organizados como un clúster ( Figura 15). Cada servidor contiene un conjunto de servicios configurables que incluyen una caché administrada, almacenamiento, administración de clúster así como servicios de datos, índice y de consulta. Uno de los aspectos centrales de Couchbase es su escalamiento horizontal que permite aumentar la capacidad de gestión de datos y rendimiento del sistema añadiendo más servidores, a diferencia de los sistemas relacionales que requieren servidores más potentes para aumentar la capacidad del sistema.

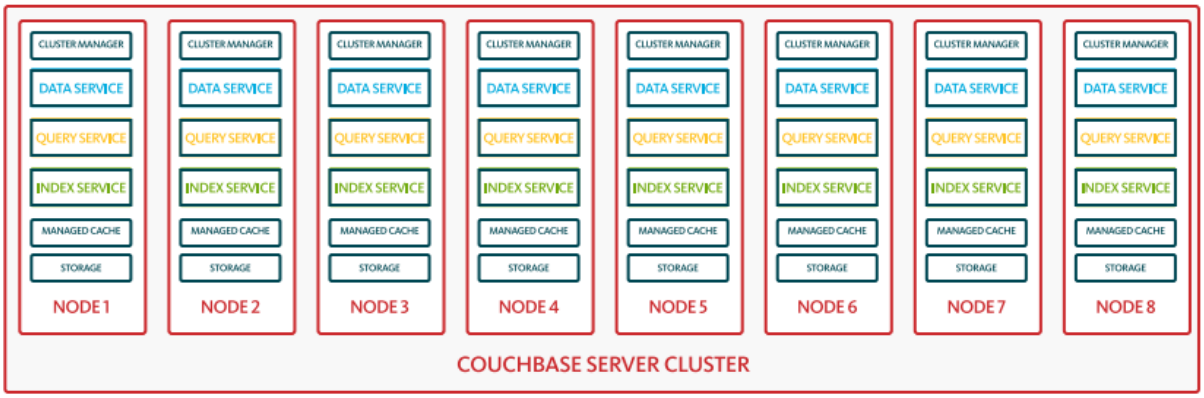



Figura 15. Clúster Couchbase
Fuente: ow.ly/OCA2307DanX
Elaboración: Couchbase

# CAPÍTULO III

# IMPLEMENTACIÓN DE LAS BASES DE DATOS NOSQL

En el presente capítulo se indica el ciclo de desarrollo para la implementación de las bases de datos NoSQL elegidas, dicho ciclo está constituido de una serie de pasos que constituyen el diseño metodológico para llevar a cabo el trabajo, se define el modelo de arquitectura distribuida que se utilizará para realizar la implementación y por último se describe una serie de procedimientos que detallen paso a paso el proceso de instalación y configuración de cada base de datos sobre el ambiente distribuido.

## 3.1. Fases para la implementación de las bases de datos NoSQL

Para el desarrollo del trabajo se utiliza como guía una serie de fases que se adaptan al modelo de desarrollo evolutivo, tomado como referencia del modelo de desarrollo evolutivo de Somerville (2005) que lo describe como un modelo que “se basa en la idea de desarrollar una implementación inicial, exponiéndola a los comentarios del usuario y refinándola a través de las diferentes versiones hasta que se desarrolla un sistema adecuado”. Dicho modelo exige la entrega del sistema sea iterativa e incremental debido a los cambios que pudieran presentarse durante el proceso de implementación. Las actividades realizadas y entregadas al director del Trabajo de Titulación para su revisión representan los incrementos y las iteraciones permiten desarrollar versiones cada vez más completas ya que las fases del modelo pueden repetirse varias veces en caso de no cumplir con los objetivos planteados. En la Figura 16 se presenta el esquema de las fases a seguir para la implementación de las bases de datos NoSQL sobre un entorno distribuido.


Figura 16. Ciclo de desarrollo
Fuente: Elaboración propia

### 3.1.1. Elección del modelo de base de datos NoSQL.

La primera fase para el desarrollo del Trabajo de Titulación consiste en la elección del modelo de la base de datos NoSQL que como se menciona en el Capítulo I, se clasifican en: documentales, clave-valor, grafos y basados en columnas.

Para elegir el modelo de bases de datos ideal se tomó en cuenta la información que gestionará la base de datos NoSQL, en este caso, los datos con los que se dispone son 20 millones de registros en formato JSON que contienen información sobre los Tweets más populares dentro de un determinado intervalo de tiempo. Por tanto en el apartado 2.2.5 del Capítulo I, las bases de datos que se seleccionaron para su implementación exigían como requisito esencial un modelo de base de datos capaz de almacenar y gestionar documentos. Algunos de los beneficios que proporcionan las bases de datos basadas en documentos, según Mancilla (2013) es la capacidad de almacenamiento y el alto rendimiento que proporcionan para almacenar grandes cantidades de información que varía desde 1 TB hasta 70 TB.

### 3.1.2. Elección de la base de datos NoSQL.

En esta fase se ha considerado la lista de bases de datos candidatas presentada en el apartado 2.2.3 y los criterios de selección finales que se muestran en el análisis realizado en el apartado 2.2.4.1. Como resultado de esta fase, las bases de datos seleccionadas para el desarrollo del Trabajo de Titulación son las siguientes: MongoDB. CouchDB y Couchbase. En el apartado 2.2.5 se describen con más detalle cada una de las bases de datos.

Las bases de datos NoSQL escogidas tienen una similitud con las bases de datos relacionales, siendo así que una colección es semejante a una tabla, un documento es semejante a una fila y un campo a una columna. En la Tabla 5, se presenta una comparación de las bases de datos NoSQL escogidas en términos de una base de datos relacional.

Tabla 5. Comparación de términos en MongoDB, CouchDB y Couchbase

| Base de datos no relacional | MongoDB | CouchDB | Couchbase |
|---|---|---|---|
| Tabla | Colección | - | - |
| Fila | Documento | Documento | Documento |
| Columna | Campo o atributo | Campo o atributo | Campo o atributo |

Fuente: Elaboración propia

### 3.1.3. Definición de la arquitectura.

En esta fase se define la arquitectura sobre la que se implementa la base de datos NoSQL, la arquitectura se refiere al diseño o la topología de red que utiliza el entorno distribuido para comunicarse con los nodos del sistema; cada nodo representa un servidor (*hardware*) donde está instalado y configurado la base de datos NoSQL (*software*).

A continuación, se indican los requerimientos necesarios de cada uno de los elementos de Software, Hardware y Red de comunicación que conforman la topología del sistema distribuido.

#### *3.1.3.1. Recursos de software.*

El software incluye la versión y otras herramientas utilizadas para la implementación de la bases de datos NoSQL sobre el sistema distribuido. La Tabla 6 muestra el software instalado de cada base de datos.

Tabla 6. Software implementado por cada base de datos NoSQL

| Base de datos | MongoDB | CouchDB | Couchbase |
|---|---|---|---|
| Versión | 3.4.4 | 2.0.0 | 4.5.0 |
| Herramienta de administración | Robomongo | Fauxton | Couchbase Web Console |
| Carga de datos | *mongoimport* | *couchimport* | *cbtransfer* |

Fuente: Elaboración propia

#### *3.1.3.2. Recursos de hardware.*

El hardware está representado por los dispositivos físicos que conforman el sistema distribuido. Para el desarrollo de este trabajo los dispositivos de hardware constituyen los nodos del sistema que se comunican para realizar la transferencia de datos. El hardware utilizado en la instalación y configuración de las bases de datos NoSQL son:

**Servidores:** Los servidores son aquellas computadoras capaces de compartir y acceder a los recursos de otros servidores que se encuentren conectados a una misma red (Béjar Heredia, 2015). En este trabajo la implementación de la base de datos incluye un servidor principal encargado de realizar las tareas de replicación o fragmentación de datos a los servidores secundarios. Los equipos utilizados son:

- 2 Equipos con sistema operativo Mac OS X, con 280 GB de disco duro y 64 GB de memoria RAM.
- 1 Equipo con sistema operativo Ubuntu 16.04, con 450 GB de disco duro y 4 GB de memoria RAM.

**Cableado:** El sistema de cableado conecta los servidores secundarios con el servidor principal y otros periféricos. En este caso, el cable UTP es el que se utiliza para la implementación del sistema distribuido.

**Tarjeta de interfaz de red:** También conocidos como adaptadores de red o solo tarjeta de red, es utilizada por cada computador para comunicarse con el resto de la red. Todos los servidores que conforma el sistema distribuido deben tener instalada una tarjeta de interfaz de red.

Debido a la masiva cantidad de datos que almacenará la base de datos NoSQL se utiliza un disco externo My Book Thunderbolt[6] de 4TB para obtener más capacidad de almacenamiento.

#### *3.1.3.3. Red de comunicación.*

La red de comunicación hace referencia a la conexión de los diferentes servidores para comunicarse e intercambiar información. En este trabajo se conectarán las computadoras a través de una red de área local (LAN). Considerando las configuraciones que se deben realizar tanto para las funcionalidades de fragmentación y replicación, la topología general que se utilizará es una red en estrella, debido a que está se conforma por un servidor central y varios servidores secundarios. Para el desarrollo del presente trabajo, los servidores secundarios están conectados directamente al servidor principal y están configurados para recibir datos. La ventaja de utilizar esta tipo de topología es la facilidad con la que se puede adaptar o configurar un nuevo servidor a la red.

## 3.2. MongoDB

### 3.2.1. Arquitectura.

#### *3.2.1.1. Replicación.*

En MongoDB la replicación consiste en mantener copias de la misma información en varios servidores. Al conjunto de servidores que almacenan la misma información, en MongoDB se conoce como conjunto de réplica (en inglés *Replica Set*) y siguen una arquitectura maestro-

[6] Más información en: https://www.wdc.com/products/wd-outlet/my-book-thunderbolt-duo.html

esclavo. Esto significa que está formado por un servidor primario y uno o varios servidores secundarios.

El servidor primario es el único que puede replicar la información a los servidores secundarios. “Esto se hace para que solo el miembro primario del conjunto de réplicas pueda aceptar operaciones de escritura, y de esta manera asegurar estricta consistencia en todas las lecturas que se hagan en él” (Poveda, 2015, p.19). Generalmente, al miembro primario es donde los usuarios acceden para lecturas, aunque es posible realizar lecturas desde los miembros secundarios. Sin embargo, para ello se debe indicarle al sistema que cambie este comportamiento. Si un miembro primario no está disponible, los servidores secundarios inician elecciones para elegir un servidor entre todos los demás para que se convierta en el nuevo primario.

Los servidores secundarios mantienen las copias de datos presentes en el servidor primario, para ello se sincronizan con el servidor primario y a partir de ese momento mantiene copias de datos realizando las mismas operaciones que se realizan sobre el primario. Y en caso de existir un servidor secundario ya sincronizado, otros servidores secundarios pueden valerse de este par sincronizarse por primera vez.

Según Gutierrez (2014) la replicación en MongoDB “es a nivel del servidor y por tanto no es posible replicar únicamente una de las colecciones o base de datos concretas. Es decir, se replica todo lo que se haya en el servidor”. En la Figura 17 se presenta la arquitectura de replicación de una base de datos en MongoDB.

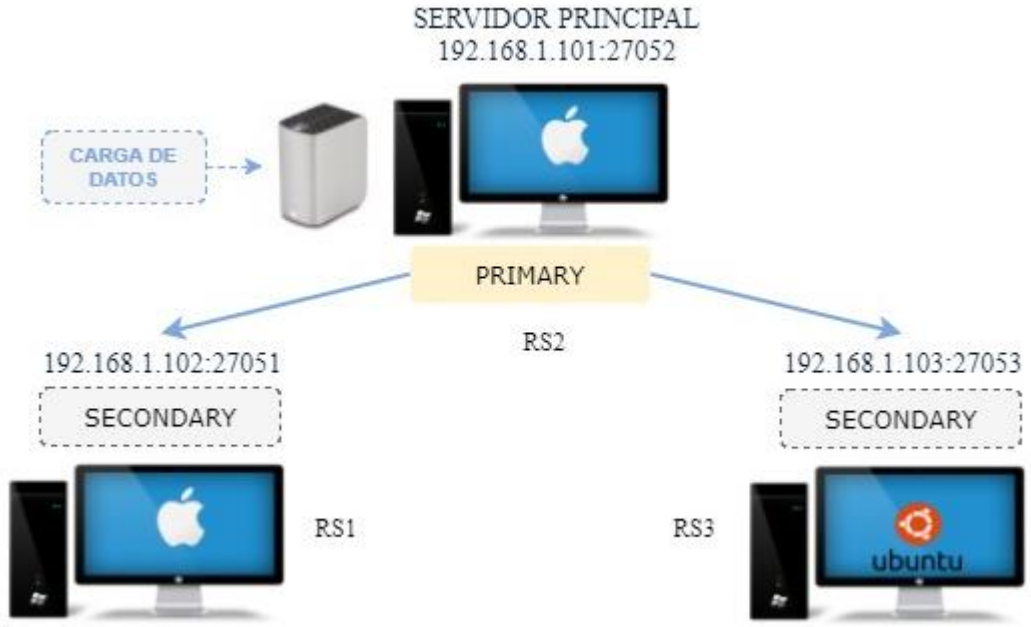



Figura 17. Topología de MongoDB para replicación
Fuente: Elaboración propia

### *3.2.1.2. Sharding.*

Mientras que la replicación en MongoDB busca realizar copias exactas de los datos en varios servidores para garantizar una alta disponibilidad, la fragmentación busca particionar los datos de la base de datos para distribuirlos entre múltiples servidores. Esto debido a que mientras la cantidad de información se hace más grande, cada vez resulta más costoso replicarla pues ocuparía demasiado espacio de almacenamiento.

En MongoDB la división de datos se realiza a nivel de colección, es decir, que se dividen las colecciones en trozos para repartir la carga de trabajo entre diferentes servidores. Para configurar un clúster de servidores en MongoDB[7] son necesarios los siguientes componentes:

- **mongos:** actúan como un router que se encarga de dirigir las peticiones de consulta de los clientes al fragmento correspondiente
- **Servidores de configuración:** almacenan información sobre qué datos contiene cada servidor. El router de consulta utiliza esta información para dirigir las operaciones a determinados fragmentos
- **Shards:** cada fragmento es un conjunto de réplicas que contiene datos fragmentados (Replica Set, la arquitectura de replicación descrita en la Figura 17)

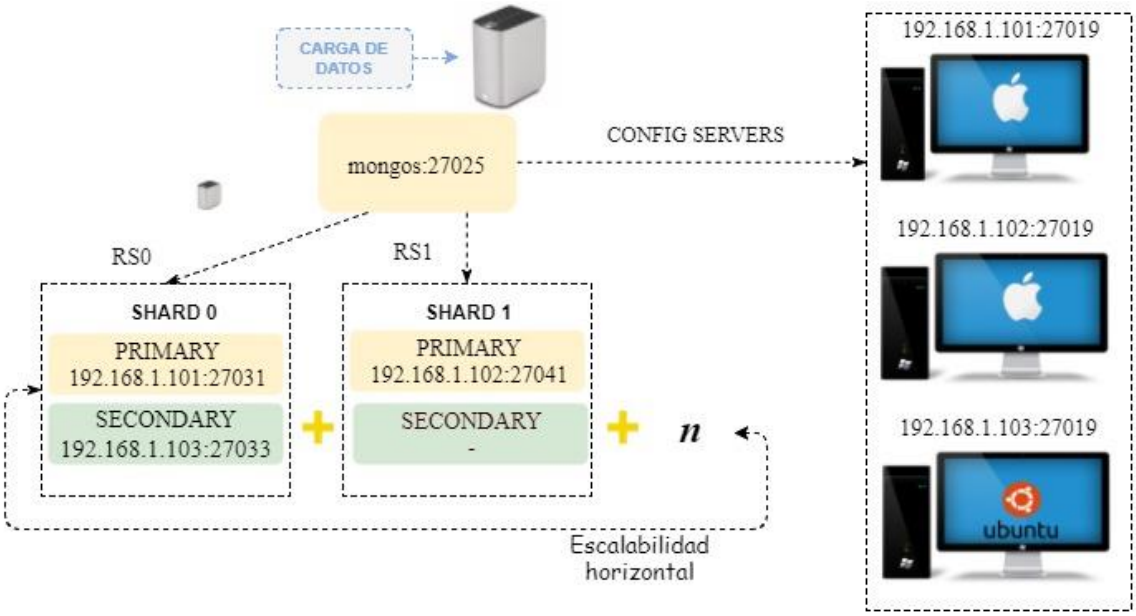



Figura 18. Topología de MongoDB para sharding
Fuente: Elaboración propia

Según Gutierrez (2014) para fragmentar una base de datos, MongoDB se basa en un sistema de claves. Se escoge un campo (generalmente un índice o un índice compuesto) que sea

[7] Más información en: https://docs.mongodb.com/manual/sharding/

común a todos los documentos de la colección que se va a particionar y se divide los valores de ese campo entre los trozos. Y luego distribuye los trozos entre los fragmentos del clúster de una forma más o menos homogénea. En la Figura 18 se muestra el diagrama de configuración general de un clúster fragmentado en MongoDB.

### 3.2.2. Escalabilidad.

Una razón común para el uso de MongoDB según Córdova & Cuzco (2013) “es su esquema de colecciones y su capacidad de desempeño a gran escala”. En MongoDB se puede escalar horizontalmente con facilidad usando sharding, este mecanismo permite aumentar el número de fragmentos sin afectar el rendimiento del sistema.

En MongoDB la fragmentación es la capacidad de particionar una colección en subconjuntos de datos para almacenarlos en varios fragmentos como se observa en la Figura 18. Uno de los beneficios de la fragmentación según Mancilla (2013) es que “garantiza que la información esté disponible en cualquier momento, esto debido a que la información se encuentra dividida de forma lógica en distintas máquinas lo que ayuda a mejorar el consumo de recursos disponibles”.

### 3.2.3. Tolerancia a fallos.

MongoDB está situado en la categoría CP según el Teorema de CAP, es decir, que asegura consistencia y tolerancia a partición en sus servidores, pero puede que algunos datos no estén disponibles. Para garantizar la consistencia de datos MongoDB puede ser implementado en un ambiente replicado en donde varios servidores mantengan copias de la misma información.

La tolerancia a partición de MongoDB viene determinada en función del servidor en el que se produce la partición. Cuando un servidor se ha particionado, el sistema continua funcionando y es posible acceder a los datos a partir de las copias almacenadas en otros servidores. En MongoDB un conjunto de réplicas tiene la capacidad suficiente para muchas operaciones de lectura distribuidas.

### 3.2.4. Implementación y configuración de MongoDB.

#### *3.2.4.1. Replicación maestro-esclavo en MongoDB.*

El siguiente procedimiento describe los pasos para la implementación de un conjunto de réplicas de tres miembros sobre la arquitectura que se muestra en la Figura 17. En MongoDB tres miembros en el conjunto de réplicas proporcionan la redundancia suficiente para sobrevivir a la mayoría de las particiones de red y otros fallos del sistema. Generalmente los

conjuntos de réplicas deben tener un número impar de miembros para asegurar que las elecciones se realicen sin inconvenientes.

### 1) Inicio de los servidores en MongoDB

En el servidor primario la base de datos se almacena en el disco externo, es por ello que para iniciarlo se usa el parámetro `--dbpath` especificando la dirección donde se almacenarán los datos `(/Volumes/DiscoExterno/MongoDB/data)`.

Para cada miembro, iniciar una instancia mongod especificando el nombre del conjunto de réplicas a través del parámetro `--replSet`.

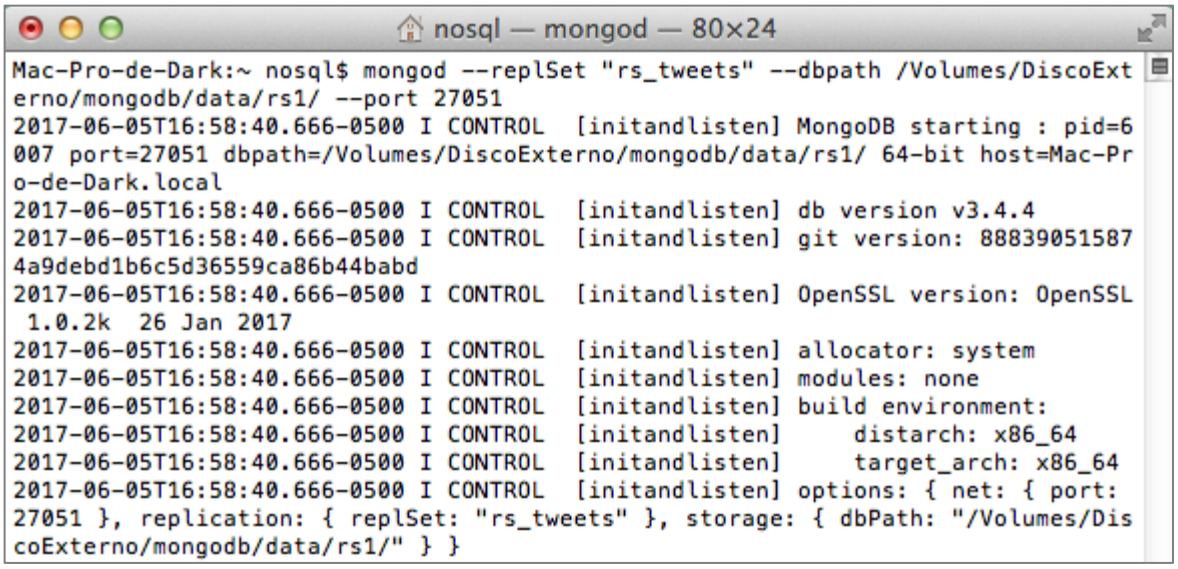



Figura 19. Inicialización de la instancia mongod en el servidor primario
Fuente: Elaboración propia

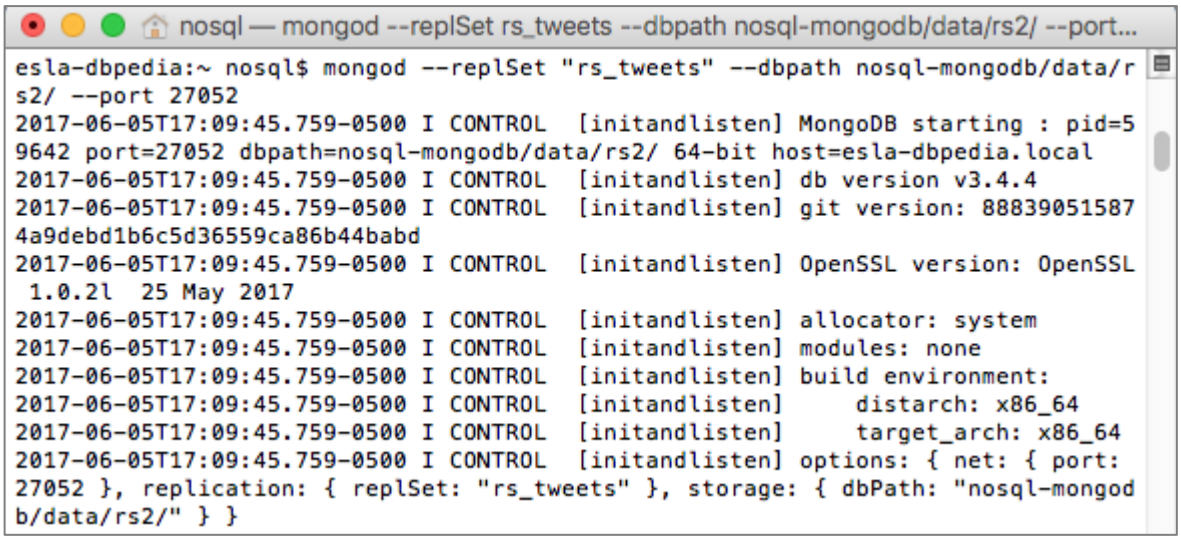



Figura 20. Inicialización de la instancia mongod en el primer servidor secundario
Fuente: Elaboración propia

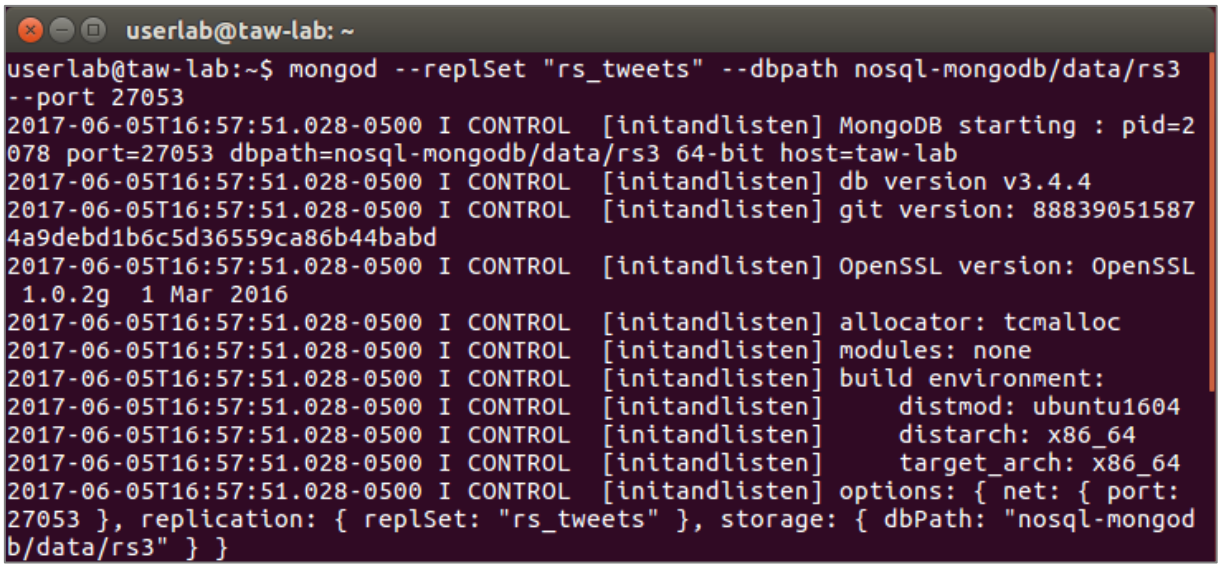



Figura 21. Inicialización de la instancia mongod en el segundo servidor secundario
Fuente: Elaboración propia

### 2) Conectarse a uno de los miembros del conjunto de réplicas

Al miembro que se acceda será automáticamente seleccionado como el servidor primario. La Figura 22 muestra la línea de comandos utilizada para conectarse al servidor primario.

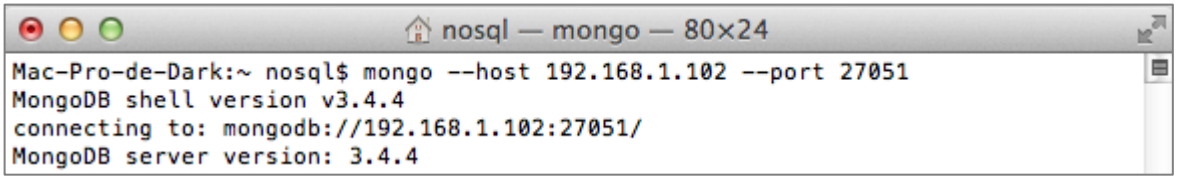



Figura 22. Comando para acceder a un miembro del conjunto de réplicas
Fuente: Elaboración propia

### 3) Inicialización del conjunto de réplicas

Para iniciar un conjunto de réplicas utilizar el método `rs.initiate()` en el miembro primario (Paso 2). En el campo `_id` se especifica el nombre del conjunto de réplicas y en el campo `members` se colocan las direcciones de todos los miembros del conjunto de réplicas.

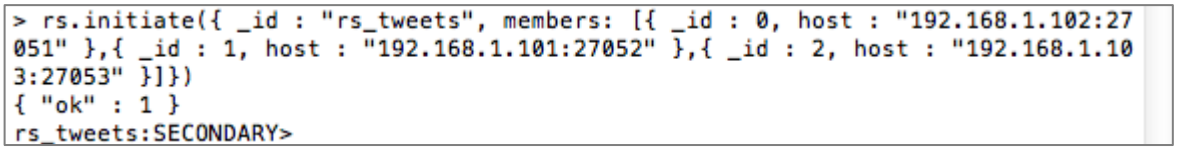



Figura 23. Comando para inicializar un conjunto de réplicas
Fuente: Elaboración propia

### 4) Verificación de la configuración del conjunto de réplicas

Para visualizar la configuración y el estado del conjunto de réplicas se usa el método `rs.status()`. Si la configuración del conjunto de réplicas fue exitoso se habrá elegido un

servidor primario y dos servidores secundarios. Es necesario esperar que MongoDB realice la sincronización entre todos los servidores.


```
"members" : [
        {
                "_id" : 0,
                "name" : "192.168.1.102:27051",
                "health" : 1,
                "state" : 1,
                "stateStr" : "PRIMARY",
                "uptime" : 80969,
                "optime" : {
                        "ts" : Timestamp(1496781575, 1),
                        "t" : NumberLong(1)
                },
                "optimeDate" : ISODate("2017-06-06T20:39:35Z"),
                "electionTime" : Timestamp(1496700682, 2),
                "electionDate" : ISODate("2017-06-05T22:11:22Z"),
                "configVersion" : 3,
                "self" : true
        },
        {
                "_id" : 1,
                "name" : "192.168.1.101:27052",
                "health" : 1,
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 77441,
                "optime" : {
                        "ts" : Timestamp(1496781575, 1),
                        "t" : NumberLong(1)
                },
                "optimeDurable" : {
                        "ts" : Timestamp(1496781575, 1),
                        "t" : NumberLong(1)
                },
                "optimeDate" : ISODate("2017-06-06T20:39:35Z"),
                "optimeDurableDate" : ISODate("2017-06-06T20:39:35Z"),
                "lastHeartbeat" : ISODate("2017-06-06T20:39:40.145Z"),
                "lastHeartbeatRecv" : ISODate("2017-06-06T20:39:40.107Z"),
                "pingMs" : NumberLong(3),
                "syncingTo" : "192.168.1.103:27053",
                "configVersion" : 3
        },
        {
                "_id" : 2,
                "name" : "192.168.1.103:27053",
                "health" : 1,
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 77434,
                "optime" : {
                        "ts" : Timestamp(1496781575, 1),
                        "t" : NumberLong(1)
                },
                "optimeDurable" : {
                        "ts" : Timestamp(1496781575, 1),
                        "t" : NumberLong(1)
                },
                "optimeDate" : ISODate("2017-06-06T20:39:35Z"),
                "optimeDurableDate" : ISODate("2017-06-06T20:39:35Z"),
                "lastHeartbeat" : ISODate("2017-06-06T20:39:38.175Z"),
                "lastHeartbeatRecv" : ISODate("2017-06-06T20:39:38.213Z"),
                "pingMs" : NumberLong(3),
                "syncingTo" : "192.168.1.102:27051",
                "configVersion" : 3
        }
],
"ok" : 1
```


Figura 24. Estado del conjunto de réplicas
Fuente: Elaboración propia

### 5) Creación de base de datos

En el servidor primario se debe crear la base de datos que se va a replicar. Cuando se crea la base de datos en el servidor primario de forma automática se replicará la misma en los servidores secundarios. En la Figura 25 se usa el comando “use” para crear la base de datos,

en este caso ya se han importado los datos por lo cual con el método `db.getCollection` se ha obtenido que el número de registros almacenados.

```
rs_tweets:PRIMARY> use db_tweets
switched to db db_tweets
rs_tweets:PRIMARY> db.getCollection('tweets').find({}).count()
20000000
```

Figura 25. Comando para crear una base de datos en MongoDB
Fuente: Elaboración propia

#### *3.2.4.2. Sharding en MongoDB.*

El siguiente procedimiento implica la creación de un clúster fragmentado en MongoDB, que consiste en un proceso mongos, dos servidores de configuración y dos fragmentos sobre la arquitectura que se muestra en la Figura 18.

**1) Configuración de los servidores de configuración**

Para indicarle a una instancia mongod que es un servidor de configuración, se crea un archivo de configuración `mongod_csrs.conf`, en donde se establece `sharding.clusterRole` a `configsvr`, y `replication.replSetName` con el nombre del conjunto de réplicas de los servidores de configuración. En la Figura 26 se muestra el archivo de configuración creado, se pueden incluir configuraciones adicionales según sea apropiado en la implementación.

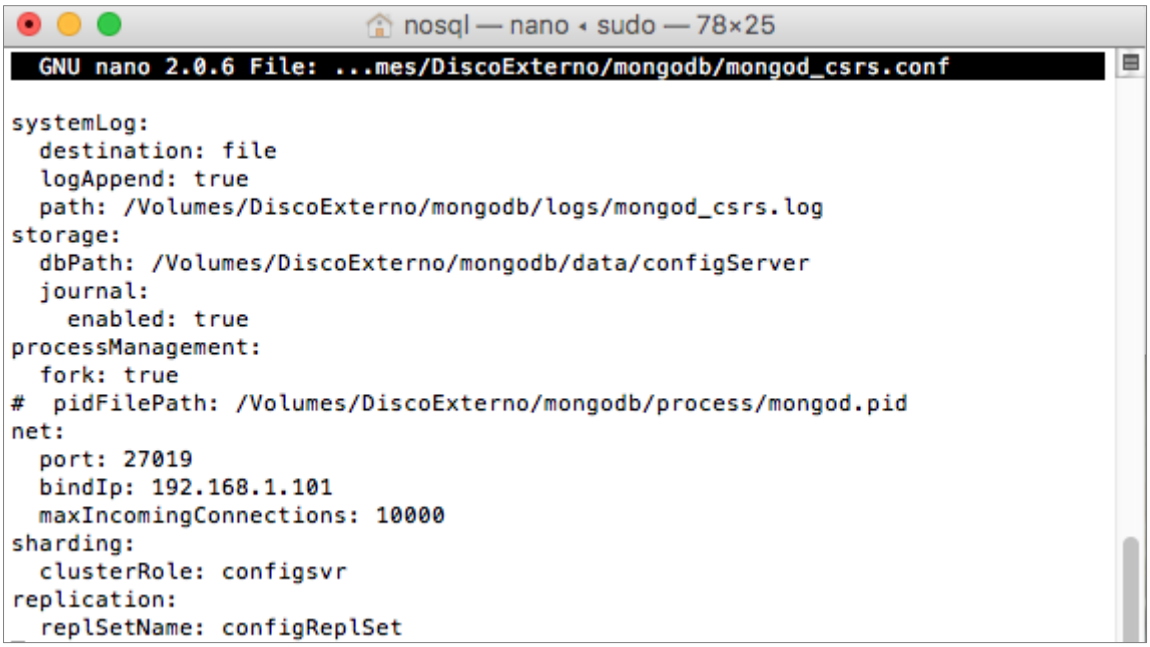



Figura 26. Archivo de configuración para servidores de configuración
Fuente: Elaboración propia

En las Figuras 27 y 28 se inician cada una las instancias mongod que se configuraron como los servidores de configuración.

**Inicio de las instancias mongod para los servidores de configuración**

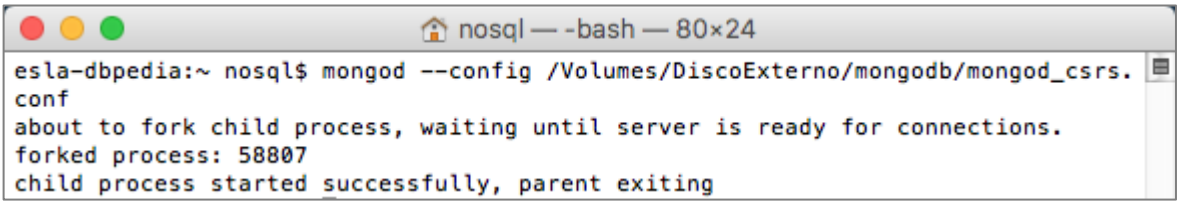



Figura 27. Inicialización de la instancia Config Server 1
Fuente: Elaboración propia

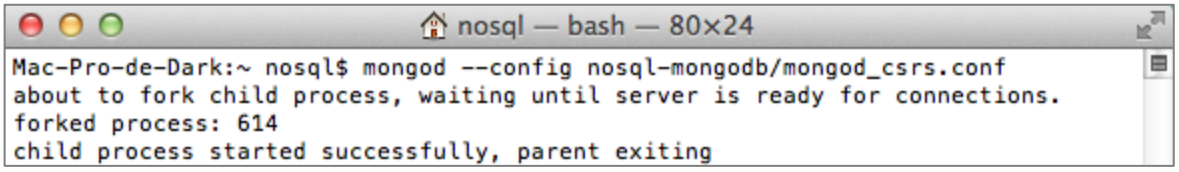



Figura 28. Inicialización de la instancia Config Server 2
Fuente: Elaboración propia

Luego es necesario iniciar el conjunto de réplicas utilizando el método `rs.initiate()`, para ello se debe acceder a una de las instancias mongod de los miembros de los servidores de configuración. A través de este método se especifican los siguientes parámetros.

- `_id`: Debe ser el mismo nombre especificado en el parámetro `replSetName` de las instancias Config Server
- `members`: Especifican los host y puertos de las instancias Config Server
- `configsvr`: Se establece en true para el conjunto de réplicas de las instancias Config Server


Figura 29. Inicialización del conjunto de réplicas de los servidores de configuración
Fuente: Elaboración propia

Una vez que el conjunto de réplicas de los servidores de configuración (CSRS) se inicia, se procede a la creación de los fragmentos como conjunto de réplicas.

**2) Configuraciones de los fragmentos como conjunto de réplicas**

Para indicarle a una instancia mongod que es un fragmento del clúster, se crea un archivo de configuración `mongod_rsX.conf` y se establece el parámetro `sharding.clusterRole` a `shardsvr`, y `replication.replSetName` con el nombre del conjunto de réplicas

(Ejemplo: rs0). En la Figura 30 se muestra el archivo de configuración creado, se puede incluir configuraciones adicionales según sea apropiado en la implementación.

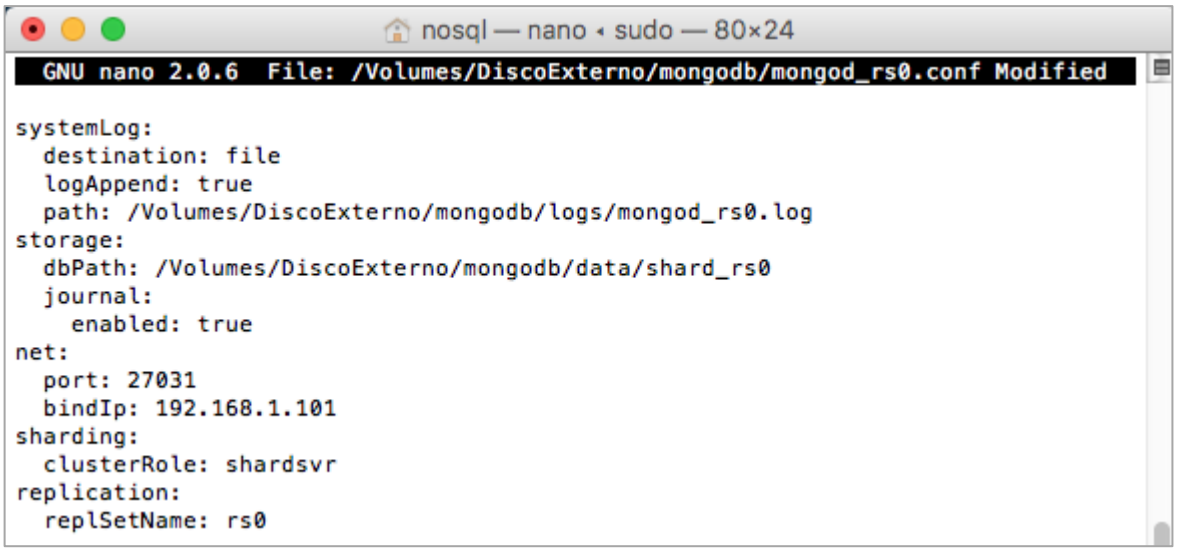



Figura 30. Archivo de configuración para un fragmento del conjunto de réplicas rs0
Fuente: Elaboración propia

Por cada fragmento que se añada al clúster se debe crear un archivo de configuración especificando los mismos parámetros, con excepción de los parámetros `port`, `bindIp` y `las direcciones de almacenamiento y logs` que deberán ser modificados para cada instancia mongod. A continuación se inician cada una las instancias mongod que se configuraron como miembros del conjunto de réplicas de cada fragmento rs0 y rs1.

**Inicio de las instancias mongod para el conjunto de réplicas**

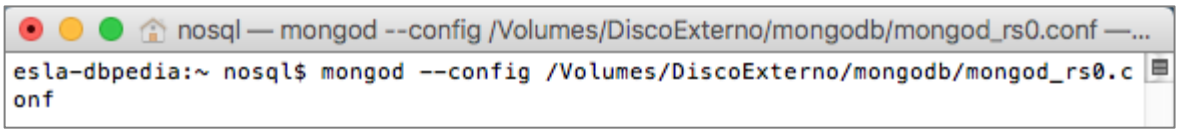



Figura 31. Inicio de miembro primario del conjunto de réplicas del fragmento rs0
Fuente: Elaboración propia

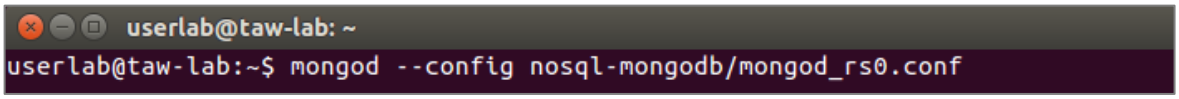



Figura 32. Inicio de miembro secundario del conjunto de réplicas del fragmento rs0
Fuente: Elaboración propia

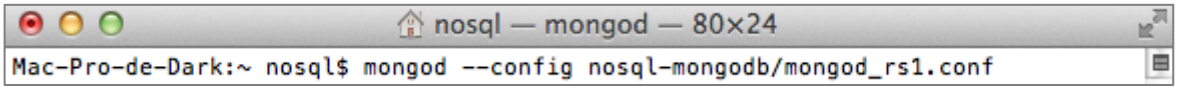



Figura 33. Inicio de miembro primario del conjunto de réplicas del fragmento rs1
Fuente: Elaboración propia

El siguiente paso es iniciar el conjunto de réplicas utilizando el método `rs.initiate()`, para ello conectarse a una instancia mongod de uno de los miembros del conjunto de réplicas. Este método desencadena una elección para elegir a uno de los miembros como el miembro primario.

- `_id`: Debe ser el mismo nombre especificado en el parámetro `replSetName` de las instancias de los miembros del conjunto de réplicas
- `members`: Especifican los host y puertos de los miembros del conjunto de réplicas

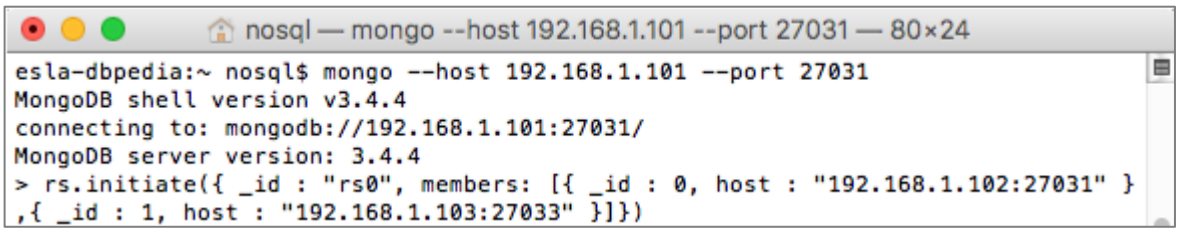



Figura 34. Inicialización de fragmento como conjunto de réplicas rs0
Fuente: Elaboración propia

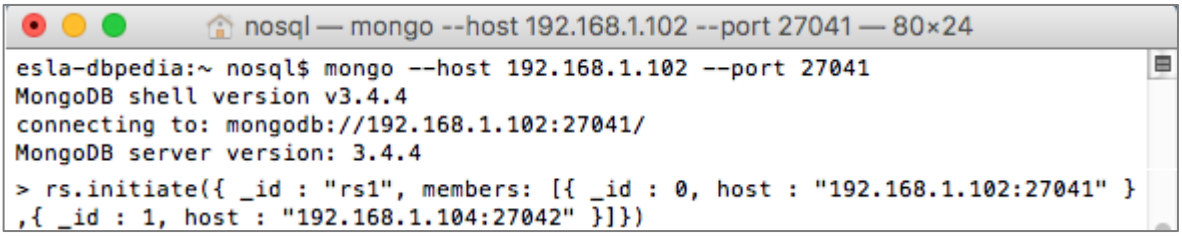



Figura 35. Inicialización de fragmento como conjunto de réplicas rs1
Fuente: Elaboración propia

Una vez que el conjunto de réplicas (RS) se inicia, es necesario conectarse al miembro primario. Se utiliza `rs.status()` para localizar el miembro primario.

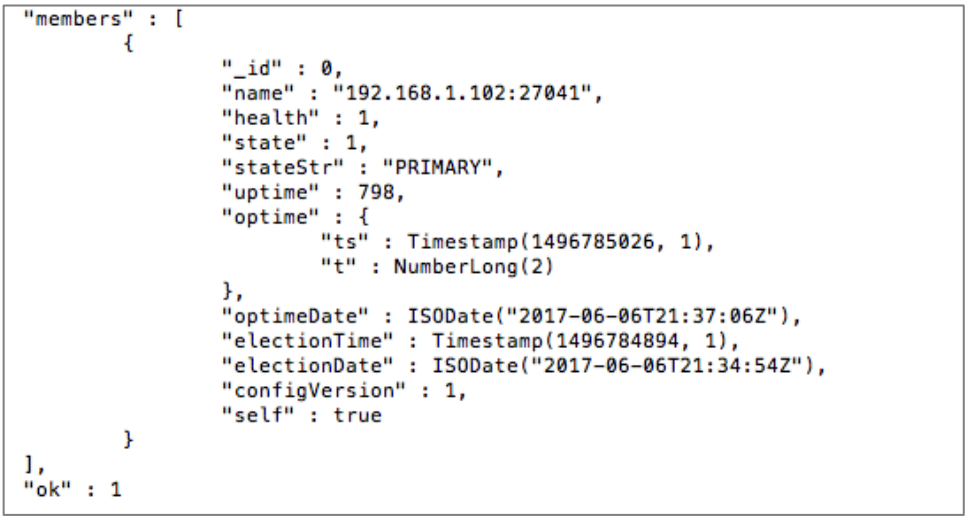



Figura 36. Identificación del miembro primario del fragmento rs1
Fuente: Elaboración propia

```
"members" : [
        {
                "_id" : 0,
                "name" : "192.168.1.101:27031",
                "health" : 1,
                "state" : 1,
                "stateStr" : "PRIMARY",
                "uptime" : 216,
                "optime" : {
                        "ts" : Timestamp(1496785018, 1),
                        "t" : NumberLong(3)
                },
                "optimeDate" : ISODate("2017-06-06T21:36:58Z"),
                "electionTime" : Timestamp(1496784886, 1),
                "electionDate" : ISODate("2017-06-06T21:34:46Z"),
                "configVersion" : 2,
                "self" : true
        },
        {
                "_id" : 1,
                "name" : "192.168.1.103:27033",
                "health" : 1,
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 146,
                "optime" : {
                        "ts" : Timestamp(1496785018, 1),
                        "t" : NumberLong(3)
                },
                "optimeDurable" : {
                        "ts" : Timestamp(1496785018, 1),
                        "t" : NumberLong(3)
                },
                "optimeDate" : ISODate("2017-06-06T21:36:58Z"),
                "optimeDurableDate" : ISODate("2017-06-06T21:36:58Z"),
                "lastHeartbeat" : ISODate("2017-06-06T21:37:01.637Z"),
                "lastHeartbeatRecv" : ISODate("2017-06-06T21:37:01.636Z"),
                "pingMs" : NumberLong(3),
                "syncingTo" : "192.168.1.101:27031",
                "configVersion" : 2
        }
],
"ok" : 1
```


Figura 37. Identificación del miembro primario del fragmento rs0
Fuente: Elaboración propia

### 3) Configuración de mongos

Para iniciar una instancia mongos, se crea un archivo de configuración `mongos.conf` y se establece el parámetro `sharding.configDB` en `<replSetName>/<host:port>`.


Figura 38. Archivo de configuración para instancia mongos
Fuente: Elaboración propia

En la Figura 38 se muestra el archivo de configuración mongos. Luego se inicia la instancia mongos con la opción `--config` y la ruta de acceso al archivo de configuración.

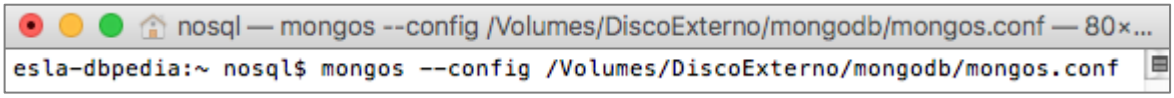



Figura 39. Inicialización de mongos
Fuente: Elaboración propia

### 4) Configuraciones de sharding

Las configuraciones de sharding de una base de datos se realizan sobre la instancia mongos.

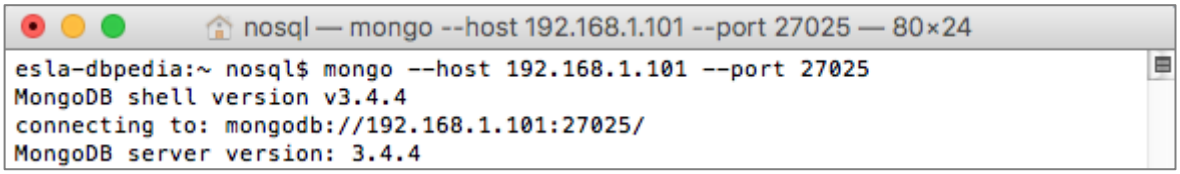



Figura 40. Conectarse a la instancia mongos
Fuente: Elaboración propia

Para agregar fragmentos al clúster se usa el método `sh.addShard()` especificando el nombre del conjunto de réplicas y los miembros del conjunto. La siguiente operación añade dos fragmentos como conjunto de réplicas al clúster.

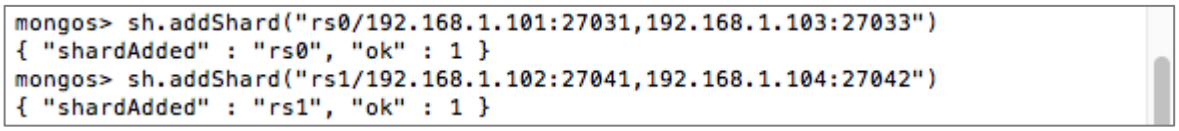



Figura 41. Añadir fragmentos al clúster
Fuente: Elaboración propia

Una vez añadidos los fragmentos para hacer posible la fragmentación de colecciones debe habilitarse la fragmentación de la base de datos utilizando el método `sh.enableSharding()`.

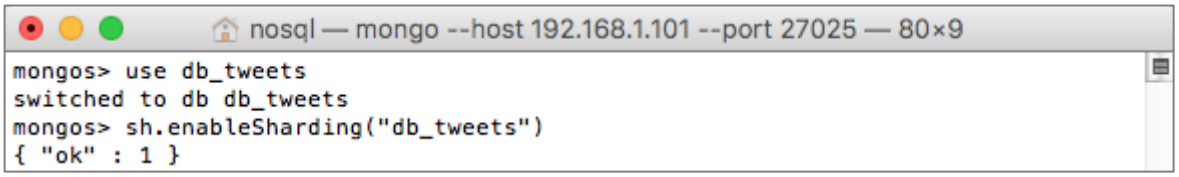



Figura 42. Habilitar fragmentación a base de datos
Fuente: Elaboración propia

Asimismo conectado al mongos utilice el método `sh.shardCollection()` para fragmentar la colección de la base de datos, se debe especificar el nombre de la colección y un documento que contiene la clave fragmento.

```
mongos> sh.shardCollection("db_tweets.tweets",{"_id":"hashed"})
{ "collectionsharded" : "db_tweets.tweets", "ok" : 1 }
```

Figura 43. Fragmentación de una colección
Fuente: Elaboración propia

En la documentación oficial de MongoDB se proporciona recomendaciones para seleccionar la clave de fragmento. En este caso, se establece el campo `_id` como la clave de fragmento debido a su cardinalidad al tener valores únicos. Asimismo para garantizar una distribución uniforme de los datos se fija el tipo `hashed` como valor de la clave de fragmento ( Figura 43).

### 3.2.5. Importación de datos en MongoDB.

En MongoDB se realiza la carga de datos desde el servidor donde se ejecuta la instancia mongos. Se utiliza la herramienta `mongoimport`[8] con los siguientes parámetros para optimizar el tiempo a la hora cargar los datos.

- `host`: Indica la dirección IP del servidor donde se ejecuta la instancia mongos, que realiza la sincronización de los Config Server
- `port`: Indica el puerto del servidor donde se ejecuta la instancia mongos
- `db`: Es el nombre de la base de datos
- `collection`: Es el nombre de la colección
- `file`: Especifica la dirección del archivo de datos que va a importase a la base de datos
- `jsonArray`: Indica que el formato del archivo de origen es JSON
- `numInsertionWorkers`: Especifica el número de procesos de inserción que se ejecutan simultáneamente para cargar los datos

En la Figura 44 se puede observar que en el parámetro `–numInsertionWorkers` se especifica que la importación de los datos se realice con 150 procesos de inserción simultáneos con fines de optimización del tiempo de carga de los datos.

[8] Más información en: https://docs.mongodb.com/manual/reference/program/mongoimport/

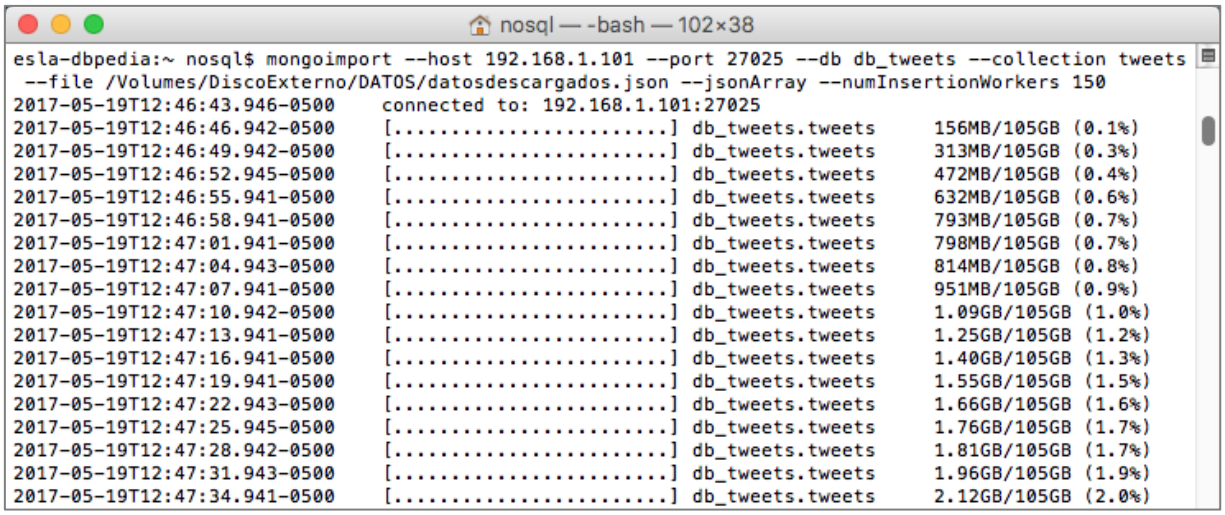



Figura 44. Comando para importar datos en MongoDB
Fuente: Elaboración propia

La finalización de la carga de datos se muestra en la Figura 45.

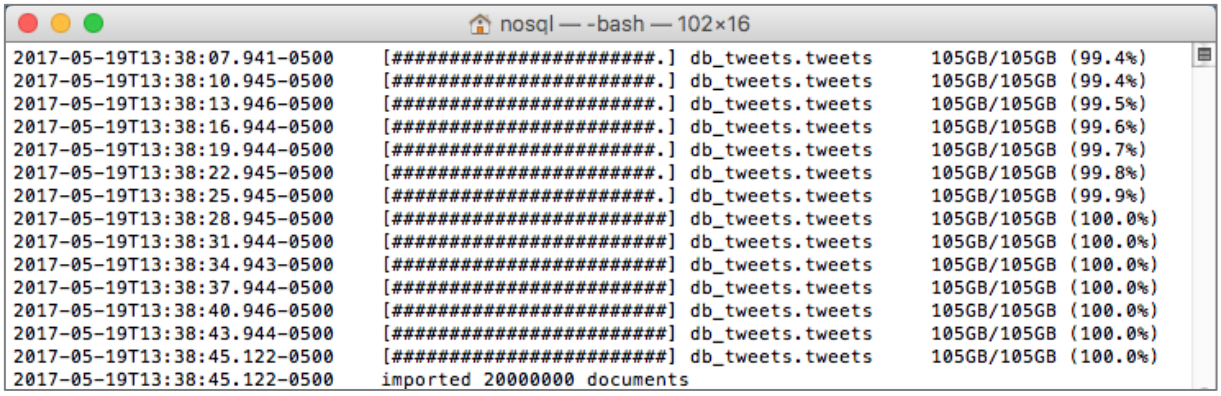



Figura 45. Terminación de la importación de datos en MongoDB
Fuente: Elaboración propia

Una vez culminada la importación de todos los datos se puede utilizar el método `db.collection.getShardDistribution()` para mostrar las estadísticas de distribución de datos de la colección fragmentada.

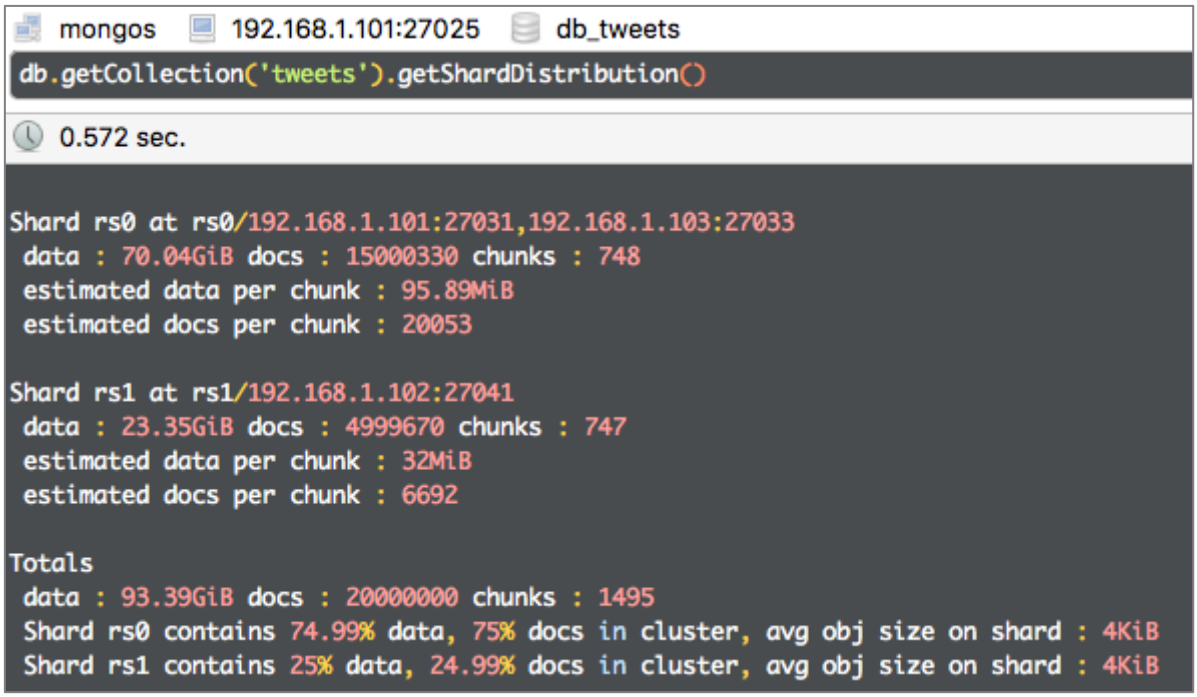



Figura 46. Distribución de datos en el clúster fragmentado MongoDB
Fuente: Elaboración propia

## 3.3. CouchDB

### 3.3.1. Arquitectura.

#### *3.3.1.1. Replicación.*

En CouchDB, se llama replicación al proceso de sincronización de datos entre múltiples servidores. Básicamente consiste en contar con varias copias de la misma información en cada uno de los servidores que forman parte del sistema distribuido. En CouchDB, la replicación adquiere un papel fundamental pues es necesaria para mantener la consistencia de los datos entre los servidores de la base de datos. Según Issa & Schiltz (2016) el proceso de replicación puede propagarse entre múltiples servidores y cuando una base de datos activa una replicación los servidores comparan sus datos y luego el servidor principal envía cambios a los servidores secundarios.

Según Gaona & Sandoval (2013) una base de datos CouchDB no tiene límites de latencia, es decir, uno de los servidores puede estar desconectado y recibir varios cambios, los cuales se propagaran cuando el servidor se vuelve a conectar. Esto significa que la replicación garantiza una mejora en el acceso a la lectura de los datos y alta disponibilidad, a través de las copias de datos almacenadas en otros servidores. En la Figura 47 se muestra el modelo de la arquitectura que se implementará con la base de datos CouchDB.

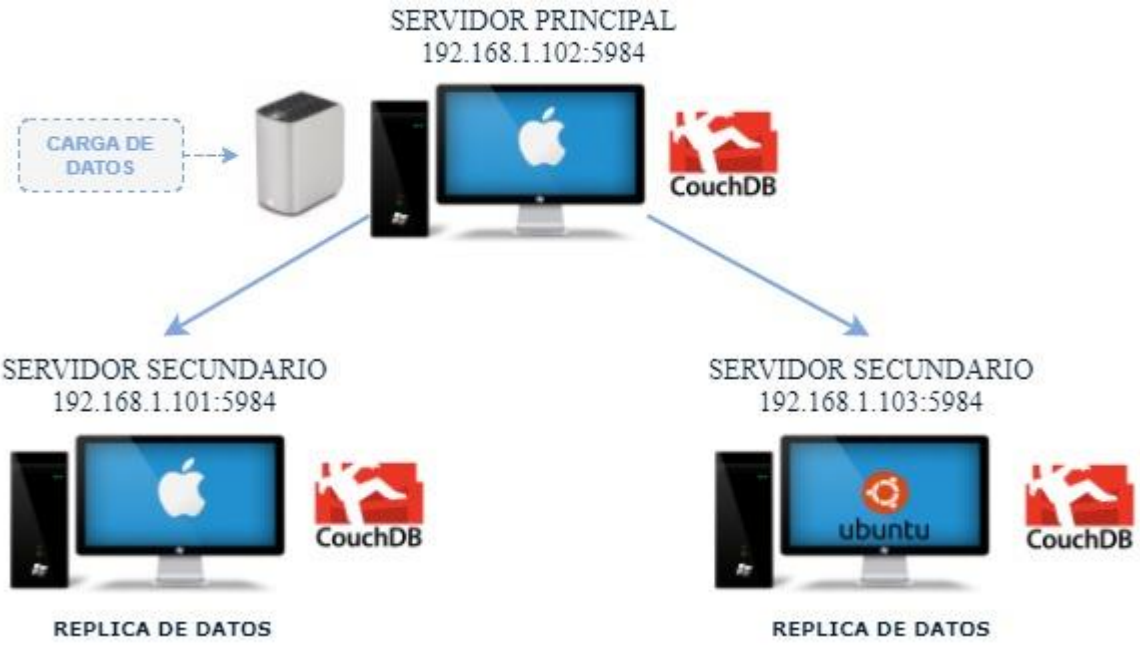



Figura 47. Topología de CouchDB para replicación
Fuente: Elaboración propia

#### *3.3.1.2. Sharding.*

A medida que los datos crecen y el número de usuarios crece es necesario escalar (CouchDB, 2017). La fragmentación en CouchDB divide una base de datos en una serie de trozos o fragmentos iguales pero separados que contiene los documentos. Un documento siempre pertenecerá a un fragmento y esto depende directamente sólo de su ID, lo que significa que cualquier nodo en el clúster CouchDB sabe exactamente donde se aloja cada documento. Capdevila (2015) afirma que “el sharding puede mejorar sustancialmente la performance general ya que se ocupa tanto de escrituras como lecturas, haciéndola una mejor opción si el sistema debe manejar muchas operaciones de escritura” (p.31).

Cuando se crea una base de datos en CouchDB, se puede especificar el número de fragmentos con $q$ y el número de copias de cada fragmento con $n$. Por omisión, los valores son $q = 8$ y $n = 3$. Para la mayoría de los usos $q = 8$ es el más adecuado, y para $n$ un valor mayor que $3$ resulta demasiado caro. En la Figura 48 se muestra el modelo de la arquitectura que se implementará para realizar las configuraciones de fragmentación de la base de datos CouchDB.

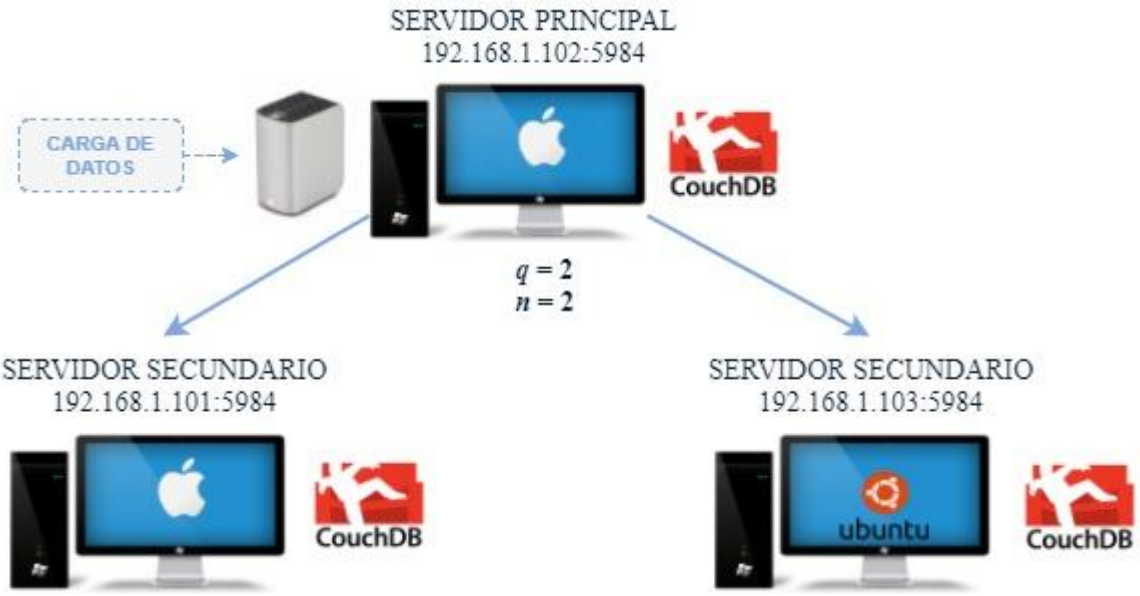



Figura 48. Topología de CouchDB para sharding
Fuente: Elaboración propia

### 3.3.2. Escalabilidad.

CouchDB ofrece uno de los sistemas más flexibles y configurables de escalabilidad (Issa & Schiltz, 2016). Una de las razones es su facilidad para replicar y añadir nodos. Wallace (2015), afirma “CouchDB sigue el estilo de clusterización de Dynamo, que permite a los datos y las aplicaciones escalar a más de cientos de nodos para alcanzar mayor rendimiento y capacidad de almacenamiento”.

La replicación hace la escalabilidad en un sistema CouchDB muy sencilla. Por ello, en este trabajo se implementa la replicación maestro-esclavo, en donde el maestro está representado por el servidor principal cuya función consiste en mantener sincronizadas las instancias de CouchDB y por otro lado, los servidores esclavos encargados de almacenar copias de datos de la base de datos maestra.

### 3.3.3. Tolerancia a fallos.

CouchDB se diseñó para ofrecer disponibilidad y tolerancia a fallos a cambio de sacrificar la consistencia de datos. La fragmentación en CouchDB divide una base de datos en fragmentos y crea múltiples copias de cada fragmento para distribuirlos a través de los diferentes servidores, de modo que, la pérdida de un servidor no es fatal y el sistema seguirá funcionando.

Según Guamán (2014) “CouchDB garantiza consistencia eventual para poder ofrecer tanto disponibilidad como tolerancia a las particiones”. El blog oficial de CouchDB explica que cuando se crea, actualiza, elimina o lee un documento, el nodo que procesa la solicitud HTTP

genera $N$ procesos en paralelo para hacer la operación en cada copia del documento. El nodo principal espera a $N/2 + 1$ respuestas antes de combinar esas respuestas como la respuesta HTTP. Y este solapamiento ayuda a presentar una visión consistente de la base de datos, aunque no se garantiza esa consistencia (CouchDB Blog, 2016).

### 3.3.4. Implementación y configuración de CouchDB.

#### *3.3.4.1. Replicación maestro – esclavo en CouchDB.*

En CouchDB una replicación puede ser local o remota. Un replicación local es cuando dos bases de datos se están ejecutando sobre el mismo servidor CouchDB y un replicación remota es cuando el destino es una base de datos que se ejecuta en otro servidor. El siguiente procedimiento detalla la configuración para realizar una replicación remota sobre la arquitectura que se muestra en la Figura 47.

*Configuración de archivos*

- En el archivo `vm.args` en el parámetro `-name CouchDB@localhost`, cambie `localhost` por la IP del servidor. En Mac Os X el archivo se localiza en `/Users/CouchDB/CouchDB/etc/vm.args` y en Ubuntu el archivo se localiza en `/home/CouchDB/etc/vm.args`. De modo que:

$$-name\ couchdb@localhost \rightarrow -name\ couchdb@0.0.0.0$$

- En los archivos `local.ini y default.ini`, cambie todos los `localhost` por la IP del servidor. En Mac Os X y Ubuntu, ambos archivos se localizan en el mismo directorio que el archivo `vm.args`.

**1) Inicio de los servidores en CouchDB**

En el servidor principal la base de datos se almacena en el disco externo, es por ello que para iniciarlo se ha modificado en `default.ini`, el parámetro `database_dir` con la dirección donde se almacenarán los datos (`/Volumes/DiscoExterno/CouchDB/data)`, por defecto se almacenan en `./data`.

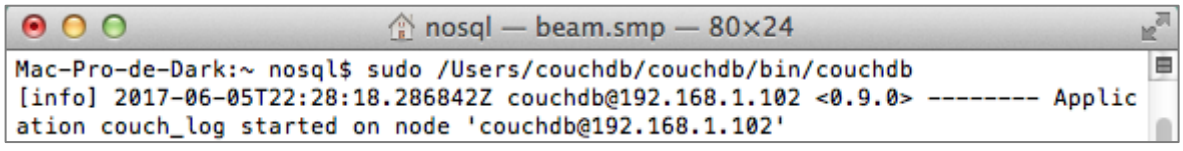



Figura 49. Inicialización del servidor principal CouchDB
Fuente: Elaboración propia

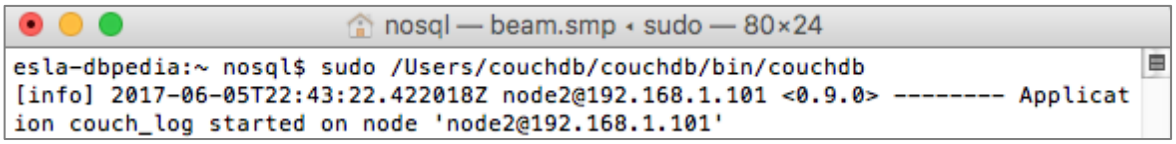



Figura 50. Inicialización del primer servidor secundario CouchDB
Fuente: Elaboración propia

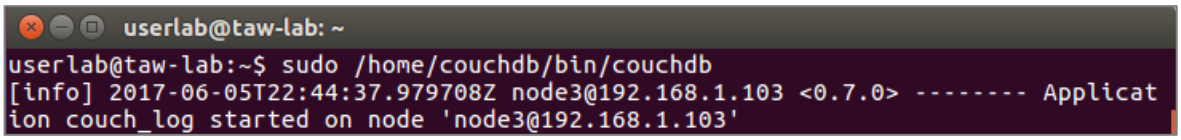



Figura 51. Inicialización del segundo servidor secundario CouchDB
Fuente: Elaboración propia

### 2) Creación de base de datos

En el servidor principal crear una base de datos y en los servidores secundarios de la misma manera crear una base datos con igual o distinto nombre. Para crear una base de datos se usa el comando:

```
curl -X PUT http://Admin:Admin@192.168.1.102:5984/base_tw
```

Para eliminar una base de datos en CouchDB utilizar el siguiente comando:

```
curl -X DELETE http://Admin:Admin@192.168.1.102:5984/base_tw
```

### 3) Configuraciones para replicación

La replicación es una operación puntual que envía una petición HTTP a CouchDB incluyendo una base de datos origen (*source*) y una de destino (*target*). En la Figura 52 y 53, se envía una petición para replicar todos los documentos de la base de datos origen a la base de datos destino en los servidores secundarios.

**Replicación de datos hacia el servidor secundario (192.168.1.101)**

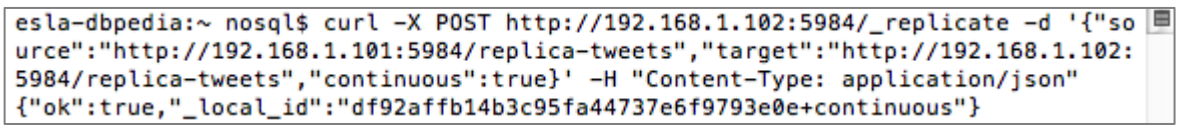



Figura 52. Inicialización del proceso de replicación al primer servidor secundario
Fuente: Elaboración propia

**Replicación de datos hacia el servidor secundario (192.168.1.103)**

```
esla-dbpedia:~ nosql$ curl -X POST http://192.168.1.102:5984/_replicate -d '{"so
urce":"http://192.168.1.101:5984/replica-tweets","target":"http://192.168.1.103:
5984/replica-tweets","continuous":true}' -H "Content-Type: application/json"
{"ok":true,"_local_id":"d03e06e997fa8c76a9f7c1e3b66d2519+continuous"}
```

Figura 53. Inicialización del proceso de replicación al segundo servidor secundario
Fuente: Elaboración propia

#### *3.3.4.2. Sharding en CouchDB.*

El siguiente procedimiento detalla la configuración para fragmentar una base de datos en CouchDB sobre la arquitectura que se muestra en la Figura 48. Para iniciar la fragmentación de datos se debe realizar la configuración de los archivos necesarios, del mismo modo que en la sección 3.3.4.1.

**1) Inicio de los servidores en CouchDB**

Iniciar el servidor principal y servidores secundarios, del mismo modo que en la sección 3.3.4.1

**2) Creación base de datos**

En el servidor principal crear una base de datos con dos fragmentos y dos replicas. En la Figura 54, $n$ es el número de réplicas por fragmento y $q$ es el número de fragmentos.


Figura 54. Fragmentación de una base de datos con dos fragmentos y dos réplicas
Fuente: Elaboración propia

Luego para comprobar que los fragmentos se han creado acceder a los metadatos de la base de datos creada como se muestra en la Figura 55.


Figura 55. Comando para obtener la metadata de una base de datos CouchDB
Fuente: Elaboración propia

La Figura 56 muestra los metadatos de la base de datos crawlerdb, en donde:

- `_id` Es el nombre de la base de datos
- `_rev` Es la revisión actual de los metadatos

- `shard_suffix` Los números después de db_tweets y antes de .couch. El número de segundos después que se creó la base de datos según Unix epoch.
- `Changelog` Contiene explicaciones. Sólo los administradores pueden leer.
- `by_node` Fragmentos que tiene cada nodo.
- `by_range` En que nodos está cada fragmento.


```
{
  "_id": "db_tweets",
  "_rev": "12-08a4925730ebb70f79cbe8a02705f26d",
  "shard_suffix": [
    46,
    49,
    52,
    57,
    49,
    53,
    50,
    57,
    51,
    54,
    57
  ],
  "changelog": [
    [
      "add",
      "00000000-7fffffff",
      "couchdb@192.168.1.102"
    ],
    [
      "add",
      "80000000-ffffffff",
      "couchdb@192.168.1.102"
    ]
  ],
  "by_node": {
    "couchdb@192.168.1.102": [
      "00000000-7fffffff",
      "80000000-ffffffff"
    ]
  },
  "by_range": {
    "00000000-7fffffff": [
      "couchdb@192.168.1.102"
    ],
    "80000000-ffffffff": [
      "couchdb@192.168.1.102"
    ]
  }
}
```


Figura 56. Metadata de base de datos db_tweets
Fuente: Elaboración propia

### 3) Añadir servidores secundarios al servidor principal

El mecanismo de Sharding en CouchDB requiere que los servidores sean añadidos al servidor principal. En la Figura 57 se muestra el comando para agregar los servidores o nodos secundarios.

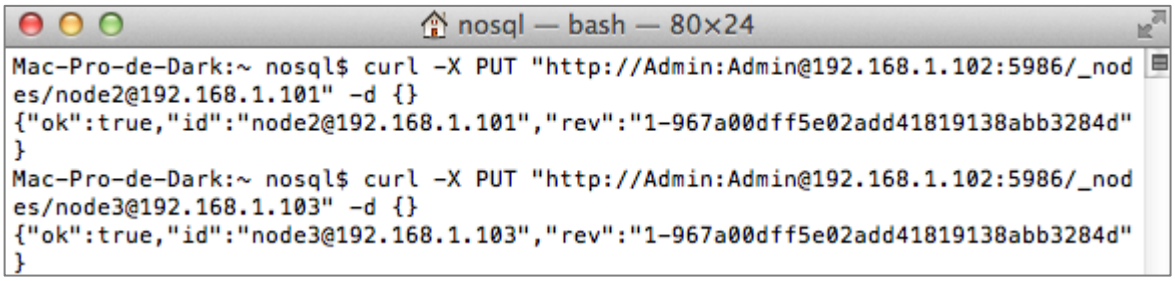



Figura 57. Añadiendo servidores secundarios al servidor principal CouchDB
Fuente: Elaboración propia

Para asegurarse que los servidores están comunicándose entre sí, comprobar que se encuentren añadidos en `http://192.168.1.101:5984/_membership` de la interfaz administrativa Fauxton de CouchDB.

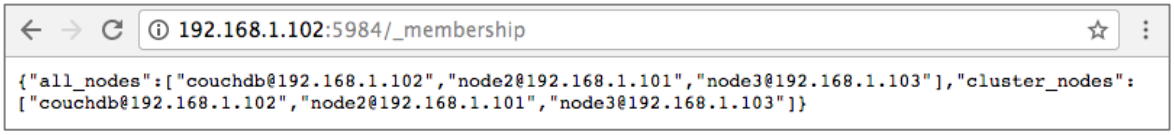



Figura 58. Servidores añadidos al servidor principal CouchDB
Fuente: Elaboración propia

### 4) Configuraciones para sharding

Es necesario modificar los metadatos de la base de datos para que CouchDB distribuya los fragmentos entre los servidores secundarios del entorno distribuido. De modo que, en comparación con la Figura 56 la metadata modificada quedaría así:

```
{
  "_id": "db_tweets",
  "_rev": "12-08a4925730ebb70f79cbe8a02705f26d",
  "shard_suffix": [
    46,
    49,
    52,
    57,
    49,
    53,
    50,
    57,
    51,
    54,
    57
  ],
  "changelog": [
    [
      "add",
      "00000000-7fffffff",
      "couchdb@192.168.1.102"
    ],
    [
      "add",
      "80000000-ffffffff",
      "couchdb@192.168.1.102"
    ],
    [
      "add",
      "00000000-7fffffff",
      "node2@192.168.1.101"
    ],
    [
      "add",
      "80000000-ffffffff",
      "node3@192.168.1.103"
    ]
  ],
  "by_node": {
    "couchdb@192.168.1.102": [
      "00000000-7fffffff",
      "80000000-ffffffff"
    ],
    "node2@192.168.1.101": [
      "00000000-7fffffff"
    ],
    "node3@192.168.1.103": [
      "80000000-ffffffff"
    ]
  },
  "by_range": {
    "00000000-7fffffff": [
      "couchdb@192.168.1.102",
      "node2@192.168.1.101"
    ],
    "80000000-ffffffff": [
      "couchdb@192.168.1.102",
      "node3@192.168.1.103"
    ]
  }
}
```


Figura 59. Metadata modificada para distribuir los datos
Fuente: Elaboración propia

En la Figura 60, se muestra el comando ejecutado para modificar la metadata de la base de datos. Cuando se actualiza la metadata los fragmentos se distribuyen en los servidores 192.168.1.101 y 192.168.1.103.

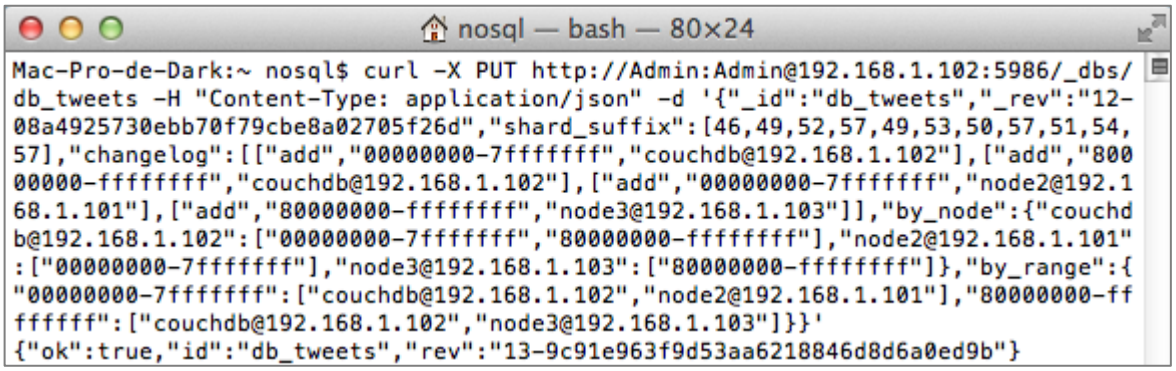



Figura 60. Comando para actualizar metadata de una base de datos CouchDB
Fuente: Elaboración propia

Si los fragmentos son demasiado grandes, entonces se puede copiarlos manualmente y luego sólo sincronizar CouchDB con los cambios de los últimos minutos.

### 3.3.5. Importación de datos en CouchDB

En CouchDB se utiliza la herramienta `couchimport`[9] para la importación de los datos a una base de datos CouchDB, especificando ciertos parámetros.

- `cat`: Es el archivo csv desde donde se va importar los datos
- `db`: Es el nombre de la base de datos donde se almacenará
- `buffer`: Define el tamaño de búfer a utilizar al escribir documentos en la base de datos (Por defecto es 500)
- `parallelism`: Es el parámetro que permite configurar el número de operaciones paralelas utilizadas para escribir datos (Por defecto es 1)
- `delimiter`: Define el delimitador de columna de los datos de entrada. En formato csv por defecto es ","
- `url`: Es la url del servidor CouchDB donde se va importar los datos

Debido a que la importación de datos es demasiado lenta cuando `parallelism` usa la configuración por defecto, en la Figura 61 se especifica que la importación de los datos se realice con 90 operaciones paralelas.

[9] Más información en: https://github.com/glynnbird/couchimport

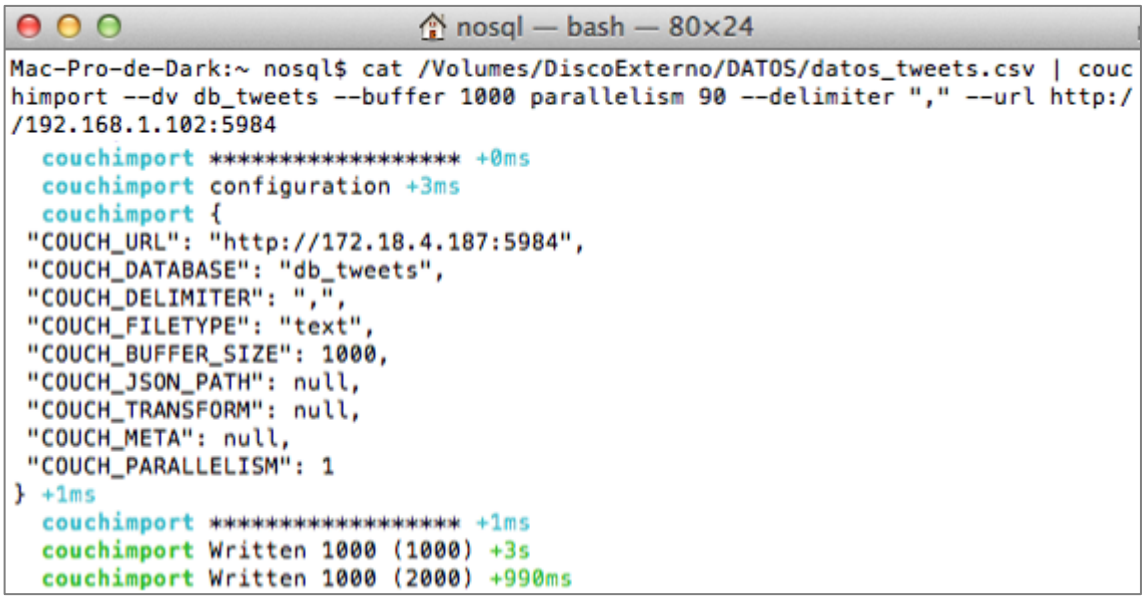



Figura 61. Comando para importar datos en CouchDB
Fuente: Elaboración propia

La finalización de la carga de datos se muestra en la Figura 62.

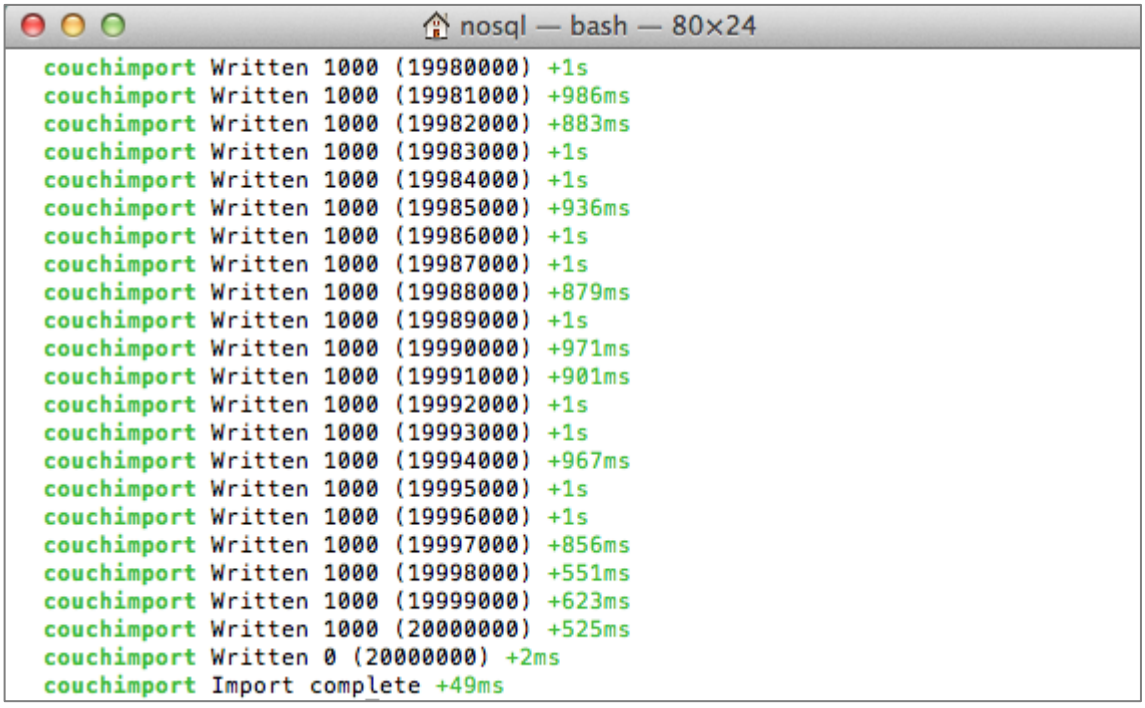



Figura 62. Terminación de la importación de datos en CouchDB
Fuente: Elaboración propia

## 3.4. Couchbase

### 3.4.1. Arquitectura.

#### *3.4.1.1. Replicación.*

En Couchbase la replicación es un mecanismo importante que permite aumentar la disponibilidad del sistema. Dentro de su arquitectura distribuida Couchbase utiliza el protocolo DCP (en inglés, *Database Change Protocol*) para la replicación de cubos, índices secundarios globales (GSI) y replicación cruzada de centros de datos (XDCR). DCP es un protocolo de replicación basado en la comunicación en memoria, la cual ayuda a reducir la latencia y aumentar considerablemente la disponibilidad evitando la pérdida de datos.

La replicación intra-clúster en Couchbase consiste en réplicas de datos que se colocan en otro servidor que se encuentra dentro del mismo clúster. El origen de los datos replicados se denomina cubo activo, y generalmente es donde se realizan las operaciones de lectura y escritura de documentos. El cubo de destino se llama cubo réplica. Los cubos de réplica reciben un flujo continuo de cambios desde el cubo activo a través del protocolo DCP. Aunque a los cubos de réplica no se accede normalmente, pueden responder a solicitudes de lectura.

En la Figura 63 se muestra la arquitectura de replicación que se implementará con la base de datos Couchbase.

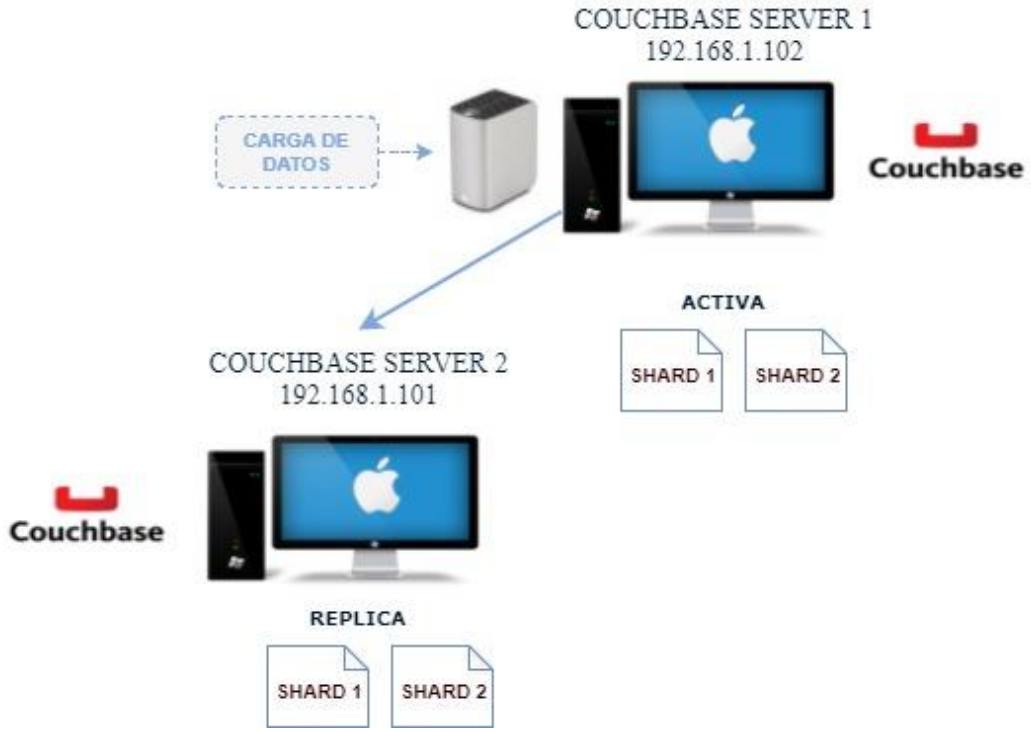



Figura 63. Topología de Couchbase para replicación intra-clúster
Fuente: Elaboración propia

### 3.4.2. Escalabilidad.

Couchbase es una base de datos NoSQL orientada a documentos con una arquitectura distribuida para el rendimiento, la escalabilidad y la disponibilidad. Aunque la mayoría de las bases de datos NoSQL son escalables, la funcionalidad de escalabilidad de Couchbase proporciona una manera muy fácil para la adición y eliminación de servidores sin que exista tiempo de inactividad pues no es necesario que el sistema sea puesto fuera de línea.

Además en Couchbase se ha introducido un nuevo enfoque de escala llamado escalamiento multidimensional[10], cuyo objetivo es aumentar sustancialmente el rendimiento. Con este tipo de escalado se puede aislar los servicios de datos, indexación y consulta en determinados servidores de un clúster Couchbase, en lugar de que cada servidor del clúster realice esas funciones; de manera que los recursos pueden ser optimizados para un único servicio.

### 3.4.3. Tolerancia a fallos.

Couchbase está construido sobre Erlang/OTP, un entorno probado para la construcción de sistemas distribuidos tolerantes a fallos. La redundancia que proporciona la replicación en Couchbase provee protección contra la pérdida de datos de cualquier servidor individual y ayuda a aumentar la disponibilidad de datos al permitir la recuperación de fallos de hardware y las interrupciones del servicio.

La replicación aumenta la disponibilidad del sistema, de tal manera que son las réplicas quienes responden a las solicitudes durante el corto período de tiempo en que una copia activa no está disponible. Cuando un servidor no responde por cualquier motivo, el orquestador que fue elegido para supervisar el clúster (uno de los servidores dentro del clúster) notifica a los nodos del clúster y promueve las réplicas correspondientes al estado activo. Si el orquestador falla o se pierde la comunicación con el grupo por cualquier motivo, los nodos restantes detectan el fallo cuando dejan de recibir sus notificaciones, por lo que se elige rápidamente un nuevo orquestador. Esto se realiza inmediatamente y es transparente para las operaciones del clúster.

### 3.4.4. Implementación y configuración de Couchbase.

#### *3.4.4.1. Replicación en Couchbase.*

Dentro de una clúster Couchbase se emplea replicación Peer-To-Peer, para mejor rendimiento y la utilización de recursos, así como la eliminación de posibles cuellos de botella

[10] Más información en http://ow.ly/F6mn30cjbHr

y puntos únicos de fallo (Couchbase, 2017). En el ANEXO G, se muestra el procedimiento para la implementación de un clúster Couchbase. En la Figura 64 se muestra la creación del cubo que va a replicarse.

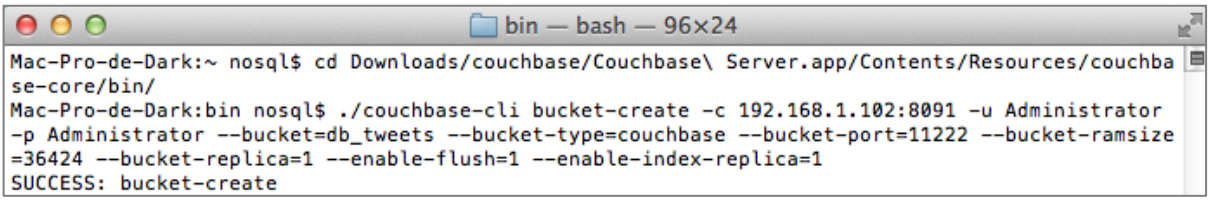



Figura 64. Creación de cubo en Couchbase
Fuente: Elaboración propia

Cuando se crea un cubo, automáticamente se crea copias de los datos activos, y se distribuyen esas réplicas a través de los servidores en el clúster Couchbase, asegurando que cada copia se encuentra en un servidor separado. Couchbase soporta de 1 a 3 réplicas de datos, si un servidor se cae, Couchbase recupera los datos mediante la activación de las réplicas que ya existen en otros servidores del clúster.

### 3.4.5. Importación de datos en Couchbase.

En Couchbase se utiliza la herramienta `cbtranfer`[11] para la importación de los datos, especificando ciertos parámetros.

- `/path/to/filename`: Especifica la dirección de un archivo .csv
- `-B`: Nombre del cubo de destino que recibe la transferencia de datos.
- `-u`: Es el nombre del usuario del servidor Couchbase
- `-p`: Es la contraseña del usuario del servidor Couchbase
- `-t`: Especifica el número de subprocesos simultáneos que realizan la importación de los datos

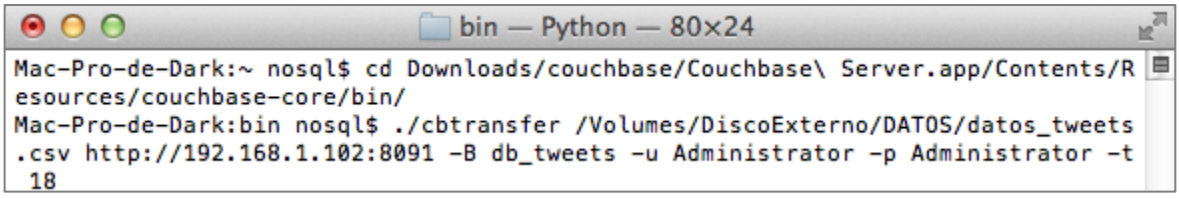



Figura 65. Comando para importar datos en CouchDB
Fuente: Elaboración propia

[11] Más información en: https://developer.couchbase.com/documentation/server/current/cli/cbtransfer-tool.html

### 3.5. Resultados previos

En esta sección se muestran los resultados obtenidos durante el proceso de importación de datos a cada una de las bases de datos. Las características que se han sido consideradas para analizar como resultados son el tiempo empleado por la base de datos para la carga total de los datos y la cantidad de espacio ocupado en el disco, cuando la información es equivalentes a 20 millones de datos.

Tabla 7. Resultados previos

| Base de datos | Tiempo de carga | Espacio de disco |
|---|---|---|
| *MongoDB* | 52 min | 36.49 GB |
| *CouchDB* | 106 min | 44.4 GB |
| *Couchbase* | 45 min | 50.82 GB |

Fuente: Elaboración propia

# CAPÍTULO IV

# DESARROLLO DE APLICACIÓN WEB Y PRUEBAS DE RENDIMIENTO DE LAS BASES DE DATOS NOSQL

En el presente capítulo se presentan las arquitecturas y las herramientas utilizadas en la aplicación web que se ha desarrollado para evaluar las bases de datos MongoDB, CouchDB y Couchbase. Y se presenta la definición de las pruebas que se llevan a cabo para medir el rendimiento de las bases de datos NoSQL en diferentes escenarios. La comparación a realizar entre las bases de datos sobre diferentes escenarios permitirá obtener conclusiones en cuanto a su capacidad de procesamiento de datos y cómo éstos gestores manejan grandes cantidades de datos.

Es importante mencionar que en el presente trabajo se considera el rendimiento, la escalabilidad y la tolerancia a fallos de las bases de datos como los factores más importantes.

## 4.1. Desarrollo de aplicación web

En esta sección se describe la arquitectura y las tecnologías utilizadas en el desarrollo de la aplicación web. En la construcción de la aplicación web se ha identificado dos componentes importantes que son el frontend y el backend, que utilizan JSON como formato de intercambio de datos. El frontend es el encargado de la recolección de los datos de las solicitudes que realiza el cliente y el backend se encarga del procesamiento de dichas solicitudes.

Del lado del servidor, el backend de la aplicación desarrollada se ha construido como una API REST[12], con el fin de asegurar que cualquier sistema o cliente, pueda consumir los datos almacenados en la base de datos NoSQL. Por el contrario, del lado del cliente en el frontend se usó HTML, JavaScript y AngularJS para la representación visual con la que interactúa el cliente.

### 4.1.1. Herramientas utilizadas en la aplicación.

A continuación se describen las tecnologías utilizadas en el desarrollo de la aplicación web.

#### *4.1.1.1. Backend.*

**NodeJS**

Es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Este motor proporciona a Node un entorno de ejecución del lado del servidor que compila y ejecuta código JavaScript a velocidades muy rápidas. Node es de código abierto, y se ejecuta en Mac OS X, Windows y Linux.

[12] REST (Representational State Transfer), es un estilo de arquitectura basado en un conjunto de principios que describen cómo se definen y se abordan los recursos en red.

**Express**

Express es una infraestructura de aplicaciones web NodeJs mínima y flexible, que proporciona un conjunto sólido de características para la creación de aplicaciones web y móviles. Express se ejecuta como un módulo dentro del entorno NodeJs

**Cradle**

Es un módulo popular para integrar CouchDB con aplicaciones NodeJs. Cradle incorpora escritura a través de la caché, lo que le da un nivel extra de velocidad, y hacer las actualizaciones de documentos y eliminación más fácil.

**Mongoose**

Es una herramienta de modelado de objetos MongoDB diseñada para trabajar en un entorno asíncrono y que envuelve el controlador nativo de NodeJs. Mongoose es open-source con la licencia MIT y es mantenido por MongoDB, Inc.

**SDK Couchbase**

El SDK de Couchbase/NodeJs permite a una aplicación conectarse a un clúster de Couchbase desde NodeJs. Es un módulo Node.Js nativo y usa la librería C de alto rendimiento de Couchbase para manejar la comunicación con el clúster a través de los protocolos binarios de Couchbase.

#### *4.1.1.2. Frontend.*

**Librerías JavaScript**

Existen varias librerías que facilitan la programación de un lenguaje y proveen un gran número de funcionalidades. La librería que usó para el desarrollo de la aplicación web está enfocada al lenguaje de programación JavaScript.

- **Jquery:** es una librería de JavaScript que simplifica la manipulación y recorrido de documentos HTML, el manejo de eventos, animación, y las interacciones Ajax para el desarrollo rápido de aplicaciones web.

**Bootstrap**

Es una colección de herramientas de código abierto que hace que el desarrollo web frontend sea rápido y fácil. Proveen plantillas HTML y CSS, formularios, botones, gráficos, e incluso el

manejo de JavaScript para componentes como: carrusel de imágenes, menús interactivos, transiciones, entre otros.

**AngularJs**

Es un framework de código abierto y escrito en Javascript, que trabaja del lado del cliente y permite construir aplicaciones web SPA (Single Page Application) dinámicas. AngularJs permite estructurar, organizar y escribir código de una manera más eficiente; además trabaja con el patrón MVC (Modelo, Vista, Controlador), lo que permite separar correctamente la lógica, el modelo de datos y la vista en una aplicación web y se enfoca en extender el vocabulario HTML.

### 4.1.2. Arquitectura de la aplicación.

El diseño de las arquitecturas que se ilustran a continuación muestran las dos partes bien diferenciadas el frontend y el backend de la aplicación web. Del lado del servidor (backend) se desarrolló una API REST con NodeJs, Express y las librerías mongoose, cradle y SDK Couchbase, para la comunicación con la base de datos MongoDB, CouchDB y Couchbase respectivamente. Del lado del cliente (frontend) se desarrolló una aplicación de una solo página con AngularJs. Las aplicaciones de una sola página (SPA) son aplicaciones que cargan los datos de forma asíncrona, y la página no se recarga en casi ningún momento pues los recursos necesarios se cargan de forma dinámica como lo requiera la página, y se van agregando como respuesta a las acciones del cliente.

En la Figura 66, se muestra la arquitectura de la aplicación web que se ha construido para consumir los datos de la base de datos MongoDB. El formato de intercambio de datos que utiliza dentro del backend y hacia afuera es el formato JSON, el mismo se ha convertido en el formato estándar para APIs web y es una de las tecnologías que más populares entre los desarrolladores. Dentro del backend de la aplicación, el framework web Express para NodeJs proporciona una plataforma de desarrollo clara y concisa para crear APIs web de alto rendimiento, y además ayuda a gestionar el enrutamiento de las solicitudes a las funciones correctas de la aplicación.

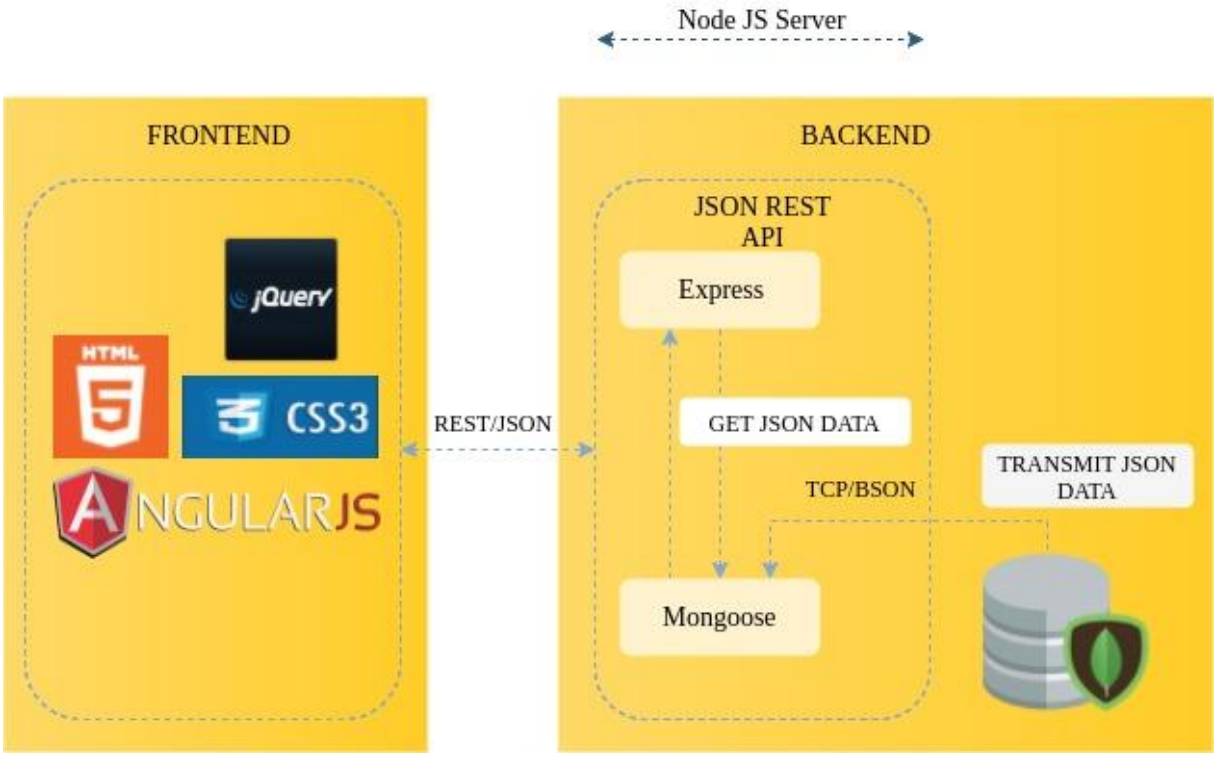



Figura 66. Arquitectura de aplicación web para acceder a la base de datos MongoDB
Fuente: Elaboración propia

La librería mongoose se utilizó para mantener la comunicación de datos entre el servicio REST y la base de datos MongoDB. Mongoose es una librería de modelado de objetos que proporciona un entorno de modelado de datos riguroso que permite definir esquemas de las colecciones almacenadas en la base de datos.

Con respecto a la base de datos CouchDB, la arquitectura de la aplicación web desarrollada para consumir los datos almacenados en CouchDB se muestra en la Figura 67. Al igual que MongoDB se ha utilizado AngularJs en el lado del frontend y Express para NodeJs del lado del backend. En este caso, se usa la librería cradle para la comunicación de datos del servicio REST y CouchDB. Cradle es una librería open-source de alto nivel que surgió de la unión de CouchDB y NodeJs que además de los dos aspectos que comparten en común (JavaScript y su comportamiento asincrónico), también permite operaciones de lectura, escritura, actualización y eliminación. Las operaciones de lectura se realizan mediante el uso del motor de vistas de MapReduce de CouchDB.

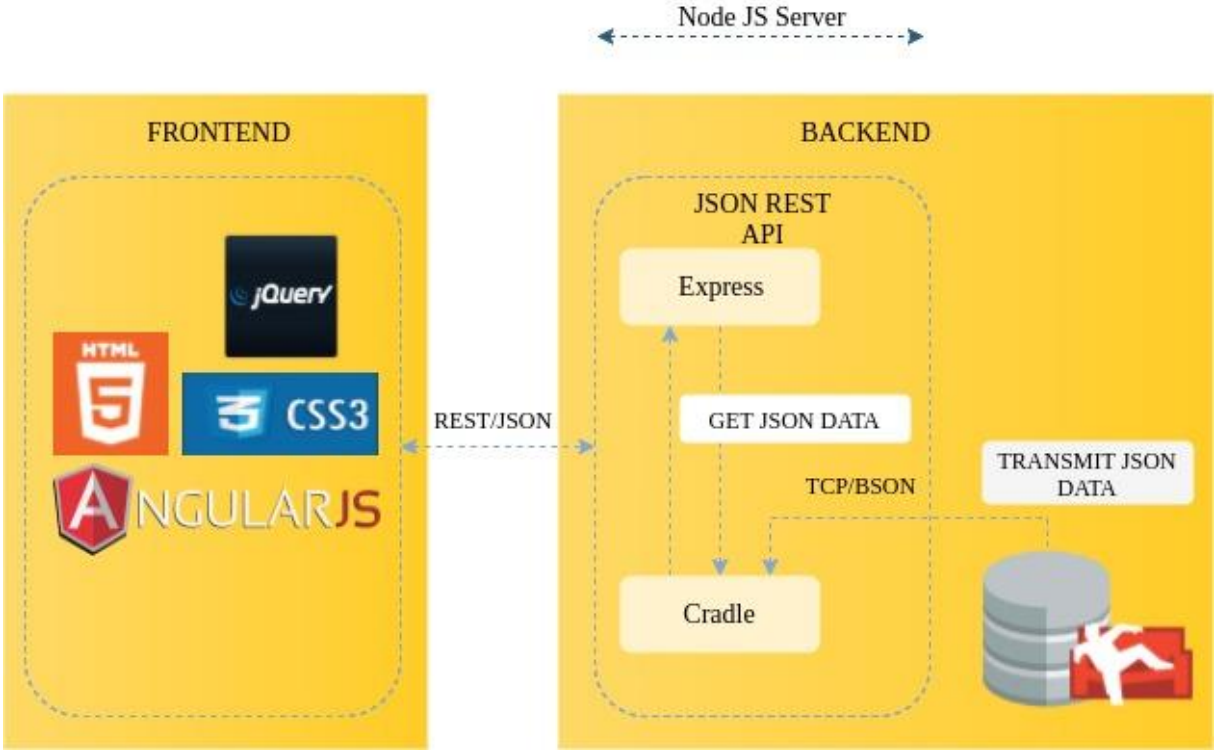



Figura 67. Arquitectura de aplicación web para acceder a la base de datos CouchDB
Fuente: Elaboración propia

Finalmente, en la Figura 68 se ilustra la arquitectura de la aplicación web desarrollada para consumir los datos almacenados en Couchbase. De manera semejante que en MongoDB y CouchDB, la aplicación web desarrollada para esta base de datos, usa las misma tecnologías de Express para NodeJs. Sin embargo en este caso la librería utilizada para comunicarse con la base de datos es el SDK Couchbase para NodeJs. El SDK de Couchbase es una librería de alto rendimiento que permite manejar la comunicación con el clúster de Couchbase, y realizar operaciones básicas de escritura, actualización y eliminación, así como operaciones de consulta que se realizan mediante el uso del motor de vistas de MapReduce de Couchbase.

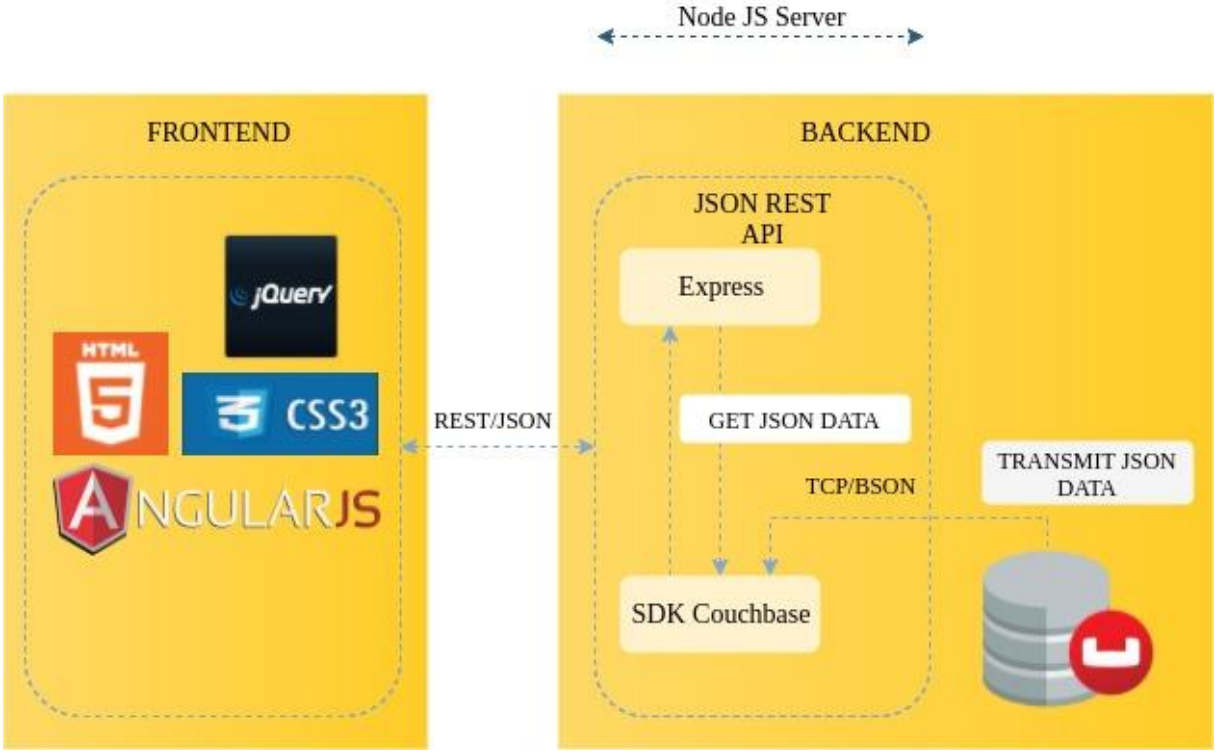



Figura 68. Arquitectura de aplicación web para acceder a la base de datos Couchbase
Fuente: Elaboración propia

### 4.1.3. Descripción de la aplicación.

La aplicación web que se ha desarrollado para cada base de datos permitirá realizar peticiones de búsqueda que retornarán una respuesta en formato JSON. Se utilizan las librerías Mongoose, Cradle y SDK Couchbase mencionadas en la sección 4.1.1.1, para obtener datos específicos de los 20 millones de tweets que se almacenan en cada base de datos. Para ello se construyó una API REST que proporciona un servicio para consumir los datos de las bases de datos mediante otra aplicación de una solo página desarrollada con AngularJs. Se desarrolló únicamente el módulo de búsquedas que se describe a continuación:

**Módulo de Búsquedas.** Es el módulo que permite realizar búsquedas de los tweets por un hashtag específico, por la fecha de creación del tweet y por el número de retweets que haya alcanzado el tweet.

### 4.1.4. Construcción de la API REST (BACKEND).

#### *4.1.4.1. MongoDB.*

Para el desarrollo de la API REST con la base de datos MongoDB se usó NodeJs, ExpressJs, y Mongoose para el mapeo de los datos. En primer lugar y para tener más clara la arquitectura definida (Figura 66) se define la jerarquía de carpetas que ha de seguir un proyecto en NodeJs. Como se observa en la Figura 69 la carpeta API_REST_MONGODB es la

contenedora de la funcionalidad y los recursos de la API. A continuación se describe con más detalle cada uno de sus componentes.

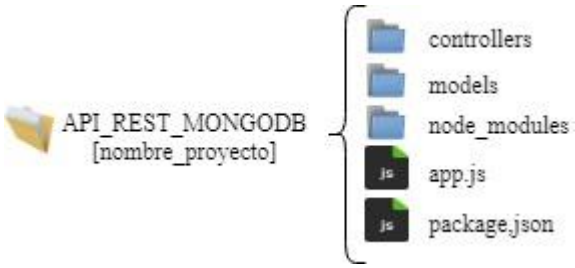



Figura 69. Jerarquía de carpetas de API REST MongoDB
Fuente: Elaboración propia

**Dependencias**

Para que la API funcione correctamente es necesario instalar todas las dependencias que se utilizarán, principalmente se requiere Express y Mongoose. Para ello en el directorio raíz de proyecto se añade el archivo package.json en donde se especifican las dependencias.

```
{
  "name": "mongodbjs",
  "version": "1.0.0",
  "dependencies": {
    "angular-datetime-input": "^5.1.2",
    "body-parser": "~1.13.2",
    "cors": "~2.8.3",
    "express": "~4.13.1",
    "method-override": "^2.1.2",
    "mongodb": "^2.2.28",
    "mongoose": "~4.4"
  }
}
```

Figura 70. Dependencias de API REST MongoDB
Fuente: Elaboración propia

Las dependencias se instalaron con NPM, el gestor de paquetes de Node. En la consola de comandos se ejecuta `npm install` para iniciar la instalación de las dependencias, las mismas se almacenarán en la carpeta node_modules que se ubica en el directorio raíz.

**Creación del modelo con Mongoose**

El modelo Mongoose hereda automáticamente los métodos crear, guardar, eliminar y buscar que permiten almacenar y recuperar instancias de modelo de MongoDB. Para la aplicación en construcción se utiliza Mongoose para crear un modelo de los tweets que únicamente se encargará de recuperar instancias de la base de datos MongoDB. Para ello, se crea la carpeta models y dentro de ella se crea el fichero models.js.

```
var mongoose = require('mongoose');
var Schema   = mongoose.Schema;

var clientSchema = new Schema({
  _id:    { type: String },
  id_str:    { type: String },
  text:    { type: String },
  retweet_count:    { type: Number },
  screen_name:    { type: String },
  tweet_date:    { type: Date },
  hashtags:    { type: String },
  urls:    { type: String },
  user_mentions:    { type: String }
});

module.exports = mongoose.model('tweets', clientSchema);
```

Figura 71. Modelo de objetos con Mongoose
Fuente: Elaboración propia

La Figura 71 muestra los atributos del tweet que se quieren recuperar como el id, el texto del tweet, el número de retweets, el nombre del usuario y la fecha en que se publicó, los hashtags asociados, entre otros.

**Conexión a la base de datos**

La conexión a una instancia de MongoDB con Mongoose es sencilla, requiriendo sólo la URL de recursos de la base de datos. Para ello en el fichero app.js se añaden las dependencias necesarias y se establece la conexión. La Figura 72 muestra la conexión a un conjunto de réplicas.

```
var express = require('express');
var bodyParser = require('body-parser');
var mongoose = require('mongoose');
var methodOverride = require("method-override");
var app = express();
var cors = require('cors');

// CONEXIÓN AL CONJUNTO DE REPLICAS DE LA BASE DE DATOS MONGODB
mongoose.connect('mongodb://192.168.1.101:27052,192.168.1.102:27051,192.168.1.103:27053/db_tweets', function(err, res) {
if(err) throw err;
 console.log('Conexión establecida');
});
```

Figura 72. Conexión a un conjunto de réplicas con Mongoose
Fuente: Elaboración propia

Y para establecer la conexión a un clúster fragmentado en MongoDB es necesario colocar la URI de la instancia mongos y establecer la opción mongos = true.

```
var express = require('express');
var bodyParser = require('body-parser');
var mongoose = require('mongoose');
var methodOverride = require("method-override");
var app = express();
var cors = require('cors');

// CONEXIÓN A CLUSTER FRAGMENTADO EN MONGODB
mongoose.connect('mongodb://192.168.1.101:27025,192.168.1.102:27026/db_tweets', {mongos: true}, function(err, res) {
if(err) throw err;
 console.log('Conexión establecida');
});
```

Figura 73. Conexión a un clúster fragmentado en MongoDB con Mongoose
Fuente: Elaboración propia

### Creación del controlador

Una vez que se ha configurado la conexión se requieren crear las rutas que definirán las llamadas a la API. Los controladores de las rutas de la API residen dentro la carpeta controllers en el fichero controllers.js. Utilizando exports se consigue modularidad para que pueda ser llamado desde el fichero principal de la aplicación. La Figura 74 muestra la función devuelve los tweets según un hashtag específico y dependiendo del número de registros que se solicite.

```
1   var mongoose = require('mongoose');
2   var Tweets  = mongoose.model('tweets');
3
4   exports.findAll = function(req, res) {...
12  };
13
14  exports.findById = function(req, res) {...
20  };
21
22  exports.findByHash = function(req, res) {
23      var tag = req.params.hashtag;
24      var lim = parseInt(req.params.lim);
25      Tweets.find(
26          { "hashtags" : { $regex: '.*'+tag+'$', $options: 'i' } },{
27              id:1,text:1,retweet_count:1,
28              screen_name:1,tweet_date:1,
29              hashtags:1,urls:1,
30              user_mentions:1
31          }).limit(lim).exec(function(err, tweets) {
32      if(err) return res.send(500, err.message);
33          console.log('GET /hashtags');
34          console.log("ENCONTRADO " +  tweets.length + " REGISTROS");
35          res.status(200).jsonp(tweets);
36      });
37  };
38
39  exports.findByRetweets = function(req, res) {...
48  };
49
50  exports.findByCreated = function(req, res) {...
63  };
```

Figura 74. Controlador de API REST MongoDB
Fuente: Elaboración propia

Luego se procede a unir las funciones del controlador a las peticiones que serán las llamadas desde el API, para ello en el fichero app.js se declara como muestra la Figura 75. El método express.Router se utiliza para crear manejadores de rutas montables y modulares.

```
23  // IMPORTAR MODELO Y CONTROLADORES
24  var models     = require('./models/models')(app, mongoose);
25  var Controller = require('./controllers/controller');
26
27  var router = express.Router();
28
29  // INDEX ROUTE
30  router.get('/', function(req, res) {
31    res.send("API REST MONGODB");
32  });
33
34  app.use(router);
35
36  // API ROUTES
37  var api = express.Router();
38
39  api.route('/alltweets/:numeroTweets')
40    .get(Controller.findAll)
41
42  api.route('/tweets/:id')
43    .get(Controller.findById)
44
45  api.route('/hashtags/:hashtag&:lim')
46    .get(Controller.findByHash)
47
48  api.route('/retweets/:num&:lim')
49    .get(Controller.findByRetweets)
50
51  api.route('/created/:minDate&:maxDate&:lim')
52    .get(Controller.findByCreated)
53
54  app.use('/api', api);
55
56  app.listen(3000, function() {
57    console.log("Aplicación corriendo en: http://192.168.1.101:3000");
58  });
```

Figura 75. Llamadas GET de la API REST MongoDB
Fuente: Elaboración propia

Finalmente, para probar el funcionamiento de la API se debe tener MongoDB ejecutándose en cualquiera de las arquitecturas implementadas y el servidor NodeJs de la aplicación corriendo. Llegados a este punto en un navegador se introducen los datos necesarios para hacer una llamada GET que recupere los registros de la base de datos en formato JSON.

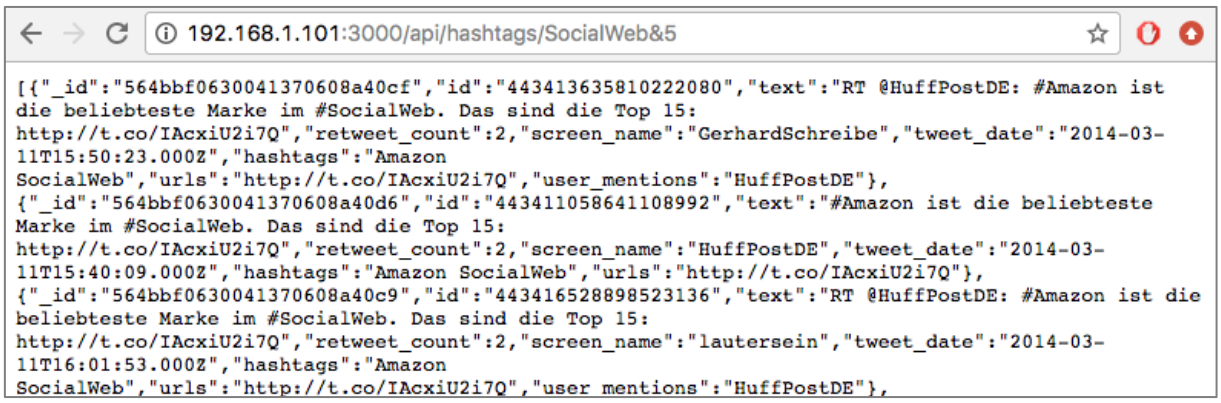



Figura 76. Llamada HTTP – GET a la API REST MongoDB
Fuente: Elaboración propia

#### *4.1.4.2. CouchDB.*

Del mismo modo que la API REST desarrollada para MongoDB, en CouchDB se utiliza las mismas tecnologías NodeJs, Express y para conexión a la base de datos se usa la librería Cradle. Todas las dependencias necesarias se instalaron ejecutando `npm install` en la consola de comandos. La Figura 77 muestra que la carpeta API_REST_COUCHDB contiene la funcionalidad y los recursos de la API. A continuación se describe con más detalle cada uno de sus componentes.


Figura 77. Jerarquía de carpetas de API REST CouchDB
Fuente: Elaboración propia

**Dependencias**

Las dependencias requeridas se añaden en el archivo package.json que se ubica en el directorio raíz del proyecto. La librería Cradle se utiliza para establecer la comunicación con CouchDB. Todas las dependencias se almacenan en la carpeta node_modules.

```
{
  "name": "couchdbjs",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "~1.17.2",
    "consolidate": "~0.9.1",
    "express": "~4.15.3",
    "cors": "^2.8.3",
    "validator": "~1.1.3",
    "method-override":"2.3.9",
    "uuid": "^3.0.1",
    "cradle": "0.6.7"
  }
}
```

Figura 78. Dependencias de API REST CouchDB
Fuente: Elaboración propia

**Creación de vista**

Para extraer datos de CouchDB y presentarlos en un orden específico es necesario la creación de vistas (views), que se escriben en JavaScript y retornan un conjunto clave-valor. La Figura 79 muestra la vista creada para obtener datos de tweets según el número de retweets que ha

alcanzado. La función `emit(key, value)` contiene siempre dos argumentos: el primero es la clave y el segundo es el valor.

```
function (doc) {
  emit(doc.retweet_count, {
    _id: doc._id,
    _rev: doc._rev,
    id_str: doc.id_str,
    text: doc.text,
    retweet_count: doc.retweet_count,
    tweet_date: doc.tweet_date,
    screen_name: doc.screen_name,
    urls: doc.urls,
    iso_language_code: doc.iso_language_code
  });
}
```

Figura 79. Vista obtener tweets por número de retweets
Fuente: Elaboración propia

En la vista que se ha creado para obtener tweets por número de registros, la clave corresponde a `doc.retweet_count` que representa el número de retweets y el valor indica todos los atributos que deben retornarse cuando se haga una llamado a esa vista.

**Creación del modelo**

Una vez que se han creado las vistas, estas se utilizan para encontrar documentos por cualquier valor o estructura que resida en ellos. Para la aplicación en construcción se creó la carpeta models y dentro de ella se crea el fichero models.js. En la Figura 80 con la función `TweetsDAO(db)` se puede tener una referencia a base de datos a través del constructor `db`, y el cual utiliza la función `view` para acceder a la vista a través de tres argumentos: el primer argumento es el nombre de la vista o documento de diseño, el segundo argumento es la combinación de clave y valor por la que se accede al documento en donde además se incluye el parámetro limit para establecer el límite de registros que deben devolverse y el tercer argumento es el manejador de la función para la respuesta.

```
1   function TweetsDAO(db) {
2       "use strict";
3       if (false === (this instanceof TweetsDAO)) {...
6       }
7
8       this.getAll = function(numw, num, callback) {...
20      }
21
22      this.getByHashtag = function(tag, lim, callback) {...
36      }
37
38      this.getRetweets = function(num, lim, callback) {
39          "use strict";
40          var numrw = num;
41          var numlim = lim;
42
43          db.view('docs/retweets',{ key: numrw, limit: numlim },
44              function (err, doc) { "use strict";
45
46              if (err) return callback(err, null);
47              console.log("ENCONTRADO " + doc.length + " REGISTROS");
48
49              callback(null, doc);
50
51          });
52      }
53
54      this.getByCreated = function(min, max, lim, callback) {...
67
68      }
69  }
70
71  module.exports.TweetsDAO = TweetsDAO;
```

Figura 80. Fichero models.js API REST CouchDB
Fuente: Elaboración propia

**Conexión a la base de datos**

La conexión a una instancia de CouchDB y la inicialización de las dependencias se hace en el fichero app.js. La Figura 81 muestra la inicialización de las dependencias necesarias para la construcción de la API.

```
1  var express = require('express'), app = express();
2  var cons = require('consolidate');
3  var routes = require('./routes'); // Routes de la aplicación aplicación
4
5  var cradle = require('cradle'); // Driver para controlador node.js CouchDB
6  var methodOverride = require("method-override");
7  var bodyParser = require('body-parser');
8  var cors = require('cors');
```

Figura 81. Inicialización de dependencias
Fuente: Elaboración propia

La conexión a una instancia de CouchDB usando la librería Cradle es sencilla, sólo se requiere la URL de recursos de la base de datos. La Figura 82 muestra la conexión al servidor maestro que está ejecutando los procesos de replicación. Para establecer la conexión a una base de

datos CouchDB fragmentada se puede establecer la URL de cualquiera de los servidores que se añadieron al servidor principal.

```
10  var databaseUrl = "replica-tweets";
11
12  var connection = new(cradle.Connection)('http://192.168.1.102', 5984, {
13        auth: { username: 'Admin', password: 'Admin' }
14    });
15
16  var db = connection.database(databaseUrl);
```

Figura 82. Conexión al servidor principal CouchDB usando Cradle
Fuente: Elaboración propia

Los middleware son capaces de manejar solicitudes y objetos de respuesta HTTP, realizar alguna operación por la petición, enviar la respuesta al usuario y pasar los resultados al siguiente middleware. En la Figura 83 se observa que se usa el método `app.use()` para cargar los middlewares en la aplicación. La mayoría de las funcionalidades que proporciona Express se implementa con sus middlewares incorporados. El middleware enrutador `routes(app,db)` es el responsable de encaminar las solicitudes HTTP a las funciones apropiadas del manejador de datos.

```
20      app.use(bodyParser.urlencoded({ extended: false }));
21      app.use(bodyParser.json());
22      app.use(methodOverride());
23      app.use(cors());
24
25      routes(app, db);
26
27      app.listen(8082);
28      console.log('Aplicación corriendo en: http://192.168.1.103:8082');
```

Figura 83. Carga de Middlewares
Fuente: Elaboración propia

**Creación del controlador**

El destino de las solicitudes HTTP URIs se definen a través de las rutas de la aplicación, las cuales deciden a que función llamar mediante el análisis de los datos en el objeto de la solicitud. La Figura 84 muestra el archivo index.js localizado dentro de la carpeta routes que contiene los manejadores de rutas.

```
var  ContentHandler = require('../controllers/controller.js')
  , ErrorHandler = require('./error').errorHandler;

module.exports = exports = function(app, db) {

    var handler = new ContentHandler(db);

    app.get('/alltweets/:numeroTweets', handler.findAll);

    app.get('/hashtags/:hashtag&:lim', handler.findByHash);

    app.get('/retweets/:num&:lim', handler.findByRetweets);

    app.get('/created/:minDate&:maxDate&:lim', handler.findByCreated);

    app.use(ErrorHandler);
}
```

Figura 84. Llamadas GET de la API REST CouchDB
Fuente: Elaboración propia

Los controladores de las rutas de la API residen dentro la carpeta controllers en el fichero controllers.js. La Figura 85 muestra la función que hace la llamada a una función del models.js para que devuelva los registros de tweets según el número de retweets y dependiendo del número de registros solicitados.

```
var TweetsDAO = require('../models/tweets').TweetsDAO
  , sanitize = require('validator').sanitize;

function ContentHandler (db) {
    "use strict";

    var Tweets = new TweetsDAO(db);

    this.findAll = function(req, res, next) {...
    }

    this.findByHash = function(req, res, next) {...
    }

    this.findByRetweets = function(req, res, next) {
        "use strict";

        var numrw = req.params.num;
        var numlim = req.params.lim;
        Tweets.getRetweets(numrw, numlim, function(err, results) {
            "use strict";

            if(err) return res.send(500, err.message);
                console.log('GET /retweets')
                res.status(200).jsonp(results);
            });
    }

    this.findByCreated = function(req, res, next) {...
    }
}

module.exports = ContentHandler;
```

Figura 85. Controlador de API REST CouchDB
Fuente: Elaboración propia

#### *4.1.4.3. Couchbase.*

La API REST desarrollada para Couchbase utiliza las NodeJs, Express y Cradle para la comunicación y mapeo de datos con la base de datos. La carpeta API_REST_COUCHBASE contiene toda la funcionalidad y los recursos de la API. A continuación se describe con más detalle cada uno de sus componentes.

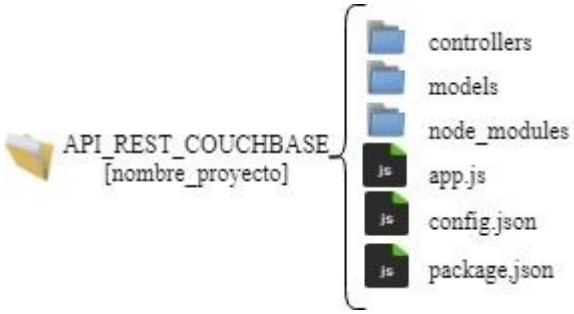



Figura 86. Jerarquía de carpetas de API REST Couchbase
Fuente: Elaboración propia

**Dependencias**

Las dependencias requeridas se añaden en el archivo package.json que se ubica en el directorio raíz del proyecto. Todas las dependencias necesarias se instalaron ejecutando `npm install` en la consola de comandos. Se usa la librería SDK Couchabse para establecer la comunicación con la base de datos Couchbase. Todas las dependencias se almacenan en la carpeta node_modules.

```
{
  "name": "couchbasejs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.17.2",
    "couchbase": "^2.3.3",
    "express": "^4.15.3",
    "uuid": "^3.0.1",
    "cors": "^2.8.3"
  }
}
```

Figura 87. Dependencias de API REST Couchbase
Fuente: Elaboración propia

La dependencia de `couchbase` se usa para comunicarse desde el servidor de aplicaciones al servidor Couchbase. La dependencia `express` permite usar el Framework ExpressJS y la dependencia `uuid` permite generar identificadores únicos.

### Creación de vista

Para extraer datos de Couchbase, al igual que en CouchDB es necesario la creación de vistas (views), que se escriben en JavaScript y devuelven un conjunto clave-valor. La Figura 88 muestra la vista creada para obtener datos de tweets según su fecha de creación. La función `emit(key, value)` contiene siempre dos argumentos: el primero es la clave que está representado por un array de la fecha de creación del tweet, y el segundo es el valor que son los atributos que deben retornarse cuando se haga una llamado a esa vista.

```
function(doc) {
  var txt = doc.tweet_date;
  txt.replace(/[/\/!.,;'']+/g, "");
  var raw_words = txt.split(/[T]+/i);
  var word_set = [];
  var words = {};
  for (var i in raw_words) {
    var word = raw_words[i];
    if (word == "") continue;
    if (!words[word]) { words[word] = 1; }
    else { words[word]++; }
  }
  for (var word in words) {
    word_set.push(word);
  }
  emit(raw_words,{
    _id: doc._id,
    _rev: doc._rev,
    id_str: doc.id_str,
    text: doc.text,
    retweet_count: doc.retweet_count,
    tweet_date: doc.tweet_date,
    screen_name: doc.screen_name,
    urls: doc.urls,
    iso_language_code: doc.iso_language_code
  });
}
```

Figura 88. Vista obtener tweets por la fecha de creación
Fuente: Elaboración propia

### Configuración del servidor

Toda la configuración de servidor estará en el fichero app.js que se encuentra en la raíz del proyecto. Sin embargo, se creó el fichero config.json para definir dos variables constantes: los servidores y el cubo Couchbase que se utilizarán.

```
{
    "couchbase": {
        "server": "192.168.1.102,192.168.1.101",
        "bucket": "db_tweets"
    }
}
```

Figura 89. Fichero config.json - API REST Couchbase
Fuente: Elaboración propia

El fichero app.js inicializa las dependencias descargadas con NPM y carga los middleware para manejar las solicitudes y objetos de respuesta HTTP.

```
var express = require("express");
var bodyParser = require("body-parser");
var methodOverride = require("method-override");
var couchbase = require("couchbase");
var path = require("path");
var config = require("./config");
var cors = require('cors');

var app = express();

module.exports.bucket = (new couchbase.Cluster(config.couchbase.server)).openBucket(
    config.couchbase.bucket).on('error', function(err) {
  console.log('CONNECT ERROR:', err);
});

app.use(express.static(path.join(__dirname, "public")));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());
app.use(cors());

var routes = require("./controllers/controller.js")(app);

var server = app.listen(3000, function () {
    console.log('Aplicación corriendo en: http://localhost:%s', server.address().port);
```

Figura 90. Fichero app.js - API REST Couchbase
Fuente: Elaboración propia

La Figura 91 muestra cómo se hace la conexión al clúster Couchbase y un cubo específico, ambos definidos en el archivo config.json.

```
module.exports.bucket = (new couchbase.Cluster(config.couchbase.server)).openBucket(
    config.couchbase.bucket).on('error', function(err) {
  console.log('CONNECT ERROR:', err);
});
```

Figura 91. Conexión a servidor Couchbase usando SDK Couchbase
Fuente: Elaboración propia

### Creación del modelo

Una vez que se han creado las vistas y antes de configurar las rutas de la aplicación, se determinan las funciones que se utilizarán para comunicarse con la base datos Couchbase. Estas funciones se describen el en fichero models.js que se ubica dentro de la carpeta models.js, y serán responsables de recuperar datos de Couchbase. La Figura 92 muestra la función que llama a la vista "`dev_tweets/created`" para obtener los registros de los tweets según la fecha de creación y dependiendo del número de registros solicitados.

```
var uuid = require("uuid");
var db = require("../app").bucket;
var config = require("../config");
var N1qlQuery = require('couchbase').N1qlQuery;
var ViewQuery = require('couchbase').ViewQuery;

function TweetsModel() { };

module.exports = TweetsModel;

TweetsModel.getById = function(id, callback) {...
};

TweetsModel.getAll = function(num, callback) {...
};

TweetsModel.getByTag = function(tag, lim, callback) {...
};

TweetsModel.getByRetweets = function(numrw, lim, callback) {...
};

TweetsModel.getByCreated = function(date_1, date_2, lim, callback) {
    var startKey = [date_1];
    var endKey = [date_2,{}];
    var query = ViewQuery.from("dev_tweets","created").range(startKey,endKey).limit(lim).skip(2);
    var res = db.query(query, function(error, result) {
        if(error) {
            return callback(error, null);
        }
        callback(null, result);

    });
    res.on('end', function(meta) {
        console.log("GET /created");
    });
};
```

Figura 92. Fichero models.js API REST Couchbase
Fuente: Elaboración propia

### Creación del controlador

Los manejadores de rutas de la API residen dentro la carpeta controllers en el fichero controllers.js (ver Figura 93). El destino de las solicitudes HTTP URIs se definen a través de las rutas de la aplicación, las cuales deciden a que función llamar mediante el análisis de los datos en el objeto de la solicitud.

```
    app.get("/api/created/:minDate&:maxDate&:lim", function(req, res) {
        TweetsModel.getByCreated(req.params.minDate,req.params.maxDate,req.params.lim,
            function(error, result, body) {
            if(error) {
                return res.status(400).send(error);
            }
            if(error) return res.send(500, error.message);
                res.status(200).jsonp(result);
                console.log("ENCONTRADOS " + result.length + " REGISTROS");
        });
    });
};

module.exports = appRouter;
```

Figura 93. Controlador de API REST Couchbase
Fuente: Elaboración propia

### 4.1.5. Construcción del FRONTEND.

El frontend es el responsable de solicitar los datos al backend desarrollado para las bases de datos MongoDB, CouchDB y Couchbase. En primer lugar y para tener más clara la estructura de la aplicación web desarrollada con AngularJs se define la jerarquía de carpetas. A continuación se describe con más detalle cada uno de sus componentes.


Figura 94. Jerarquía de carpetas de aplicación web
Fuente: Elaboración propia

Para empezar aplicación web que se ha desarrollado usa AngularJS, pero también está utilizando librerías JavaScript y Twitter Bootstrap. Todas estas bibliotecas o librerías se han descargado y colocado dentro de la carpeta assets en el directorio raíz.

**Creación de la página de la aplicación**

La aplicación tiene la finalidad de consumir un servicio web REST y en primer lugar se ha creado el archivo que contendrá la plantilla sobre la que AngularJs mostrará el contenido que se consumirá desde bases de datos NoSQL y que se presentará en formato JSON.

```
<!doctype html>
<html lang="en" ng-app='app-services'>

<head>
    <title>DATOS</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <!-- CSS -->
    <link rel="stylesheet" href="assets/css/bootstrap.min.css">
    <link rel="stylesheet" href="assets/css/vendor/icon-sets.css">
    <link rel="stylesheet" href="assets/css/main.min.css">
    <link rel="stylesheet" href="assets/css/demo.css">
    <!-- GOOGLE FONTS -->
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:300,400,600,700" rel="stylesheet">
    <!-- Angular Js -->
    <script type="text/javascript"
        src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.js"></script>
    <!-- Jquery -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.0.0/jquery.min.js"></script>
    <!--Controllers-->
    <script type="text/javascript" src="controllers/controller.js"></script>
    <!--Services-->
    <script type="text/javascript" src="services/services.js"></script>
    <script type="text/javascript">...
    </script>
</head>

<body ng-app="app-services" ng-controller="controller">...
</body>

</html>
```

Figura 95. Fichero index.html básico
Fuente: Elaboración propia

La Figura 102 muestra las librerías que se añadieron para crear la página HTML y además cargará el controlador en el navegador web del usuario. La aplicación está orientada a la búsqueda y recuperación de datos, es por ello que se ha implementado un buscador con diferentes parámetros para realizar las solicitudes HTTP. El diseño que se muestra en la Figura 96 permite el ingreso de dos parámetros: el hashtag que se desea buscar en algún tweet y el número de registros que debe devolver como resultados.


Figura 96. Diseño de buscador
Fuente: Elaboración propia

La Figura 97 muestra en cambio el código fuente del formulario del buscador, como se observa se pueda realizar búsquedas por hashtags, por número de retweets y por fecha de creación.

```
<form class="navbar-form navbar-left hidden-xs">
    <div class="input-group" ng-show="all" >
        <input type="text" ng-model="a.numw" class="form-control" placeholder="NUMERO DE TWEETS">
        <span class="input-group-btn">
            <button type="submit" class="btn btn-primary" ng-click="getAllTweets()">BUSCAR</button>
        </span>
    </div>
    <div class="input-group" ng-show="hash" >
        <input type="text" ng-model="a.hash" class="form-control" placeholder="# HASHTAG">
        <input type="number" ng-model="a.lim" class="form-control" placeholder="LIMITE DE REGISTROS">
        <span class="input-group-btn">
            <button type="submit" class="btn btn-primary" ng-click="getAllHashtags()">BUSCAR</button>
        </span>
    </div>
    <div class="input-group" ng-show="retweet" >
        <input type="text" ng-model="a.numre" class="form-control" placeholder="# DE RETWEETS">
        <input type="number" ng-model="a.lim" class="form-control" placeholder="LIMITE DE REGISTROS">
        <span class="input-group-btn">
            <button type="submit" class="btn btn-primary" ng-click="getAllRetweets()">BUSCAR</button>
        </span>
    </div>
    <div class="input-group" ng-show="created" >
        <input type="date" id="initialdate" ng-model="a.initialdate" class="form-control">
        <input type="date" id="lastdate" ng-model="a.lastdate" class="form-control">
        <input type="number" ng-model="a.lim" class="form-control" placeholder="LIMITE DE REGISTROS">
        <span class="input-group-btn">
            <button type="submit" class="btn btn-primary" ng-click="getAllCreated()">BUSCAR</button>
        </span>
    </div>
</form>
```

Figura 97. Código fuente de buscador
Fuente: Elaboración propia

### Controlador

En el controlador se encuentra la lógica que define el servicio REST dentro de la aplicación, para ello se ha creado el módulo "app-services". El controlador AngularJS crea el objeto `$scope` de forma automática y lo embebe dentro de la ejecución de aplicación. Este objeto permite manipular y acceder a los valores de los modelos y también sirve como una interfaz entre el HTML (vista) y el JavaScript (controlador).

```
angular.module('app-services', ['services']);

angular.module('app-services').controller('controller', ['$scope','request',controller]);

function controller($scope, request) {
    $scope.posts={};
    $scope.mensaje="";

    $scope.getAllTweets = function(){...
    };

    $scope.getAllHashtags = function(){...
    };

    $scope.getAllRetweets = function(){
        var start = new Date();
        request.searchNumRetweet($scope.a.numre,$scope.a.lim).then(function (data){
            console.log(data.data);
            $scope.posts=data.data;
            $scope.posts.exist=1;
            var time = new Date() - start;
            $scope.t = time;
        });
    };

    $scope.getAllCreated = function(){
        var start = new Date();
        var initial = new Date ($scope.a.initialdate);
        var last = new Date ($scope.a.lastdate);
        $scope.initialdate = initial.getFullYear()+"-"+(initial.getMonth()+1)+"-"+initial.getDate();
        $scope.lastdate =  last.getFullYear()+"-"+(last.getMonth()+1)+"-"+last.getDate();
        request.searchCreated($scope.initialdate,$scope.lastdate,$scope.a.lim).then(function (data){
            console.log(data.data);
            $scope.posts=data.data;
            $scope.posts.exist=1;
            var time = new Date() - start;
            $scope.t = time;
        });
    };

}
```

Figura 98. Controlador de la aplicación
Fuente: Elaboración propia

### Servicios

El servicio `$http` es uno de los servicios más utilizados en las aplicaciones de AngularJS. El servicio hace una solicitud al servidor y permite que su aplicación maneje la respuesta. En la Figura 99 en la variable `path` se establece la dirección del servidor de aplicaciones en donde se encuentra corriendo la API REST. Además se observa que los servicios definidos permiten hacer solicitudes HTTP para recuperar una lista limitada de tweets, búsqueda de tweets por hashtag, por número de retweets y por fecha de creación del tweet.

```
angular.module('services', [])
    .factory('request', function($http) {
        var path = "http://192.168.1.101:3000/api/";//PATH DE LA API

        return {
            lista : function(numw){
                global = $http.get(path+'alltweets/'+numw);
                return global;
            },
            searchHashtag : function(hash, lim){
                global = $http.get(path+'hashtags/'+hash+"&"+lim);
                return global;
            },
            searchNumRetweet : function(numre, lim){
                global = $http.get(path+'retweets/'+numre+"&"+lim);
                return global;
            },
            searchCreated : function(date_1,date_2, lim){
                global = $http.get(path+'created/'+date_1+"&"+date_2+"&"+lim);
                return global;
            }
        }
    });
```

Figura 99. Servicios $http
Fuente: Elaboración propia

Finalmente para probar la aplicación es necesario que la API REST de cualquiera de las bases de datos este corriendo en el servidor de aplicaciones y que a su vez dicha base de datos este ejecutándose en cualquiera de las arquitecturas implementadas. A continuación se presenta un ejemplo del resultado que se obtiene al realizar una consulta mediante la aplicación frontend desarrollada.

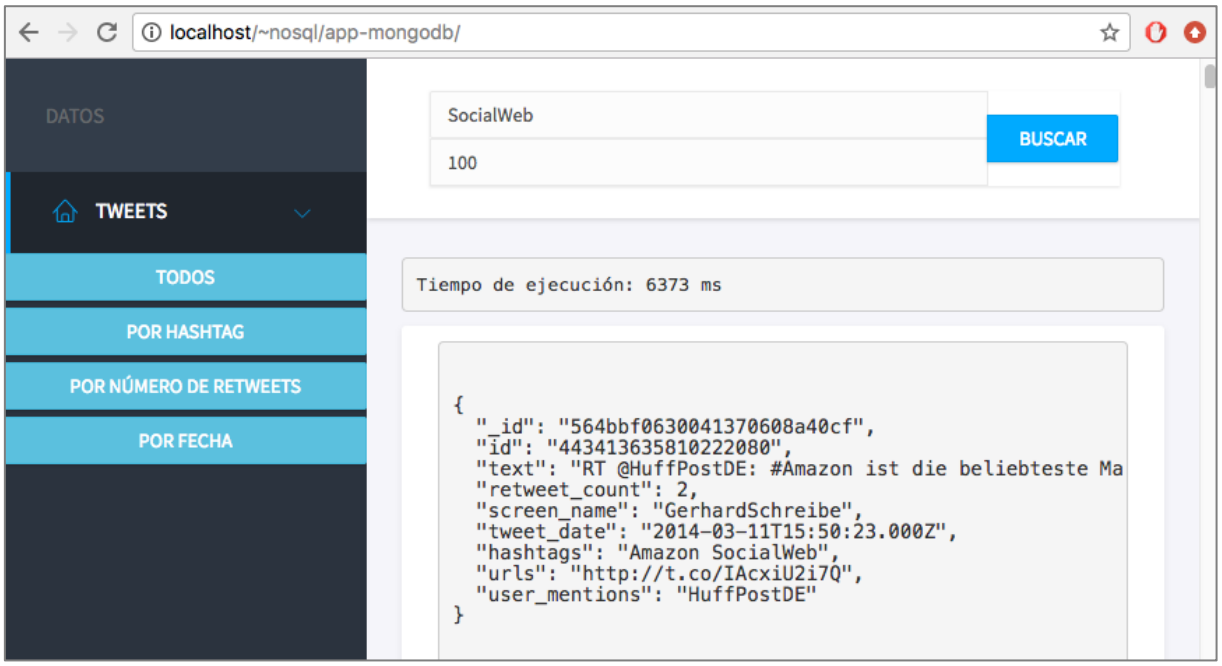



Figura 100. Frontend de la aplicación
Fuente: Elaboración propia

## 4.2. Pruebas

Las pruebas de rendimiento que se llevarán a cabo se enfocan principalmente en operaciones de consulta, que intentan simular un entorno informático donde un usuario o un grupo de usuarios ejecutan consultas a un sistema de base de datos NoSQL escalable y tolerante a fallos. La realización de estas pruebas servirá para determinar qué tan eficaz es la base de datos al momento de realizar consultas, y paralelamente validar y verificar la escalabilidad, tiempo de respuesta y uso de recursos de la base de datos en el ambiente distribuido.

Como parte de las pruebas de rendimiento y alta disponibilidad la aplicación web para cada una de las base de datos NoSQL servirán para demostrar que, ante la caída de un servidor de base de datos, la aplicación sigue operando de manera transparente para los usuarios. Todas las pruebas se realizaron bajo las mismas condiciones de hardware, tiempo, y disponibilidad de recursos, con el fin de no proporcionar ninguna clase de ventaja o desventaja a cualquiera de las bases de datos que se evaluarán.

### 4.2.1. Herramienta de simulación.

Se utilizó la herramienta de simulación Siege[13] para evaluar el entorno distribuido de la base de datos sobre el escenario en donde varios usuarios concurrentes realizan peticiones. Siege es una herramienta benchmark[14] de código abierto y con gran utilidad indicar el nivel de concurrencia en las peticiones a través de la simulación de varios usuarios simultáneos. Siege envía un número de peticiones al servidor web, según los parámetros que se indiquen se muestran las métricas sobre su comportamiento. A continuación se muestran las opciones de configuración más comunes:

```
siege [-b] [-c NUM] [-d NUM] [-f FILE] [-g] [-i] [-r NUM] [-t
NUM] [-v] [protocol://]hostname[:port][/path/file]
```

- `-c [NUM]`: Establece el número de usuarios concurrentes
- `-r [NUM]`: Especifica el número total de repeticiones de cada ráfaga de usuarios concurrentes
- `-t [NUM]`: Establece el tiempo límite para el cual se ejecuta la prueba (-t10s)
- `-d [NUM]`: Inserta un retardo comprendido entre 0 y NUM antes de lanzar la siguiente ráfaga de peticiones
- `-b`: Lanza la siguiente ráfaga de peticiones sin demora
- `-f [FILE]`: Ejecuta las peticiones con un archivo que contiene la lista de URL

[13] Más información en: https://github.com/JoeDog/siege
[14] Es un punto de referencia utilizado para medir el rendimiento en este caso de los gestores de bases de datos en el mercado

- `-i`: Con la opción `-f [FILE]`, selecciona aleatoriamente la URL que se ejecturá
- `-g`: Despliega los encabezados HTTP
- `-l`: Genera un archivo log de registro
- `[http[s]://]hostname[:port]/path`: Es la URL que se evaluará

### 4.2.2. Especificaciones técnicas.

Las pruebas se realizarán sobre varios escenarios con las siguientes especificaciones técnicas de hardware y software para la simulación de las consultas a las bases de datos por parte del usuario.

- Sistema operativo Ubuntu 16.04
- Procesador Intel® Core™ i7
- Memoria RAM de 6 GB
- Disco duro con capacidad de 500 GB

A continuación se presenta en la Tabla 8, los aspectos y características más importantes de las base de datos NoSQL sometidas a las pruebas de rendimiento y disponibilidad.

Tabla 8. Especificaciones técnicas

| Base de datos | MongoDB | CouchDB | Couchbase |
|---|---|---|---|
| **Versión** | 3.4.4 | 2.0.0 | 4.5.0 |
| **Tipo** | Documental | Documental | Documental |
| **Cantidad de nodos** | 3 | 3 | 2 |
| **Sistema Operativo** | Mac OS X (2), Linux (1) | Mac OS X (2), Linux (1) | Mac OS X (2) |
| **RAM** | Mac OS X (64 GB de RAM)<br>Linux – Ubuntu 16.04 (4 GB de RAM) | | |

Fuente: Elaboración propia

### 4.2.3. Definición de escenarios de prueba.

Considerando como base las especificaciones antes descritas se plantean los siguientes escenarios de prueba, para cada una de las bases de datos NoSQL. Se definió el tiempo de respuesta y consumo de recursos, como las variables de análisis para determinar el rendimiento de las bases de datos en los escenarios que se definen a continuación.

Tabla 9. Escenarios de prueba

| Escenario | Descripción | |
|---|---|---|
| **Escenario 1: Consulta general** | *Operación* | 1 usuario recuperando 100, 1000, 10000 y 100000 registros |
| | *Objetivo* | Obtener el tiempo total y tiempo promedio de la consulta |
| | *Descripción* | Se realizará la consulta especificando el número de registros que deben ser devueltos |
| | *Variables* | Tiempo de respuesta y consumo de recursos |
| **Escenario 2: Consulta masiva** | *Operación* | Simulación de 10.000 usuarios realizando consultas a la base de datos durante 10 minutos |
| | *Objetivo* | Obtener el tiempo total y tiempo promedio de consulta de las diferentes consultas a las bases de datos NoSQL |
| | *Descripción* | Se realizará tres tipos de consultas con diferentes grados de complejidad, para obtener resultados más confiables |
| | *Variables* | Tiempo de respuesta y consumo de recursos |
| **Escenario 3: Caída de uno de los servidores** | *Operación* | Dar de baja uno de los servidores |
| | *Objetivo* | Probar el comportamiento de las bases de datos NoSQL ante la inesperada caída de uno de los servidores |
| | *Descripción* | Analizar el comportamiento de la base de datos ante un fallo en uno de los servidores del sistema distribuido |
| | *Variables* | Comportamiento de la base de datos |

Fuente: Elaboración propia

A continuación se describirán brevemente las variables consideradas en el análisis de rendimiento de las bases de datos CouchDB, MongoDB y Couchbase.

- Tiempo de respuesta: hace referencia al tiempo que la base de datos tarda en realizar el total de consultas solicitadas.
- Consumo de recursos: es el porcentaje de uso de CPU y memoria RAM utilizado durante las consultas solicitadas a la base de datos.

### 4.2.4. Pruebas de rendimiento.

Como se describió en las secciones previas, las pruebas realizadas a las bases de datos consisten en consultas generales y consultas masivas con 1000 usuarios concurrentes. A continuación se muestran y describen los resultados obtenidos en los diversos escenarios. Hay que mencionar, que para realizar el procesamiento de los datos se creó previamente en

MongoDB índices y en CouchDB y Couchbase se crearon “Views” para construir índices eficientes y mejorar el rendimiento global de sistema.

En CouchDB y Couchbase, se crearon cuatro vistas para indexar los documentos, de esta manera se pre-computa y almacena los resultados de las consultas programadas como vistas, asegurando una recuperación más rápida de las lecturas de datos. Estas vistas son índices construidos usando JavaScript y funciones MapReduce.

#### *4.2.4.1. Consulta general.*

En este escenario se realiza la consulta general de datos según el número de registros a retornarse. Hay que tener en cuenta que esta consulta no es una consulta masiva con varios usuarios concurrentes. El objetivo de esta prueba es observar el comportamiento de la base de datos, según el número de registros que se solicitan retornar. La Tabla 10 muestra el tiempo promedio que se demora en retornar resultados para los diferentes números de registros en MongoDB, CouchDB y Couchbase.

Tabla 10. Tiempos de respuesta de consulta general en arquitectura de replicación

| **Replicación** | | **Tiempo de consulta (milisegundos)** | |
|---|---|---|---|
| **# de registros recuperados** | **MongoDB** | **CouchDB** | **Couchbase** |
| **100** | 10 | 15 | 50 |
| **1000** | 90 | 70 | 145 |
| **10000** | 850 | 1525 | 1045 |
| **100000** | 8770 | 11455 | 11375 |

Fuente: Elaboración propia

Tabla 11. Tiempos de respuesta de consulta general en arquitectura de fragmentación

| **Fragmentación** | **Tiempo de consulta (milisegundos)** | |
|---|---|---|
| **# de registros recuperados** | **MongoDB** | **CouchDB** |
| **100** | 55 | 25 |
| **1000** | 130 | 100 |
| **10000** | 1085 | 1445 |
| **100000** | 9710 | 10820 |

Fuente: Elaboración propia

#### *4.2.4.2. Consulta masiva.*

Se realizarán tres tipos de consultas con diferentes grados de complejidad con el fin de obtener resultados más confiables del comportamiento de las bases de datos frente a cada tipo de consulta. El tiempo de respuesta se medirá en mili-segundos. Las siguientes tablas muestran el tiempo promedio de ejecución de las consultas en las tres bases de datos NoSQL:

Tabla 12. Tiempos de respuesta de consulta masiva de tweets con más de 1000 retweets en arquitectura de replicación con diferentes cláusulas *limit*

| | | Tiempo de consulta (milisegundos) | | |
|---|---|---|---|---|
| **Replicación** | **# de registros recuperados** | **MongoDB** | **CouchDB** | **Couchbase** |
| Todos los servidores funcionando | **1** | 940 | 1660 | 2440 |
| | **10** | 18880 | 4260 | 4540 |
| | **100** | 7680 | 9300 | 5040 |
| | **1000** | 76260 | 87940 | 32640 |
| Dos servidores funcionando | **1** | 1300 | 1500 | 4120 |
| | **10** | 2660 | 3860 | 4720 |
| | **100** | 8880 | 11480 | 5340 |
| | **1000** | 73420 | 88540 | 35680 |

Fuente: Elaboración propia

Tabla 13. Tiempos de respuesta de consulta masiva de tweets según la fecha de creación en arquitectura de replicación con diferentes cláusulas *limit*

| | | Tiempo de consulta (milisegundos) | | |
|---|---|---|---|---|
| **Replicación** | **# de registros recuperados** | **MongoDB** | **CouchDB** | **Couchbase** |
| Todos los servidores funcionando | **1** | 1380 | 1720 | 4280 |
| | **10** | 2380 | 2660 | 4440 |
| | **100** | 7780 | 4540 | 5060 |
| | **1000** | 78420 | 116120 | 33960 |
| Dos servidores funcionando | **1** | 1000 | 1460 | 4480 |
| | **10** | 1760 | 1720 | 4740 |
| | **100** | 8780 | 3760 | 5540 |

| | | | | |
|---|---|---|---|---|
| | **1000** | 45980 | 57280 | 36340 |

Fuente: Elaboración propia

Tabla 14. Tiempos de respuesta de consulta masiva según hashtag en arquitectura de replicación con diferentes cláusulas *limit*

| | | **Tiempo de consulta (milisegundos)** | | |
|---|---|---|---|---|
| **Replicación** | **# de registros recuperados** | **MongoDB** | **CouchDB** | **Couchbase** |
| Todos los servidores funcionando | **1** | 2000 | 10480 | 4260 |
| | **10** | 21760 | 11100 | 4500 |
| | **100** | 93520 | 15740 | 504 |
| | **1000** | 272160 | 55340 | 41940 |
| Dos servidores funcionando | **1** | 2100 | 15620 | 4680 |
| | **10** | 21320 | 16380 | 4740 |
| | **100** | 97260 | 21060 | 6400 |
| | **1000** | 2688880 | 53460 | 42320 |

Fuente: Elaboración propia

Tabla 15. Tiempos de respuesta de consulta masiva de tweets con más de 1000 retweets en arquitectura de fragmentación con diferentes cláusulas *limit*

| | | **Tiempo de consulta (milisegundos)** | |
|---|---|---|---|
| **Fragmentación** | **# de registros recuperados** | **MongoDB** | **CouchDB** |
| Todos los servidores funcionando | **1** | 1340 | 1780 |
| | **10** | 1800 | 4120 |
| | **100** | 8120 | 12620 |
| | **1000** | 84280 | 104340 |
| Dos servidores funcionando | **1** | 1200 | 1360 |
| | **10** | 2120 | 3880 |
| | **100** | 8340 | 11980 |
| | **1000** | 7238 | 92620 |

Fuente: Elaboración propia

Tabla 16. Tiempos de respuesta de consulta masiva de tweets según la fecha de creación en arquitectura de fragmentación con diferentes cláusulas *limit*

| | | Tiempo de consulta (milisegundos) | |
|---|---|---|---|
| **Fragmentación** | **# de registros recuperados** | **MongoDB** | **CouchDB** |
| Todos los servidores funcionando | **1** | 1340 | 1660 |
| | **10** | 2240 | 3880 |
| | **100** | 7880 | 7180 |
| | **1000** | 78080 | 106680 |
| Dos servidores funcionando | **1** | 1220 | 1300 |
| | **10** | 1760 | 2120 |
| | **100** | 7480 | 3980 |
| | **1000** | 71260 | 51720 |

Fuente: Elaboración propia

Tabla 17. Tiempos de respuesta de consulta masiva según hashtag en arquitectura de fragmentación con diferentes cláusulas *limit*

| | | Tiempo de consulta (milisegundos) | |
|---|---|---|---|
| **Fragmentación** | **# de registros recuperados** | **MongoDB** | **CouchDB** |
| Todos los servidores funcionando | **1** | 1400 | 9940 |
| | **10** | 28140 | 10960 |
| | **100** | 85340 | 16060 |
| | **1000** | 255620 | 56080 |
| Dos servidores funcionando | **1** | 1660 | 9480 |
| | **10** | 27260 | 15600 |
| | **100** | 89920 | 20340 |
| | **1000** | 260660 | 46940 |

Fuente: Elaboración propia

Al mismo tiempo que se realizaron las consultas en MongoDB, CouchDB y Couchbase, se monitorizó el estado de los recursos de los servidores del entorno distribuido construido, a continuación se muestran los resultados correspondientes.

Tabla 18. Consumo de recursos promedio de las bases de datos

| Recurso | MongoDB | CouchDB | Couchbase |
|---|---|---|---|
| **Memoria RAM** | 23,78 % | 26,16 % | 58,1 % |
| **Procesador** | 26,16 % | 30,29 % | 60,30 % |

Fuente: Elaboración propia

Al mismo tiempo que se realizaron las consultas en MongoDB, CouchDB y Couchbase, se monitorizó el estado de los recursos de los servidores del ambiente distribuido construido, a continuación se muestran los resultados correspondientes al consumo de recursos.

### 4.2.5. Pruebas de disponibilidad.

Como ya se conoce, las pruebas de disponibilidad de datos se las llevaron a cabo con el fin de analizar el comportamiento de la base de datos ante un fallo en uno de los servidores del sistema distribuido.

#### *4.2.5.1. Caída en uno de los servidores.*

Para llevar a efecto esta prueba se realiza la simulación de la caída de uno de los servidores en entorno distribuido de la base de datos. A continuación se analiza el comportamiento del entorno distribuido cuando se produce la caída inesperada de uno de los servidores. En el caso de MongoDB, cuando cae el servidor primario del conjunto de réplicas, uno de los servidores secundarios pasa a ocupar su lugar mediante la votación que se desencadena entre los nodos secundarios para la elección del nuevo servidor primario. En la Figura 101 se muestra todos los servidores del conjunto de réplicas con los que se está trabajando.

```
{
        "_id" : 1,
        "name" : "192.168.1.101:27052",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 52880,
        "optime" : {
                "ts" : Timestamp(1499715290, 1),
                "t" : NumberLong(10)
        },
        "optimeDate" : ISODate("2017-07-10T19:34:50Z"),
        "electionTime" : Timestamp(1499707364, 1),
        "electionDate" : ISODate("2017-07-10T17:22:44Z"),
        "configVersion" : 3,
        "self" : true
},
{
        "_id" : 2,
        "name" : "192.168.1.103:27053",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 7936,
        "optime" : {
                "ts" : Timestamp(1499715290, 1),
                "t" : NumberLong(10)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1499715290, 1),
                "t" : NumberLong(10)
        },
        "optimeDate" : ISODate("2017-07-10T19:34:50Z"),
        "optimeDurableDate" : ISODate("2017-07-10T19:34:50Z"),
        "lastHeartbeat" : ISODate("2017-07-10T19:34:58.476Z"),
        "lastHeartbeatRecv" : ISODate("2017-07-10T19:34:58.604Z"),
        "pingMs" : NumberLong(44),
        "syncingTo" : "192.168.1.101:27052",
        "configVersion" : 3
}
```


Figura 101. MongoDB – Servidores del conjunto de réplicas
Fuente: Elaboración propia

Ahora veamos la situación de los servidores del conjunto de réplicas después de que el servidor primario ha fallado, en la Figura 102 se observa como uno de los servidores secundarios se ha convertido en el nuevo servidor primario. Tras las elecciones el nuevo servidor primario es el que realiza las operaciones hasta que el antiguo primario se integre nuevamente al conjunto de réplicas.

```
{
        "_id" : 1,
        "name" : "192.168.1.101:27052",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 25,
        "optime" : {
                "ts" : Timestamp(1499715870, 1),
                "t" : NumberLong(11)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1499715870, 1),
                "t" : NumberLong(11)
        },
        "optimeDate" : ISODate("2017-07-10T19:44:30Z"),
        "optimeDurableDate" : ISODate("2017-07-10T19:44:30Z"),
        "lastHeartbeat" : ISODate("2017-07-10T19:44:31.348Z"),
        "lastHeartbeatRecv" : ISODate("2017-07-10T19:44:31.821Z"),
        "pingMs" : NumberLong(64),
        "syncingTo" : "192.168.1.103:27053",
        "configVersion" : 3
},
{
        "_id" : 2,
        "name" : "192.168.1.103:27053",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 448,
        "optime" : {
                "ts" : Timestamp(1499715870, 1),
                "t" : NumberLong(11)
        },
        "optimeDate" : ISODate("2017-07-10T19:44:30Z"),
        "electionTime" : Timestamp(1499715848, 1),
        "electionDate" : ISODate("2017-07-10T19:44:08Z"),
        "configVersion" : 3,
        "self" : true
}
```


Figura 102. MongoDB – Situación después de la caída del nodo primario
Fuente: Elaboración propia

Dado que MongoDB usa el mecanismo journaling para proteger la integridad de los datos en caso de un fallo en el servidor producido por fallos de energía, reinicios inesperados o fallos de hardware. Con journaling se guarda las operaciones que va a realizar en un archivo del disco duro antes de aplicarlas, en caso de producirse un fallo inesperado, MongoDB podrá restaurar el estado de la base de datos a un estado consistente. Cuando el antiguo servidor primario se ha recuperado del fallo, se inician nuevas elecciones y este vuelve a colocarse como el miembro primario.

```
{
        "_id" : 1,
        "name" : "192.168.1.101:27052",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 26,
        "optime" : {
                "ts" : Timestamp(1499716138, 1),
                "t" : NumberLong(14)
        },
        "optimeDate" : ISODate("2017-07-10T19:48:58Z"),
        "electionTime" : Timestamp(1499716136, 1),
        "electionDate" : ISODate("2017-07-10T19:48:56Z"),
        "configVersion" : 3,
        "self" : true
},
{
        "_id" : 2,
        "name" : "192.168.1.103:27053",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 9,
        "optime" : {
                "ts" : Timestamp(1499716002, 1),
                "t" : NumberLong(13)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1499716002, 1),
                "t" : NumberLong(13)
        },
        "optimeDate" : ISODate("2017-07-10T19:46:42Z"),
        "optimeDurableDate" : ISODate("2017-07-10T19:46:42Z"),
        "lastHeartbeat" : ISODate("2017-07-10T19:48:59.070Z"),
        "lastHeartbeatRecv" : ISODate("2017-07-10T19:48:59.082Z"),
        "pingMs" : NumberLong(12),
        "configVersion" : 3
}
```


Figura 103. MongoDB – Situación tras el regreso del antiguo servidor primario
Fuente: Elaboración propia

En CouchDB se configuró el mecanismo de replicación como maestro-esclavo (A hacia B y A hacia C), con lo cual cualquier modificación que se realice sobre el servidor principal (A) se traspasará de manera instantánea al resto de servidores. En caso de la caída en uno de los servidores esclavos, el proceso de replicación queda pausado hasta que el servidor de destino se levante de nuevo.

```
[
  {
    "node": "couchdb@192.168.1.102",
    "pid": "<0.1232.0>",
    "changes_pending": 1278,
    "checkpoint_interval": 30000,
    "checkpointed_source_seq": "7199333-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f
    "continuous": true,
    "database": null,
    "doc_id": null,
    "doc_write_failures": 0,
    "docs_read": 0,
    "docs_written": 0,
    "missing_revisions_found": 0,
    "replication_id": "3ebbac405997da665c3a90c057ce52ab+continuous",
    "revisions_checked": 7533617,
    "source": "http://192.168.1.102:5984/db_tweets/",
    "source_seq": "7533617-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f___z8rgzmJwbLKJx
    "started_on": 1501791364,
    "target": "http://192.168.1.101:5984/replica-tweets/",
    "through_seq": "7533617-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f___z8rgzmJwbLKJ
    "type": "replication",
    "updated_on": 1501791805,
    "user": "Admin"
  },
  {
    "node": "couchdb@192.168.1.102",
    "pid": "<0.8098.0>",
    "changes_pending": 18505,
    "checkpoint_interval": 30000,
    "checkpointed_source_seq": "1494377-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f___
    "continuous": true,
    "database": null,
    "doc_id": null,
    "doc_write_failures": 0,
    "docs_read": 0,
    "docs_written": 0,
    "missing_revisions_found": 0,
    "replication_id": "f8d68eec279858e166f95366d74228dc+continuous",
    "revisions_checked": 1530692,
    "source": "http://192.168.1.102:5984/db_tweets/",
    "source_seq": "1531192-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f___z8rgzmJgXulai
    "started_on": 1501791371,
    "target": "http://192.168.1.103:5984/replica-tweets/",
    "through_seq": "1530409-g1AAAACveJzLYWBgYMpgTmEQTc4vTc5ISXIwtDTSMzSz0DPUMzQwygFJJzIk1f___z8rgzmJgXv5w
    "type": "replication",
    "updated_on": 1501791804,
    "user": "Admin"
  }
]
```


Figura 104. CouchDB – Tareas de replicación en servidor maestro
Fuente: Elaboración propia

Como se puede observar en la Figura 104, las tareas de replicación solo transfieren los cambios en una sola dirección. Cuando se produce la caída en el servidor maestro las tareas de replicación permanecen pausadas, sin embargo, la base de datos replicada es capaz de responder a las solicitudes de lectura de los usuarios. Por otro lado, para lograr que los cambios se transfieran bidireccionalmente, es posible establecer dos tareas de replicación de manera que cuando un cambio se replica desde la base de datos de A hacia B en la primera tarea, la segunda tarea de B hacia A, descubrirá que el nuevo cambio en B ya existe y esperará a más cambios.

En el caso de la base de datos Couchbase cada documento se replica automáticamente a un servidor diferente del clúster, asegurando así la alta disponibilidad en caso de que un servidor este fuera de servicio. Cuando un servidor no responde, el orquestador que fue elegido para

supervisar el clúster notifica a los nodos del clúster y promueve las réplicas que existen en otras partes del clúster al estado activo.

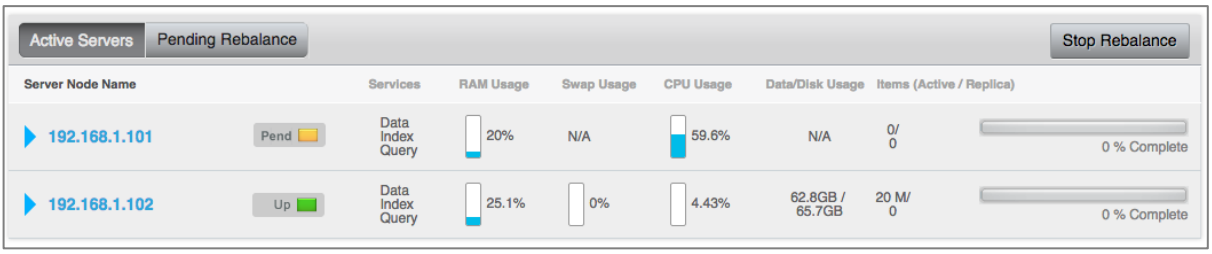



Figura 105. Couchbase – Proceso de rebalanceo desde orquestador
Fuente: Elaboración propia

Si el orquestador falla o se pierde la comunicación con el grupo por cualquier motivo, los nodos restantes detectan el fallo cuando dejan de recibir sus notificaciones, por lo que se elige rápidamente un nuevo orquestador. Esto se realiza inmediatamente y es transparente para las operaciones del clúster.

### 4.3. Análisis de resultados

Durante la fase de carga, se realizó la inserción de los 20 millones de datos que se utilizados para evaluar el rendimiento en cada una de las bases de datos. Luego de realizar la importación de los datos se obtuvo resultados similares en MongoDB y Couchbase. En MongoDB el tiempo de carga fue de 52 minutos y en Couchbase fue de 45 minutos. Sin embargo, en el caso de CouchDB, el tiempo de carga superó los 60 minutos.

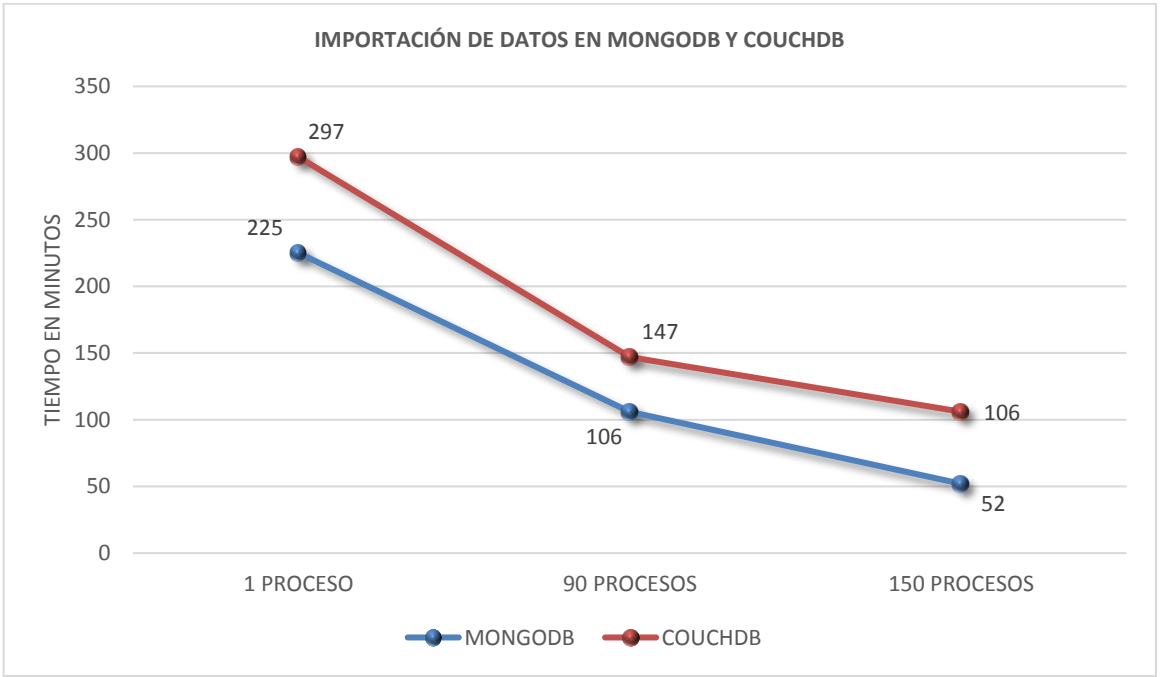



Figura 106. Tiempo de carga de importación de datos en MongoDB y CouchDB
Fuente: Elaboración propia

La Figura 106 muestran el tiempo en minutos de la importación de datos que se realizó utilizando 1, 90 y 150 procesos ejecutándose en el caso de MongoDB y CouchDB. Para la base de datos Couchbase el parámetro -t de cbtransfer, el límite de subprocesos simultáneos es de 18, es por ello que se realizó la carga de datos con 1 y 18 procesos en ejecución.

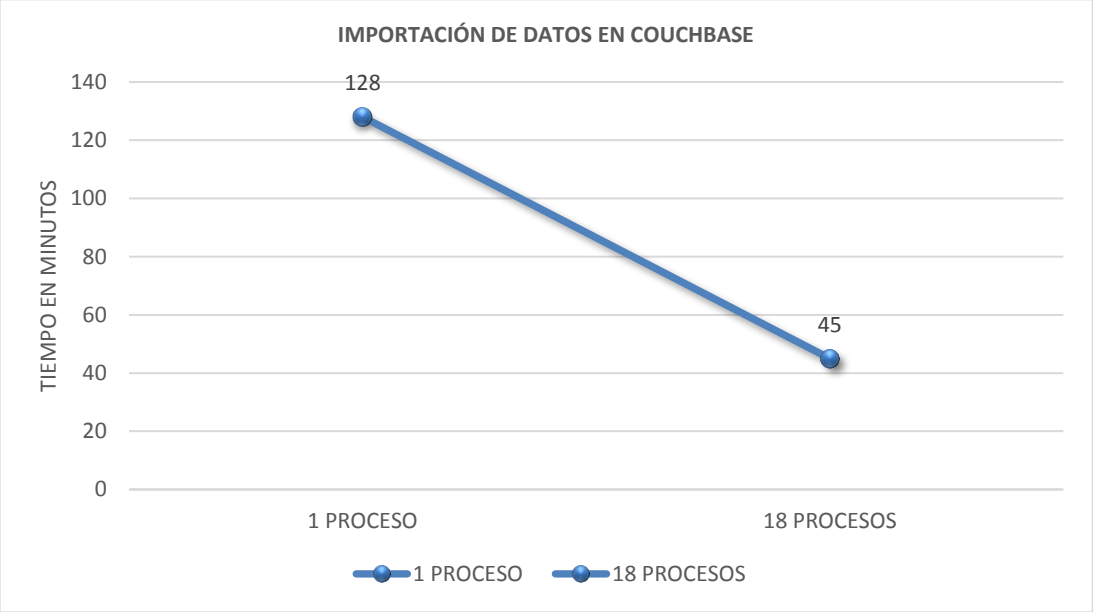



Figura 107. Tiempo de carga de importación de datos en Couchbase
Fuente: Elaboración propia

En la fase de pruebas de los escenarios propuestos, el rendimiento de las bases de datos se medirá en mili-segundos. En el escenario donde se realiza una consulta general para obtener 100, 1000, 10000 y 100000 sobre la arquitectura de replicación se observa que la base de datos MongoDB proporciona los mejores tiempos de respuesta. En cambio, la base de datos CouchDB y Couchbase mantienen tiempos de respuesta casi similares a medida que el número de registros recuperados se incrementa.

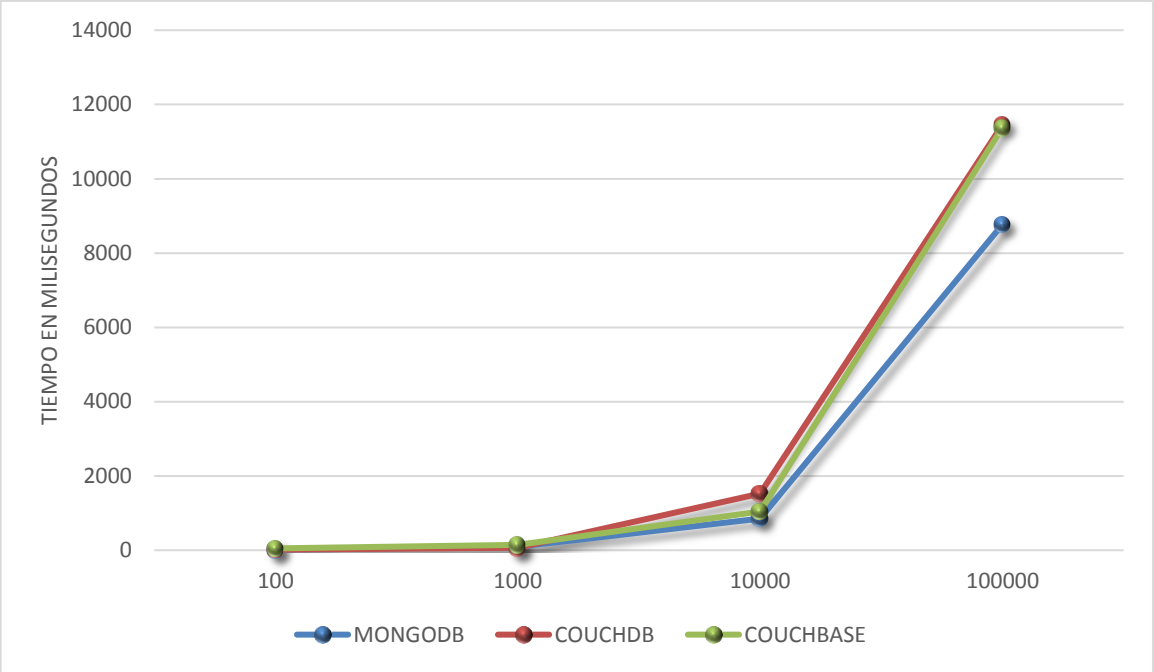



Figura 108. Tiempo promedio de consulta general en arquitectura de replicación
Fuente: Elaboración propia

De igual manera, al realizar la misma consulta sobre la arquitectura de fragmentación se observa que al igual que en la arquitectura de replicación de las bases de datos implementadas, la base de datos MongoDB proporciona el mejor rendimiento.

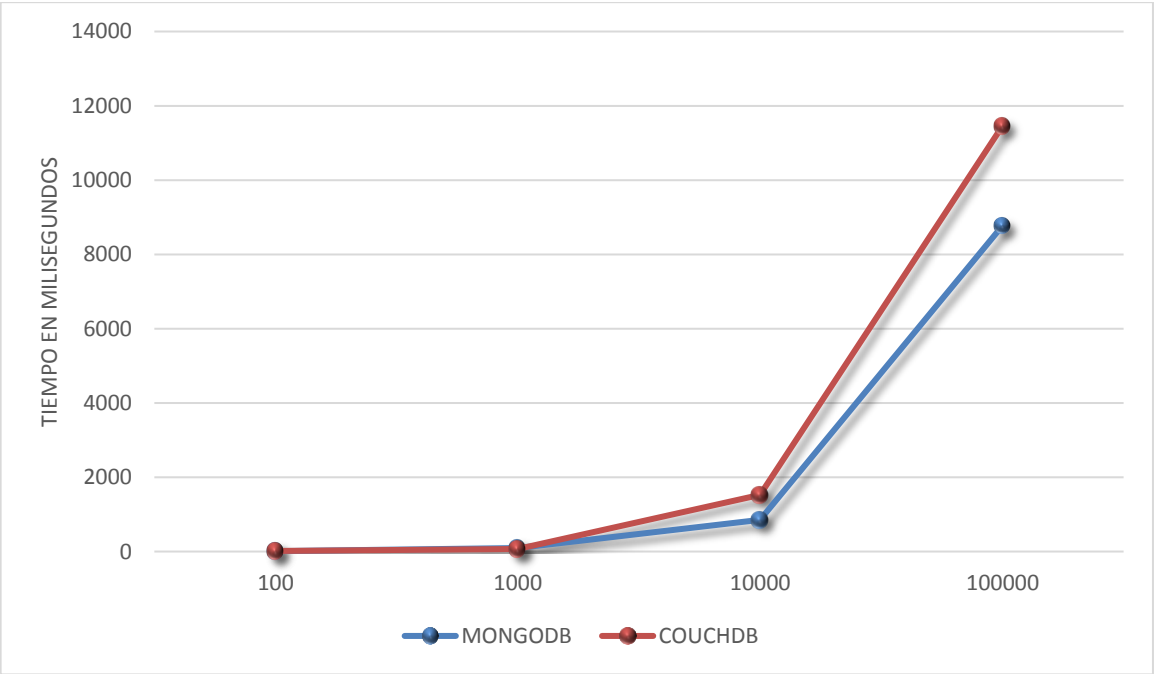



Figura 109. Tiempo promedio de consulta general en arquitectura de fragmentación
Fuente: Elaboración propia

El segundo escenario de prueba que consiste en la consulta masiva de datos para diferentes tipos de consulta con grados de complejidad distintos y 1000 usuarios concurrentes, se

realizan las pruebas sobre las arquitecturas de replicación y fragmentación con todos los servidores funcionando. A continuación se muestra de forma gráfica los resultados obtenidos en la primera consulta de prueba realizada. En la Figura 110 se observa los resultados que se obtuvieron en la arquitectura de replicación y en la Figura 111 se presentan los resultados obtenidos al realizar la misma consulta sobre la arquitectura de fragmentación.

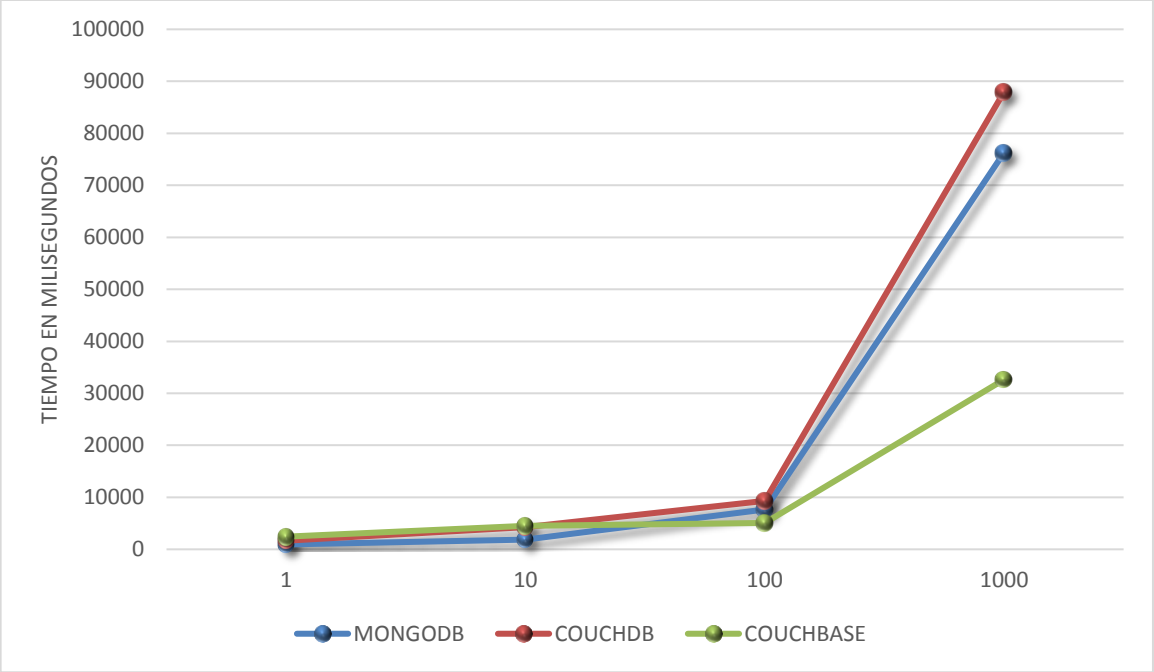



Figura 110. Tiempo promedio de Consulta Masiva 1 en arquitectura de replicación
Fuente: Elaboración propia

Los tiempos de respuesta en la arquitectura de replicación muestran que las tres bases de datos mantienen tiempos similares pero mientras el número de registros recuperados se incrementa, la base de datos Couchbase es la que proporciona el mejor tiempo de respuesta. Por otra parte, en la arquitectura de fragmentación se observa que la base de datos MongoDB alcanza los mejores tiempos de respuesta en comparación con la base de datos CouchDB.

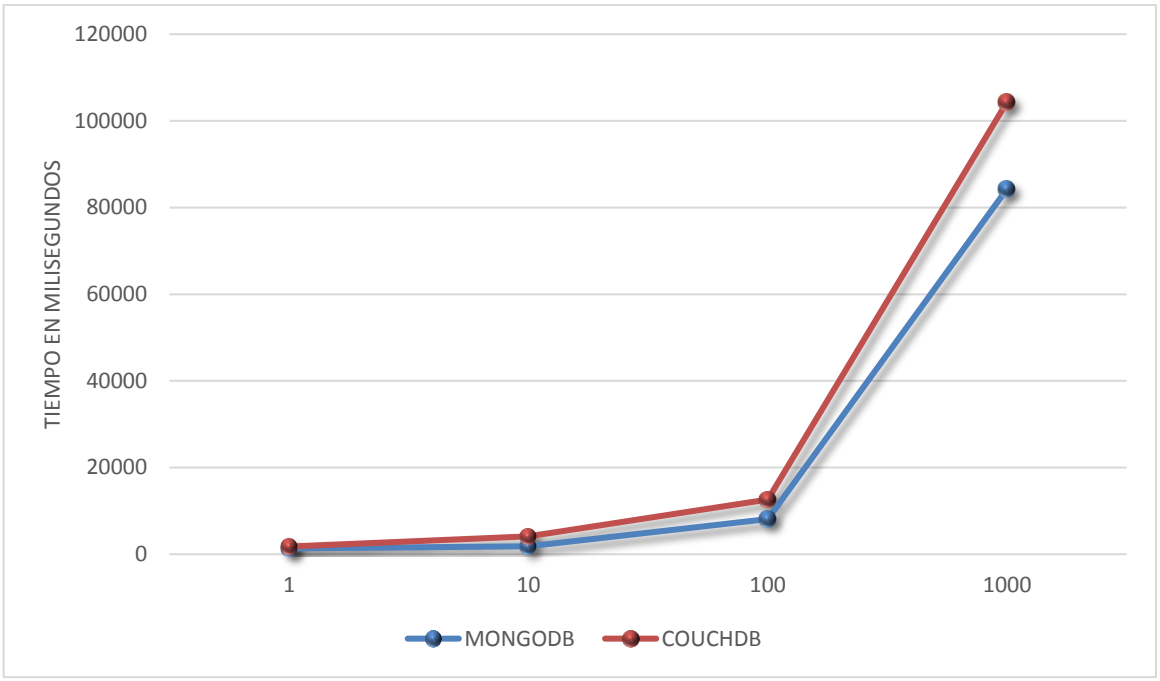



Figura 111. Tiempo promedio de Consulta Masiva 1 en arquitectura de fragmentación
Fuente: Elaboración propia

Las siguientes figuras representan los resultados obtenidos en cuanto a la segunda consulta realizada. Al igual que en la primera consulta se ejecutó esta prueba en el escenario en donde todos los servidores estaban funcionando. Como se observa en la Figura 112 en esta prueba la base de datos Couchbase ha logrado el mejor rendimiento. En relación con la arquitectura de fragmentación en la Figura 113 se observa que la base de datos MongoDB es la que proporciona un buen rendimiento mientras el número de registros recuperados se incrementa.

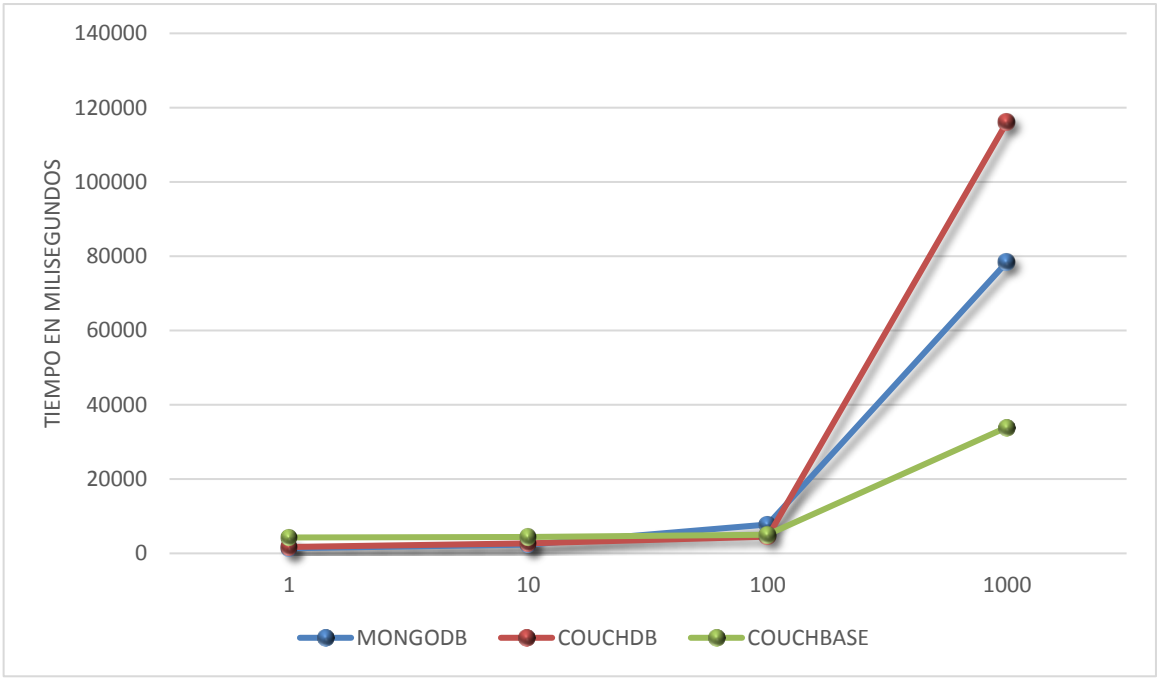



Figura 112. Tiempo promedio de Consulta Masiva 2 en arquitectura de replicación
Fuente: Elaboración propia

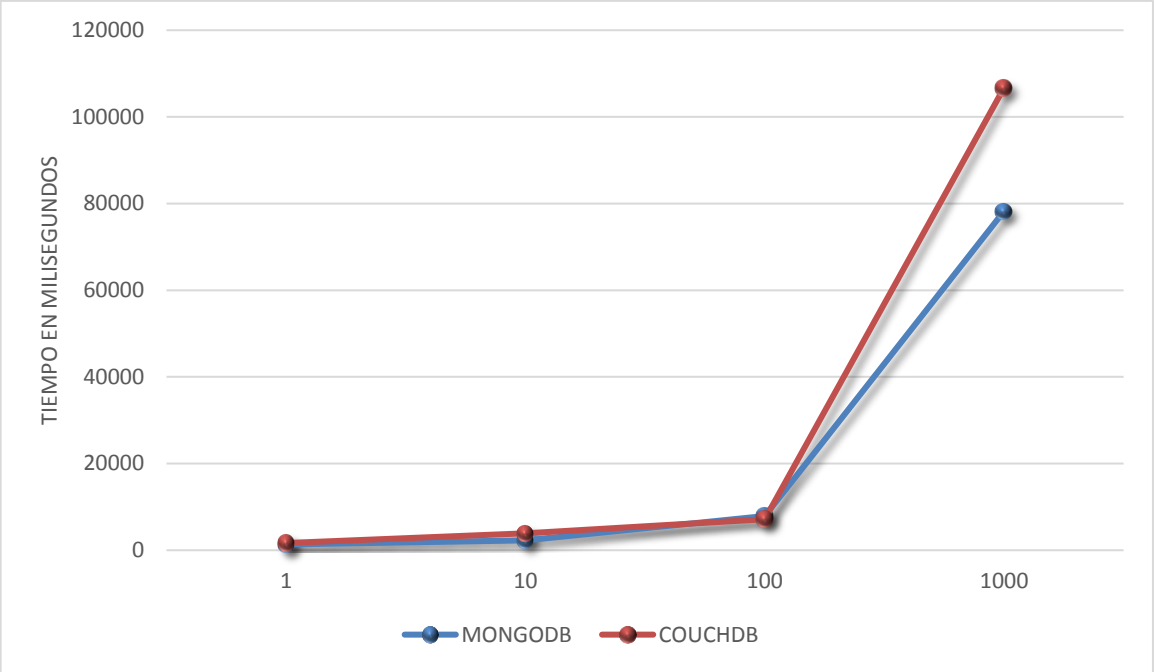



Figura 113. Tiempo promedio de Consulta Masiva 2 en arquitectura de fragmentación
Fuente: Elaboración propia

Finalmente, en cuanto a la última consulta de prueba que busca los tweets que contengan el hashtag #SocialWeb, los resultados mostraron que la base de datos CouchDB y Couchbase son las que proporcionan los mejores resultados y la base de datos MongoDB obtuvo los tiempos de respuestas más altos.

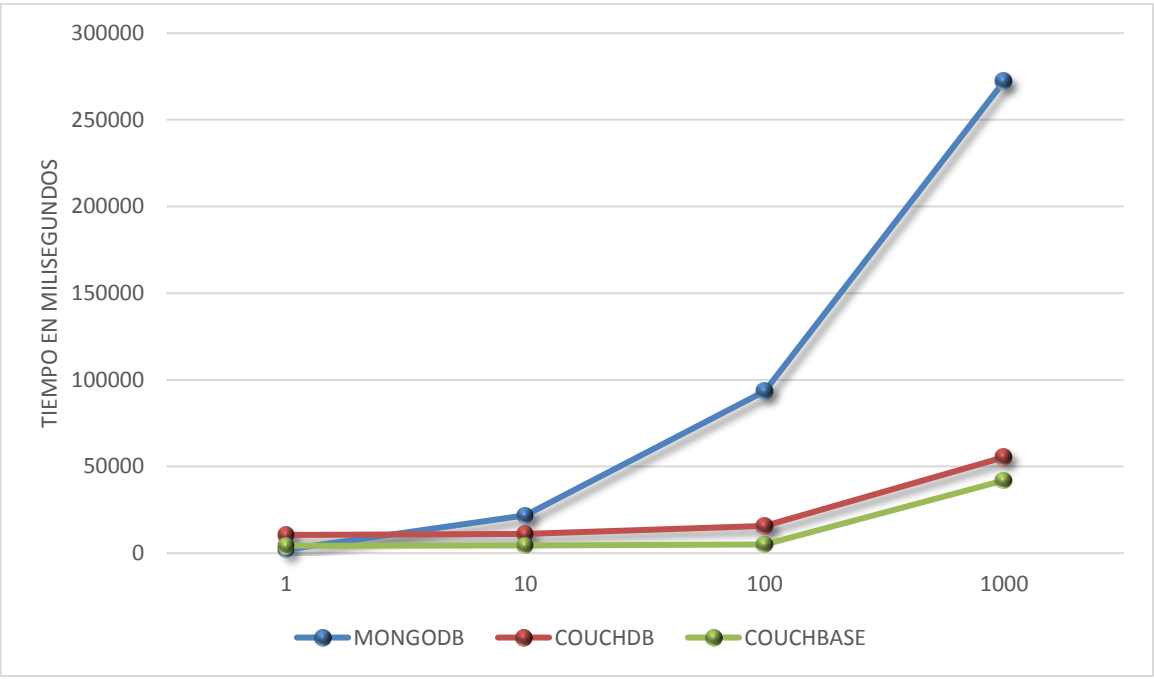



Figura 114. Tiempo promedio de Consulta Masiva 3 en arquitectura de replicación
Fuente: Elaboración propia

Es probable que las vistas que se crearon en CouchDB y Couchbase utilizando las funciones MapReduce mejorarán considerablemente los tiempos de respuesta para esta consulta. En la Figura 114 y Figura 115, se muestran de forma gráfica los resultados obtenidos para cada una de las arquitecturas en las que se realizó las pruebas.

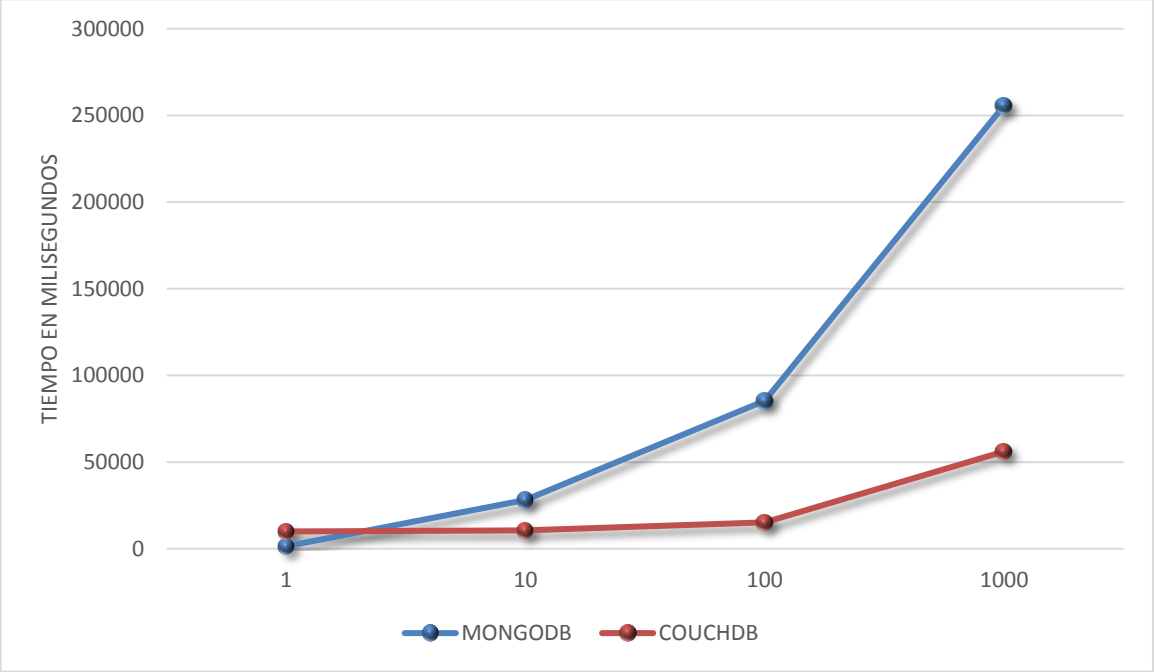



Figura 115. Tiempo promedio de Consulta Masiva 3 en arquitectura de fragmentación
Fuente: Elaboración propia

Como ya se ha mencionado en secciones anteriores, el tercer escenario de prueba al igual que el segundo escenario, consiste en la consulta masiva de datos para diferentes tipos de consulta y con 1000 usuarios concurrentes. Pero, en este caso se ha provocado la caída en uno de los servidores antes de realizar las pruebas para cada una de las arquitecturas.

En las siguientes figuras se observa los resultados obtenidos en la ejecución de la consulta primera consulta sobre el escenario en donde uno de los servidores está fuera de servicio. En la Figura 116 se observa que la base de datos Couchbase alcanza mejores resultados en esta prueba. La Figura 117 representa los resultados obtenidos en la arquitectura de fragmentación de las bases de datos MongoDB y CouchDB, en la gráfica se puede observar que MongoDB provee el mejor rendimiento. De manera comparativa con las Figura 110 y Figura 111 que muestra los resultados para la misma consulta pero sobre un escenario en que todos los servidores están funcionando, al igual que en este escenario la base de datos Couchbase continúa siendo la que alcanza los mejores resultados en la arquitectura de replicación y la base de datos MongoDB proporciona el mejor rendimiento en la arquitectura de fragmentación.

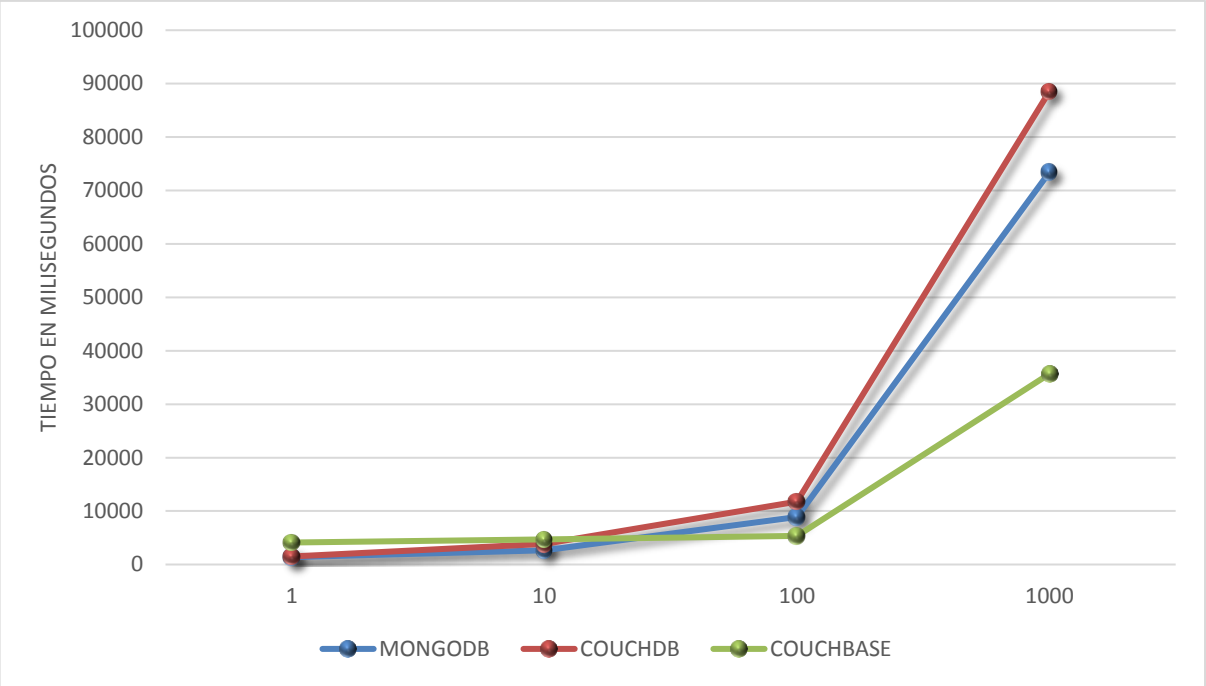



Figura 116. Tiempo promedio de Consulta Masiva 1 en arquitectura de replicación con fallo en un servidor
Fuente: Elaboración propia

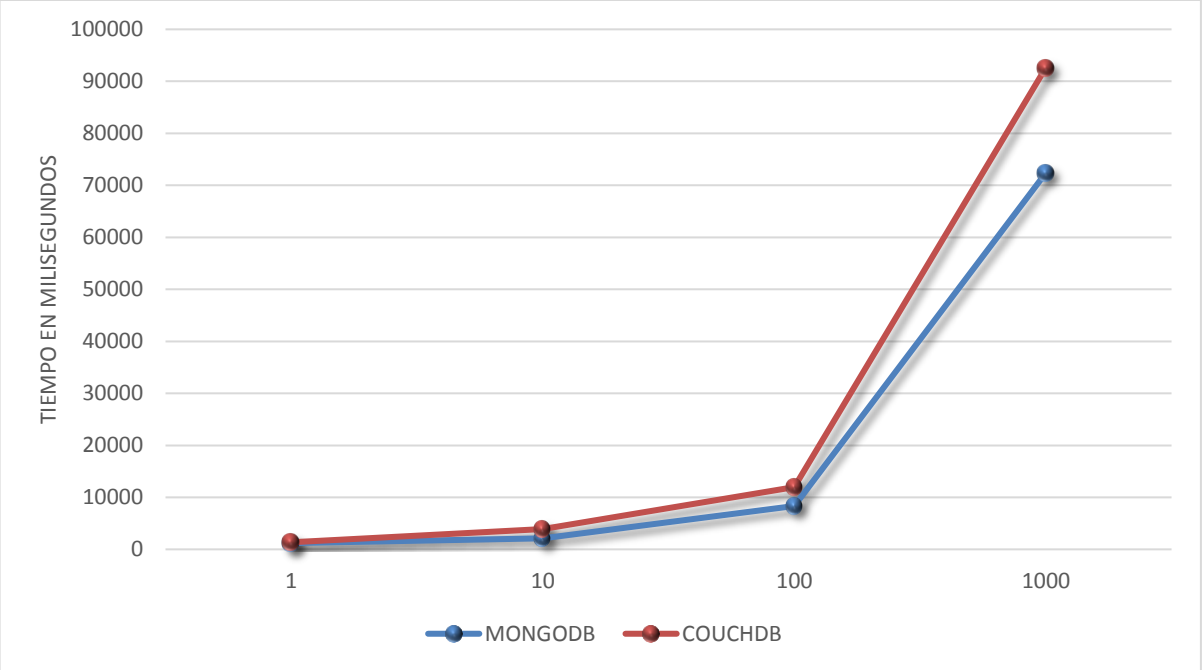



Figura 117. Tiempo promedio de Consulta Masiva 1 en arquitectura de fragmentación con fallo en un servidor
Fuente: Elaboración propia

Para la segunda prueba realizada sobre este escenario que consiste en la obtención de tweets según su fecha de creación, los resultados obtenidos muestran que en la arquitectura de replicación las tres bases de datos proporcionan tiempos de respuestas no muy distantes. Es probable que con la caída del servidor, el nuevo servidor principal al que se asignan las

peticiones tenga más recursos disponibles. De manera comparativa con la Figura 112 que realiza la misma consulta en un escenario en donde todos los servidores están funcionando, se observa que Couchbase, al igual que en este escenario obtuvo los mejores resultados en la arquitectura de replicación.

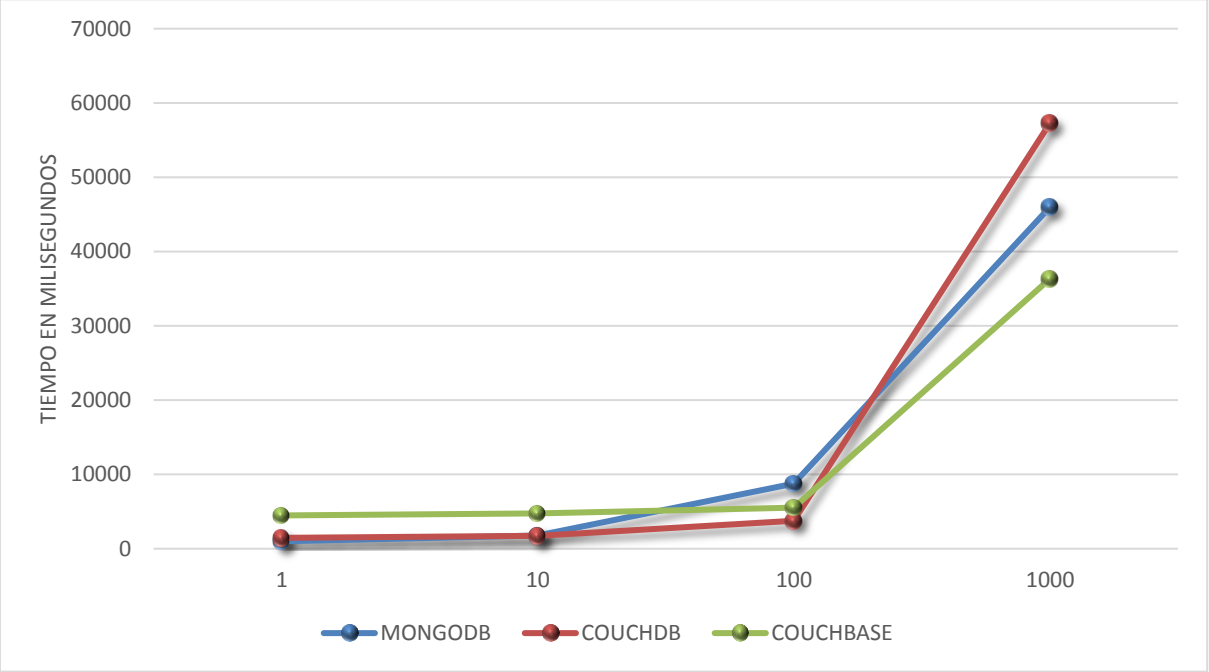



Figura 118. Tiempo promedio de Consulta Masiva 2 en arquitectura de replicación con fallo en un servidor
Fuente: Elaboración propia

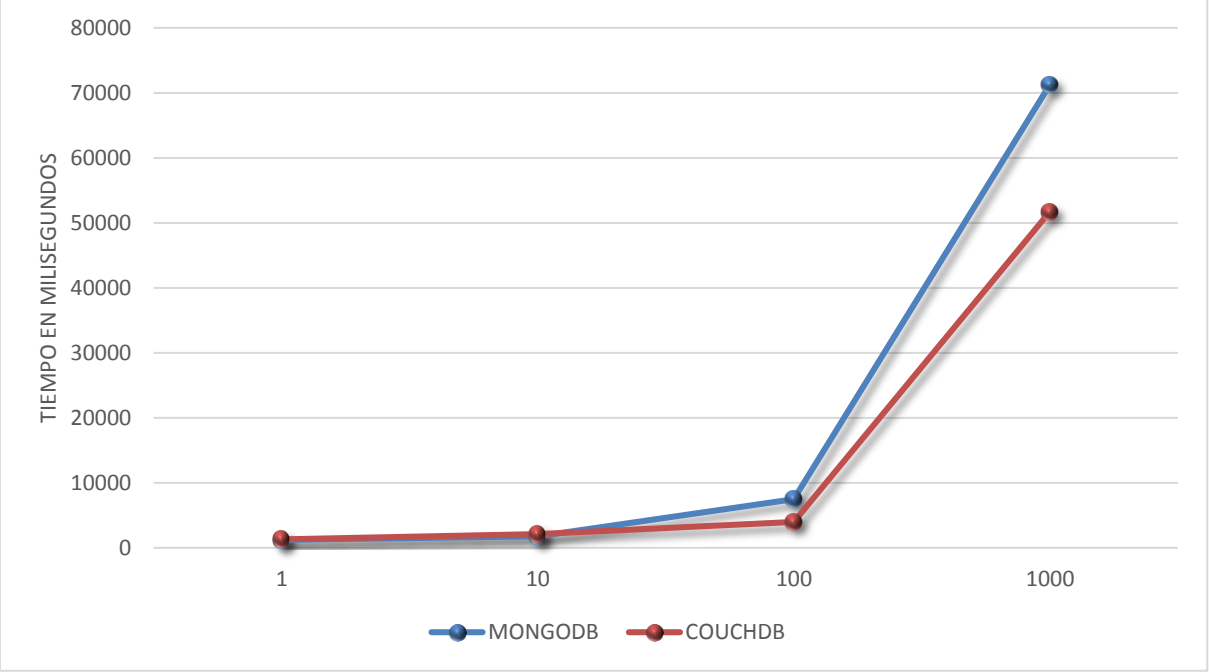



Figura 119. Tiempo promedio de Consulta Masiva 2 en arquitectura de fragmentación con fallo en un servidor
Fuente: Elaboración propia

En la última consulta que busca los tweets que contengan un hashtag especifico, los resultados obtenidos muestran que la base de datos CouchDB y Couchbase proporcionan los mejores tiempos de respuesta y MongoDB presenta los tiempos de respuesta más altos.

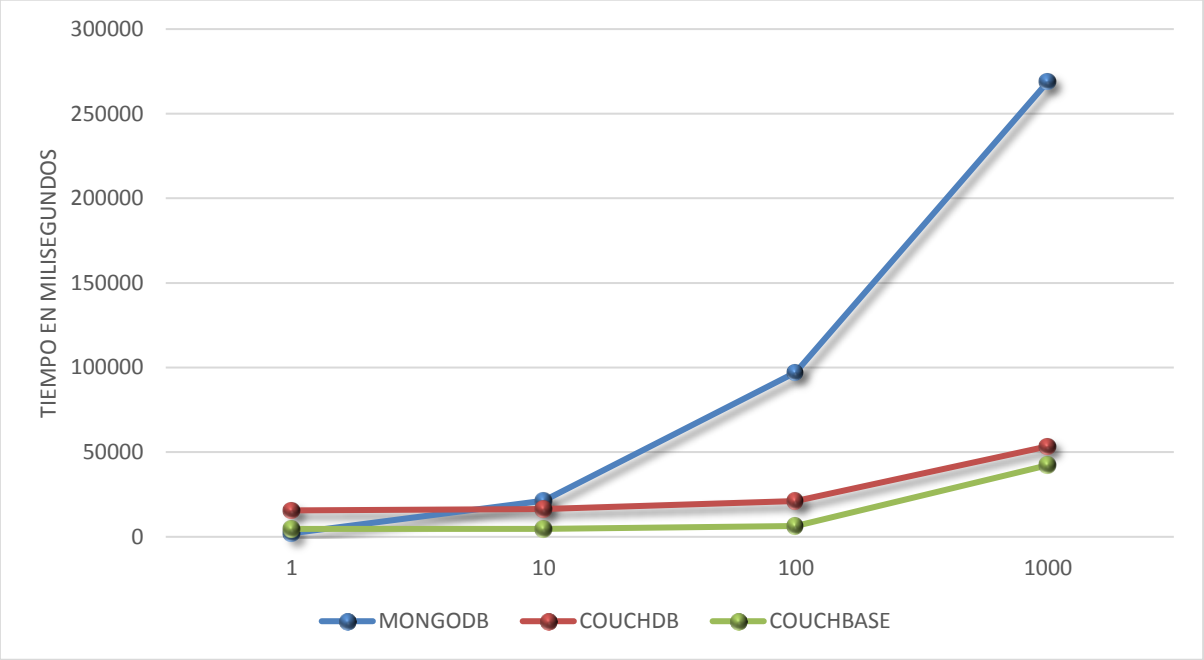



Figura 120. Tiempo promedio de Consulta Masiva 3 en arquitectura de replicación con fallo en un servidor
Fuente: Elaboración propia

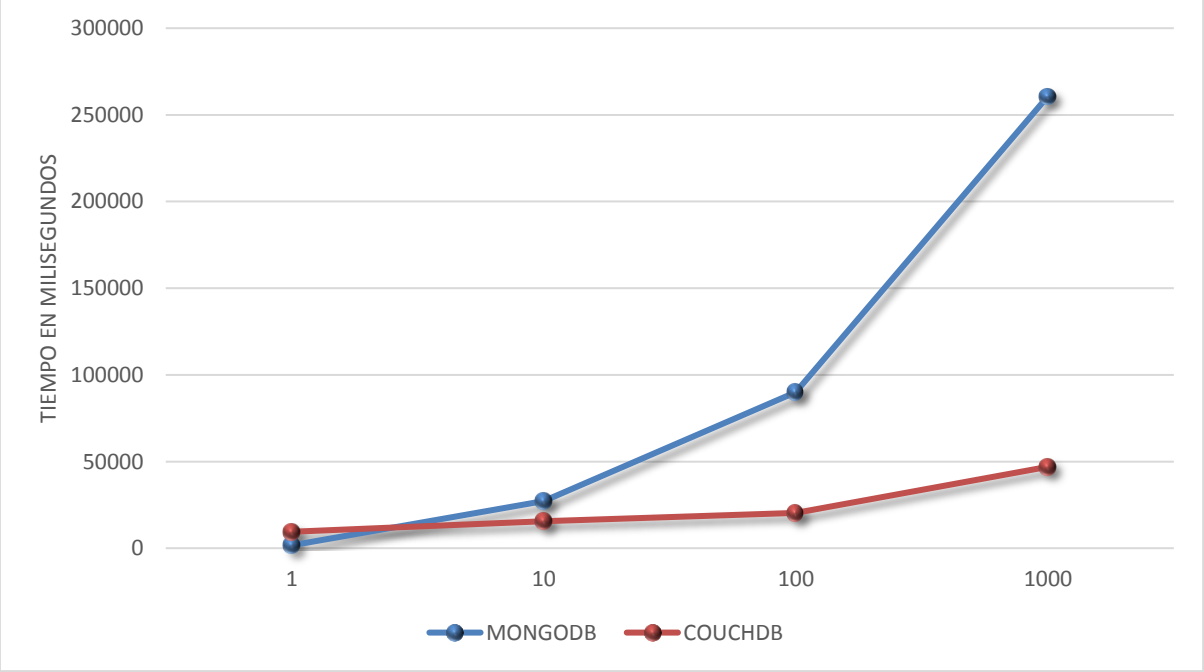



Figura 121. Tiempo promedio de Consulta Masiva 3 en arquitectura de fragmentación con fallo en un servidor
Fuente: Elaboración propia

En la Figura 120 y Figura 121, se muestran de forma gráfica los resultados obtenidos para cada una de las arquitecturas que se ha implementado. Tales resultados muestran un comportamiento similar en comparación a la Figura 114 y Figura 115, donde se muestran los resultados para la misma consulta pero sobre un escenario en que todos los servidores están funcionando.

En las Tabla 19 y Tabla 20 se resumen los aspectos evaluados en los escenarios de prueba sobre arquitecturas de replicación y fragmentación. En cada prueba se ha asignado ponderaciones de 1, 2 y 3, así pues la base de datos que ha logrado el mejor tiempo de respuesta obtendrá una puntuación de 3, y la que ha logrado el tiempo de respuesta más bajo una puntuación de 1, y de la misma forma de acuerdo para los recursos de RAM y CPU que ha utilizado la base de datos.

Tabla 19. Resumen de evaluación de pruebas en arquitectura de replicación

| Escenario | Prueba | MongoDB | CouchDB | Couchbase |
|---|---|---|---|---|
| **Consulta general** | *Consulta general* | 3 | 2 | 1 |
| **Consulta masiva con todos los servidores funcionando** | *Consulta por número de retweets* | 2 | 1 | 3 |
| | *Consulta por fecha de creación* | 2 | 1 | 3 |
| | *Consulta por hashtags* | 1 | 2 | 3 |
| **Consulta masiva con un servidor caído** | *Consulta por número de retweets* | 2 | 1 | 3 |
| | *Consulta por fecha de creación* | 2 | 1 | 3 |
| | *Consulta por hashtags* | 1 | 2 | 3 |
| **Recursos** | *Mejor consumo de memoria RAM* | 3 | 2 | 1 |
| | *Mejor consumo de procesador* | 3 | 2 | 1 |

Fuente: Elaboración propia

En la Tabla 20 se resumen los escenarios de prueba evaluados en la arquitectura de fragmentación. De la misma manera se ha asignado a cada prueba ponderaciones de 1, 2 y 3, es así que la base de datos alcanzó el mejor tiempo de respuesta obtendrá una puntuación de 3, y la que ha logrado el tiempo de respuesta más bajo una puntuación de 1, y de la misma forma de acuerdo para los recursos de RAM y CPU que ha utilizado la base de datos.

Tabla 20. Resumen de evaluación de pruebas en arquitectura de fragmentación

| Escenario | Prueba | MongoDB | CouchDB |
|---|---|---|---|
| **Consulta general** | *Consulta general* | 2 | 1 |
| **Consulta masiva con todos los servidores funcionando** | *Consulta por número de retweets* | 2 | 1 |
| | *Consulta por fecha de creación* | 2 | 1 |
| | *Consulta por hashtags* | 2 | 1 |
| **Consulta masiva con un servidor caído** | *Consulta por número de retweets* | 2 | 1 |
| | *Consulta por fecha de creación* | 2 | 1 |
| | *Consulta por hashtags* | 2 | 1 |
| **Recursos** | *Mejor consumo de memoria RAM* | 2 | 1 |
| | *Mejor consumo de procesador* | 2 | 1 |

Fuente: Elaboración propia

Con base en los resultados descritos en la Tabla 19, es evidente que la base de datos Couchbase se considera la más apropiada, a la hora de gestionar grandes volúmenes de datos. Sin embargo, siendo la base de datos Couchbase evaluada únicamente sobre una arquitectura de replicación se puede concluir que se está base de datos proporciona el mejor rendimiento en cuanto a la replicación de datos, y se recomienda utilizarla cuando se disponga servidores con potentes recursos de espacio en disco y memoria RAM. Por otra parte, en las pruebas realizadas a la arquitectura de fragmentación y con base en los resultados presentados en la Tabla 20 es evidente que MongoDB es la mejor opción al momento de implementar un entorno distribuido escalable, tolerante a fallos y de alto rendimiento.

En las Figuras 90 y 91 se puede observar de manera gráfica la evaluación de las bases de datos NoSQL como una solución para el manejo de grandes volúmenes de datos a través de una arquitectura escalable y tolerante a fallo. En este caso particular la información gestionada fueron 20 millones de datos. A continuación se presenta el porcentaje de éxito de implementar como solución las bases de datos evaluadas en los apartados anteriores sobre un entorno distribuido para replicación de datos y fragmentación de datos

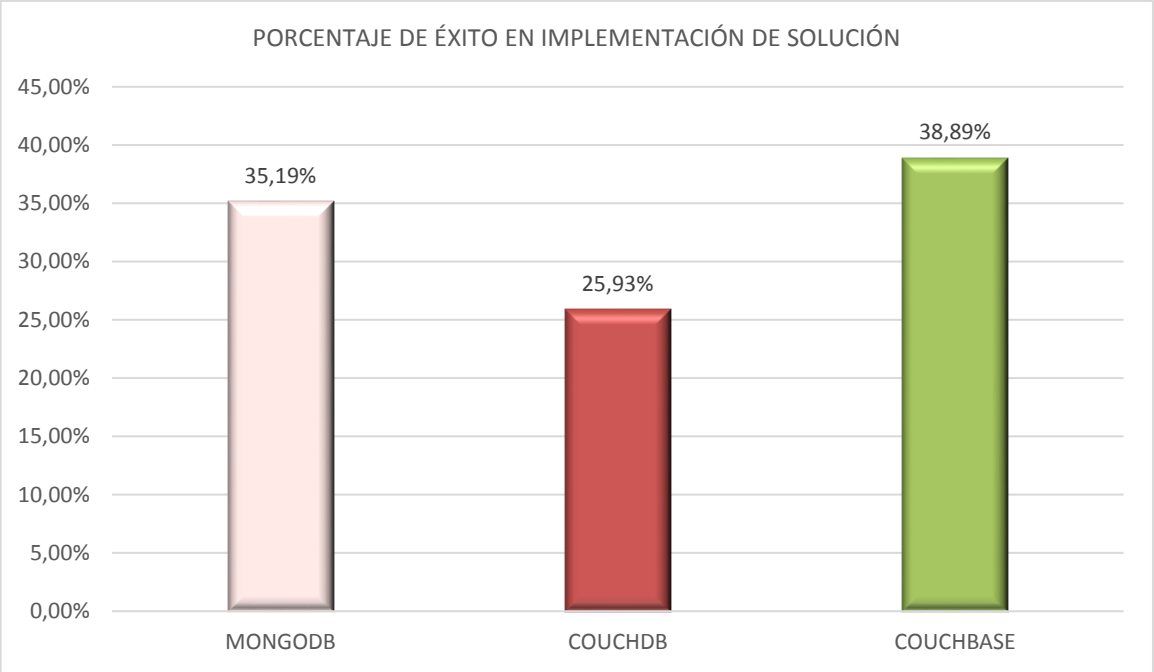



Figura 122. Resultado de evaluación de bases de datos NoSQL en arquitectura de replicación
Fuente: Elaboración propia

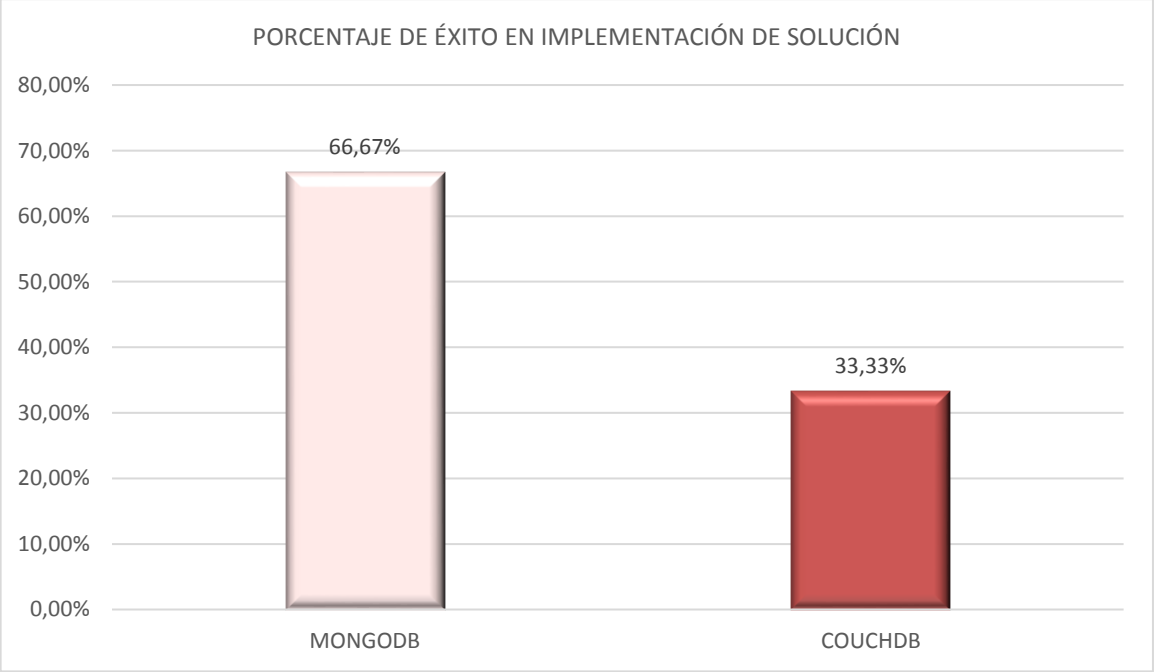



Figura 123. Resultado de evaluación de bases de datos NoSQL en arquitectura de fragmentación
Fuente: Elaboración propia

Las bases de datos documentales, sin duda, es una de la mejores sistemas de almacenamiento NoSQL, con base a las pruebas realizadas es notable el alto rendimiento que proporcionan al gestionar grandes cantidades de datos.

## GLOSARIO

- **NoSQL:** Expresión asociada a las bases de datos para indicar que son sistemas no relacionales.

- **Escalabilidad:** Capacidad que tienen las bases de datos de mantener su calidad y funcionamiento, a pesar del volumen datos.

- **Tolerancia a partición:** Capacidad de las bases de datos de mantenerse siempre disponible a pesar que un servidor falle.

- **Ambiente distribuido:** Conjunto de equipos hardware (*PC's, computadoras*) separados físicamente pero conectadas entre sí y que transfieren información de un lugar a otro.

- **Replicación:** Mecanismo para propagar copias de datos en un ambiente distribuido con el objetivo de tener mejor rendimiento y confiabilidad.

- **Fragmentación:** Mecanismo dividir un conjunto de datos muy grandes en fragmentos de datos más pequeños.

- **Rendimiento:** Capacidad de cálculo de un sistema.

- **Disponibilidad:** Capacidad de un sistema de mantenerse a pesar de un fallo en alguno de los servidores del clúster.

- **Indexación:** Mecanismo crucial para optimizar el rendimiento del sistema y la escalabilidad, mientras proporcionan un acceso flexible a los datos.

- **Tiempo de respuesta:** Es el tiempo necesario para completar una tarea, incluyendo los accesos al disco, a la memoria y las actividades de E/S. Es el tiempo que percibe el usuario.

- **ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad):** Es un conjunto de propiedades de transacciones de bases de datos destinadas a garantizar la validez incluso en caso de errores o fallos de energía

## CONCLUSIONES

En este trabajo se ha comparado distintos puntos de un sistema de bases de datos NoSQL. El objetivo de éste, era encontrar el mejor sistema de base de datos distribuida entre aquellas que actualmente están en el mercado.

- El escalamiento horizontal es sin duda una potente técnica que implica tener varios servidores trabajando como uno sólo para equilibrar de carga de trabajo sobre los diferentes servidores mejorando así el procesamiento de múltiples peticiones. Esta técnica proporciona capacidad para adaptarse a las necesidades de rendimiento a medida que el número de usuarios crece y la información aumenta.

- Los mecanismos de replicación y fragmentación de datos utilizados en el presente trabajo ofrecen grandes beneficios cuando se gestiona grandes volúmenes de información pues aseguran una mejora en el rendimiento al estar replicados o distribuidos los datos en diferentes servidores y aumenta la disponibilidad asegurando que los datos permanecerán disponibles y a salvo en varios servidores cuando pueda ocurra algún fallo.

- Las bases de datos implementadas en este trabajo fueron elegidas en base determinados criterios, principalmente enfocados en la capacidad de que puedan ser implementadas en entornos distribuidos a través de la replicación o fragmentación de datos, y la transparencia que provee en la re-configuración de los servidores del clúster, con lo que garantiza la escalabilidad.

- La configuración del entorno distribuido en CouchDB y Couchbase es muy clara y fácil de comprender, por lo que se requiere de un procedimiento menos complejo que en el caso de MongoDB donde es necesario una investigación profunda de su documentación para comprender con claridad su funcionamiento.

- De las arquitecturas implementadas, se determina que una arquitectura de fragmentación se recomienda cuando se desea equilibrar la carga en servidores que dispongan de pocos recursos (RAM, Disco Duro), y se sugiere la replicación cuando se cuenten con equipos con altos recursos, debido a que mediante la replicación los datos se replican en cada servidor configurado.

- Las arquitecturas de sistema de base de datos NoSQL distribuidas que se presentan en este trabajo son flexibles y puede adaptarse a cambios constantes como el crecimiento de la información, y se presenta como una solución de tolerancia a fallos para hacer frente a fallos de partición de red.

- Mientras se agregan más servidores al entorno distribuido, el rendimiento en la carga de datos y, en la fragmentación y replicación de los datos aumenta, ya que cuando se añade un nuevo servidor, el servidor principal del sistema inicia el proceso de re-balanceo para redistribuir los datos en el entorno distribuido, sin que se afecte el rendimiento global del sistema.

- La comunicación entre las diferentes bases de datos NoSQL aplicadas en este trabajo, no requiere que todas se ejecuten sobre un mismo sistema operativo pues la conexión se realiza a través de los protocolos de red HTTP y TCP. Sin embargo, si se recomienda utilizar la misma versión de software de la base de datos en todo el entorno distribuido.

- La funcionalidad Map/Reduce que proveen las bases de datos CouchDB y Couchbase, están expresamente preparadas para este tipo de entornos, y son capaces de multiplicar el rendimiento del sistema para la rápida obtención de datos.

- Las bases de datos CouchDB y Couchbase son muy parecidas en cuanto a la funcionalidad que ofrecen. Entre ellas, Couchbase es la base de datos que ocupa mayor cantidad de recursos a nivel de hardware (RAM, disco duro), en cambio, MongoDB además de proporcionar una distribución uniforme de los datos, es la base de datos que menos ocupa espacio en el disco.

- En cuanto a la replicación de datos, se considera la base de datos Couchbase como la más apropiada para gestionar grandes volúmenes de datos, y se recomienda utilizarla cuando se disponga servidores con potentes recursos.

- En cuanto a la fragmentación de datos, se considera que MongoDB es la mejor opción al momento de implementar un entrono distribuido de alto rendimiento para la gestión de grandes volúmenes de datos, pues no solo provee tiempos de respuesta más rápidos, si no también proporciona una gran ventaja en cuanto a la facilidad para escalar horizontalmente.

- Este trabajo servirá como referencia para implementar un sistema distribuido de base de datos NoSQL escalable y capaz de responder a las necesidades actuales de empresas u organizaciones, en donde es esencial el aprovechamiento de grandes volúmenes de datos para convertirlos en conocimiento útil.

- Es posible el desarrollo de aplicaciones web de alto rendimiento utilizando bases de datos NoSQL documentales, ya que han demostrado ser mucho más eficientes a la hora de gestionar grandes cantidades de datos.

- El principal beneficio del desarrollo de aplicaciones web utilizando bases de datos NoSQL es la flexibilidad de recuperar los datos pues no definen una estructura como ocurre en las base de datos relacionales y usan JSON como formato de intercambio de información. Y asimismo son aplicables a la mayor parte tecnología existente por lo que se puede construir diferentes aplicaciones web o móviles.

## RECOMENDACIONES

- Se recomienda revisar las especificaciones que provee cada servidor para optimizar la carga de datos en el entorno distribuido. En MongoDB el parámetro -numInsertionsWorkers (mogoimport), en CouchDB el parámetro -parallelism (couchimport), y en Couchbase el parámetro -t (cbtransfer). Todos estos parámetros crean subprocesos simultáneos que consumen recursos del sistema según el valor que sea asignado.

- En MongoDB, para la sincronización exitosa de los servidores de configuración es necesario que la hora de todos los servidores sea la misma en el entorno distribuido, o una diferencia mínima de 10 segundos. Se puede usar el protocolo NTP (Network Time Protocol) para realizar la sincronización automática, o en su defecto cambiar la hora manualmente.

- Se recomienda deshabilitar los cortafuegos (Firewall) de todos los servidores del entorno distribuido, con el fin que no exista problemas de comunicación, o en su defecto gestionar las conexiones con IPTABLES y habilitar las IP's y puertos de cada servidor. Asimismo, todos los servidores deben estar conectados a la misma red para facilitar la sincronización entre ellos.

- Utilice las versiones disponibles de 64 bits en cada base de datos cuando se vaya a realizar su implementación en entornos de producción, debido a que las versiones de 32 bits traen consigo algunas limitaciones. De igual manera utilizar la misma versión de software de base de datos para todos los servidores inmersos en el entorno distribuido.

- Las bases de datos presentadas en este trabajo se presentan como una solución a los desarrolladores de software como una nueva forma de almacenar y gestionar los datos para enfrentar sus proyectos de Big Data. Sin embargo, a pesar que para este caso particular se ha demostrado que los mejores resultados se han logrado con MongoDB en la arquitectura de fragmentación y Couchbase en la arquitectura de replicación, se recomienda analizar la naturaleza para que será utilizada la base de datos, para verificar qué tipo de sistema NoSQL es la más apta para la solución de las problemáticas que se posean.

## BIBLIOGRAFÍA


Anderson, C., Lehnardt, J., & Slater, N. (2010). *CouchDB. The Definitive Guide*. (M. Loukides, Ed.) (1ra ed.). United States of America: O'Reilly Medica Inc.

Barragan Charry, A. M., & Forero Sanabria, A. (2013). *Implementacion de una base de datos Nosql para la generacion de la matriz O/D*. *Journal of Chemical Information and Modeling*. Universidad Católica de Colombia. https://doi.org/10.1017/CBO9781107415324.004

Béjar Heredia, M. de la C. (2015). *Selección, instalación, configuración y administración de los servidores de transferencia de archivos (UF1275).* IC Editorial.

Bender, C., Deco, C., González Sanabria, J., Hallo, M., & Ponce Gallegos, J. C. (2014). *Tópicos Avanzados de Bases de Datos*. (LATIn, Ed.) (Primera Ed).

Brown, M. (2012). Uso de Hadoop con Couchbase, 1–14.

Bsonspec.org. (2013). BSON - Binary JSON. Retrieved January 31, 2017, from http://bsonspec.org/

Camargo-Vega, J. J., Camargo-Ortega, J. F., & Joyanes-Aguilar, L. (2015). Conociendo Big Data. *Revista Facultad de Ingeniería (Fac. Ing.), Enero-Abril*, *24*(38), 63–77. https://doi.org/10.15517/eci.v6i1.19005

Córdova, F., & Cuzco, E. (2013). Análisis comparativo entre bases de datos relacionales con bases de datos no relacionales, 88. Retrieved from http://dspace.ups.edu.ec/bitstream/123456789/6977/1/UPS-CT003639.pdf

Couchbase. (2017). Handling replication. Retrieved June 16, 2017, from https://developer.Couchbase.com/documentation/server/3.x/admin/Tasks/tasks-manage-replication.html

CouchDB. (2017). Sharding — Apache CouchDB 2.0 Documentation. Retrieved April 3, 2017, from http://docs.CouchDB.org/en/2.0.0/cluster/sharding.html

CouchDB Blog. (2016). CouchDB 2.0 Architecture – CouchDB Blog. Retrieved April 4, 2017, from https://blog.CouchDB.org/2016/08/01/CouchDB-2-0-architecture/

Dans, E. (2011). Big Data: una pequeña introducción. Retrieved November 21, 2016, from https://www.enriquedans.com/2011/10/big-data-una-pequena-introduccion.html

DB-engines. (2016). DB-Engines Ranking - popularity ranking of database management systems. Retrieved January 1, 2017, from http://db-engines.com/en/ranking

Faraj, A., Rashid, B., & Shareef, T. (2014). COMPARATIVE STUDY OF RELATIONAL AND

NON RELATIONS DATABASE PERFORMANCES USING ORACLE AND MONGODB SYSTEMS. *International Journal of Computer Engineering and Technology*, *5*(11), 11–22.

Fernández, A., del Río, S., López, V., Bawakid, A., del Jesus, M. J., Benítez, J. M., & Herrera, F. (2014). Big Data with Cloud Computing: An insight on the computing environment, MapReduce, and programming frameworks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *4*(5), 380–409. https://doi.org/10.1002/widm.1134

Gaona, P. A., & Sandoval, E. P. (2013). *Estudio comparativo de modelos de base de datos relacionales y no relacionales y su incidencia de los procesos en sistemas informáticos*. Universidad Nacional de Chimborazo.

García, F. M. (2004). Aplicaciones de la supercomputación. *Manual Formativo de ACTA*, *32*, 73–85.

Guamán, F. O. (2014). *Vulnerabilidades SQL de Bases de Datos Relacionales que podrían reflejarse en Bases de Datos NoSQL*. Universidad Politecnica de Madrid.

Gutierrez, E. (2014). *Aplicación de Persistencia Políglota con Sistemas de Almacenamiento NoSQL*.

Issa, A., & Schiltz, F. (2016). Document Oriented Database, 50.

Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *Computer*, *43*(2), 12–14. https://doi.org/10.1109/MC.2010.58

Mancilla Escobar, S. E. (2013). Uso De Bases De Datos Nosql Documentales Para Crear Sitios Web De Alto Rendimiento.

MongoDB. (2016). Reinventando la gestión de datos - MongoDB. Retrieved January 1, 2017, from https://www.MongoDB.com/es

Moniruzzaman, A. B. M., & Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *International Journal of Database Theory and Application*, *6*(4). Retrieved from http://scholar.google.com/scholar?q=Nosql database: New era of databases for big data analytics-classification, characteristics and comparison&btnG=&hl=en&num=20&as_sdt=0%2C22 VN - readcube.com

Poveda, J. P. (2015). *Propuesta de Notación Gráfica para el Modelo Orientado a Documento de MongoDB*.

Romero, A. C., Sanabria, J. S. G., & Cuervo, M. C. (2012). Utilidad y funcionamiento de las

bases de datos NoSQL. *Revista Facultad de Ingeniería, UPTC*, *21*(33), 21–32. https://doi.org/10.1007/s13398-014-0173-7.2

Silberschatz, A. (Bell L., Korth, H. F. (Bell L., & Sudarshan, S. (Instituto Indio de Tecnología, B. (2002). *Fundamentos de bases de datos*. *Victoria*. https://doi.org/10.1017/CBO9781107415324.004

Sommerville, I. (2005). Ingeniería del software. *Danielr.Obolog.Es*. https://doi.org/http://zeus.inf.ucv.cl/~bcrawford/Modelado%20UML/Ingenieria%20del%20Software%207ma.%20Ed.%20-%20Ian%20Sommerville.pdf

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. *Communication*. https://doi.org/10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C

Vazques, Y., & Sotolongo, A. (2013). Mirada a bases de datos NoSQL de código abierto orientadas a documentos. *Serie Científica de La Universidad de Las Ciencias Informáticas*, *6*(9), 61–67.

w3.org. (2016). Extensible Markup Language. Retrieved January 31, 2017, from https://www.w3.org/XML/

w3schools.com. (2012). JSON - Introduction. Retrieved January 31, 2017, from http://www.w3schools.com/js/js_json_intro.asp

Wallace, M. (2015). Apache: Big Data 2015: CouchDB 2.0: The Awkward Bits - Mike Wal... Retrieved April 4, 2017, from https://apachebigdata2015.sched.com/event/3zu9

Zdnet. (2010). Big Data Analytics. Retrieved January 22, 2017, from http://www.zdnet.com/topic/big-data/

**ANEXOS**

## ANEXO A. Comparativa general de bases de datos NoSQL

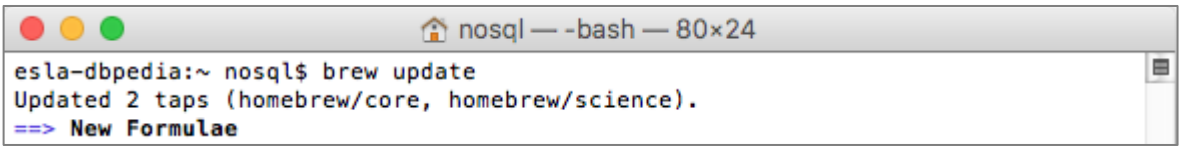
| Nombre | MongoDB | CouchDB | Couchbase | Riak KV | Neo4j |
|---|---|---|---|---|---|
| Ranking | #1<br>Document store | #4<br>Document store | #3<br>Document store | #3<br>Key-value store | #1<br>Graph DBMS |
| Sitio web | www.MongoDB.com | CouchDB.apache.org | www.Couchbase.com | basho.com/products/riak-overview | neo4j.com |
| Soporte | docs.MongoDB.com/manual | wiki.apache.org/CouchDB | developer.Couchbase.com/server | docs.basho.com | neo4j.com/docs/developer-manual/current |
| Desarrollador | MongoDB, Inc | Apache Software Foundation | Couchbase, Inc. | Basho Technologies | Neo Technology |
| Última versión | 3.4.1 | 2.0.0 | 4.5.1 | 2.2.0 | 3.1 |
| Licencia | Libre y comercial | Libre | Libre | Libre y comercial | Libre y comercial |
| Modelo de base de datos | Document store | Document store | Document store | Key-value store | Graph DBMS |
| Lenguaje de implementación | C++ | Erlang | C, C++, Go and Erlang | Erlang | Java |
| Sistemas operativos | Linux, OS X, Solaris, Windows | Android, BSD, Linux, OS X, Solaris, Windows | Linux<br>OS X, Windows | Linux<br>OS X | Linux<br>OS X, Windows |
| Esquema de datos | Libre de esquemas | Libre de esquemas | Libre de esquemas | Libre de esquemas | Libre de esquemas |
| SQL | No | No | - | No | - |
| Métodos de acceso y APIs | Proprietary protocol using JSON | RESTful HTTP/JSON API | Memcached protocol<br>RESTful HTTP API | HTTP API<br>Native Erlang Interface | Cypher query language<br>Java API<br>RESTful HTTP API |
| Lenguaje de programación que soportan | C, C#, C++, Actionscript, Clojure, ColdFusion, Delphi, Erlang, Haskell, Lisp, MatLab, Perl, PHP, Prolog, Python, R, Ruby, Go, Groovy, Scala, Smalltalk, Java, JavaScript | C, C#, ColdFusion, Erlang, Haskell, Lisp, Lua, Objective-C, Ocaml, Perl, PHP, PL/SQL, Python, Ruby, Java JavaScript, Smalltalk | .Net, C, Erlang, Go, Java, Clojure, ColdFusion, JavaScript, Perl, PHP, Python, Ruby, Scala, Tcl | C, C#, C++, Haskell, Java, JavaScript, Lisp, Clojure, Dart, Erlang, Go, Groovy, Perl, PHP, Python, Ruby, Scala,Smalltalk | .Net, Clojure, Go, Groovy, Java, JavaScript, Python, Ruby, Scala, Perl, PHP |
| Escalabilidad horizontal | Si | Si | - | Si | No |
| Método de partición | Sharding | - | Sharding | Sharding | No |
| Método de replicación | Maestro-esclavo | Maestro-maestro<br>Maestro-esclavo | Maestro-maestro<br>Maestro-esclavo | Selectable replication factor | Maestro-esclavo |
| Conceptos de consistencia | Consistencia eventual<br>Consistencia inmediata | Consistencia eventual | Consistencia eventual<br>Consistencia inmediata | Consistencia eventual | Consistencia eventual<br>Consistencia inmediata |
| Capacidades en memoria | Si | No | - | - | - |

| Nombre | Redis | DynamoDB | Cassandra | Accumulo | HBase |
|---|---|---|---|---|---|
| Ranking | #2<br>Key-value store | Key-value store | #1<br>Wide column store | #3<br>Wide column store | #2<br>Wide column store |
| Sitio web | redis.io | aws.amazon.com/dynamodb | cassandra.apache.org | accumulo.apache.org | hbase.apache.org |
| Soporte | redis.io/documentation | aws.amazon.com/documentation/dynamodb | www.datastax.com/docs | - | hbase.apache.org |
| Desarrollador | Salvatore Sanfilippo | Amazon | Apache Software Foundation | Apache Software Foundation | Apache Software Foundation |
| Última versión | 3.2.6 | - | 3.9 | 1.8.0 | 1.2.4 |
| Licencia | Libre y comercial | Comercial | Libre | Libre | Libre |
| Modelo de base de datos | Key-value store | Key-value store | Wide column store | Wide column store | Wide column store |
| Lenguaje de implementación | C | | Java | Java | Java |
| Sistemas operativos | BSD, Linux. OS X, Windows | Multi-plataforma | BSD, Linux, OS X, Windows | Multi-plataforma | Linux, Unix, Windows |
| Esquema de datos | Libre de esquemas | Libre de esquemas | Libre de esquemas | - | Libre de esquemas |
| SQL | No | - | No | - | No |
| Métodos de acceso y APIs | Proprietary protocol | RESTful HTTP API | Proprietary protocol | - | Java API<br>RESTful HTTP API<br>Thrift |
| Lenguaje de programación que soportan | C, C#, C++, Tcl, Python, R, Rebol, Ruby, Rust, Clojure, Crystal, D, Dart, Elixir, Erlang, Fancy, Go, Java, JavaScript, (Node.js), Lisp, Lua, MatLab, Objective-C, Ocaml, Perl, PHP, Prolog, Pure Data, Scala, Scheme, Smalltalk, Haskell, Haxe | .Net, ColdFusion, Erlang, Groovy, Perl, PHP, Python, Ruby, Java, JavaScript<br>Perl | C#, C++, Clojure, Erlang, Go, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby, Scala | NIX 32 entries Operating System | C, C#, C++, Java PHP, Groovy, Python Scala |
| Escalabilidad horizontal | - | Si | Si | Si | Si |
| Método de partición | Sharding | Sharding | Sharding | Sharding | Sharding |
| Método de replicación | Maestro-esclavo | Si | Selectable replication factor | - | Selectable replication factor |
| Conceptos de consistencia | Consistencia eventual | Consistencia eventual<br>Consistencia inmediata | Consistencia eventual<br>Consistencia inmediata | - | Consistencia inmediata |
| Capacidades en memoria | Si | - | - | - | No |

## ANEXO B. Instalación de MongoDB 3.4.4 en Mac OS X

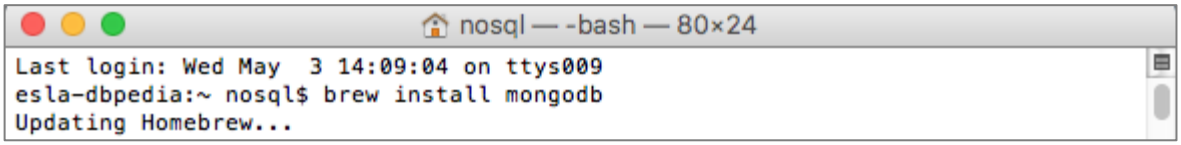
La instalación de MongoDB en Mac Os se realiza a través del gestor de paquetes Homebrew[15].

**Paso 1.** Actualizar el gestor de paquetes Homebrew




**Paso 2.** Instalación

Se ejecuta el siguiente comando para instalar la última versión de MongoDB




Al ejecutar este comando automáticamente se instalan todos los paquetes y dependencias que son requeridas para el correcto funcionamiento de MongoDB.

**Paso 3.** Comprobar versión de MongoDB

Ejecutar el siguiente comando para verificar la versión de MongoDB instalada


[15] Más información en: https://brew.sh/index_es.html

**ANEXO C. Instalación de MongoDB 3.4.4 en Ubuntu 16.04**

**Paso 1.** Importar la clave pública utilizada por el sistema de gestión de paquetes

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
recv 0C49F3730359A14518585931BC711F9BA15703C6
```

**Paso 2.** Crear un archivo de lista de MongoDB

Se ejecuta el siguiente comando para configurar el repositorio de MongoDB en un archivo de lista en Ubuntu 16.04.

```
echo "deb [ arch=amd64,arm64 ]
http://repo.MongoDB.org/apt/ubuntu xenial/MongoDB-org/3.4
multiverse" | sudo tee /etc/apt/sources.list.d/MongoDB-org-
3.4.list
```

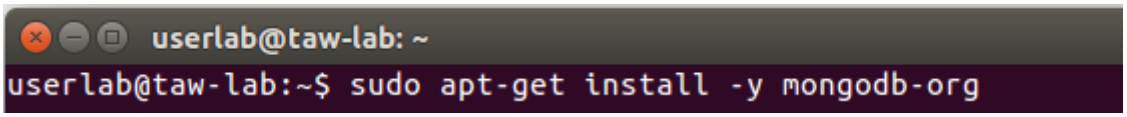
**Paso 3.** Actualizar la base de datos local de paquetes

Para actualizar la base de datos local de los paquetes se debe ejecutar el comando.

```
sudo apt-get update
```

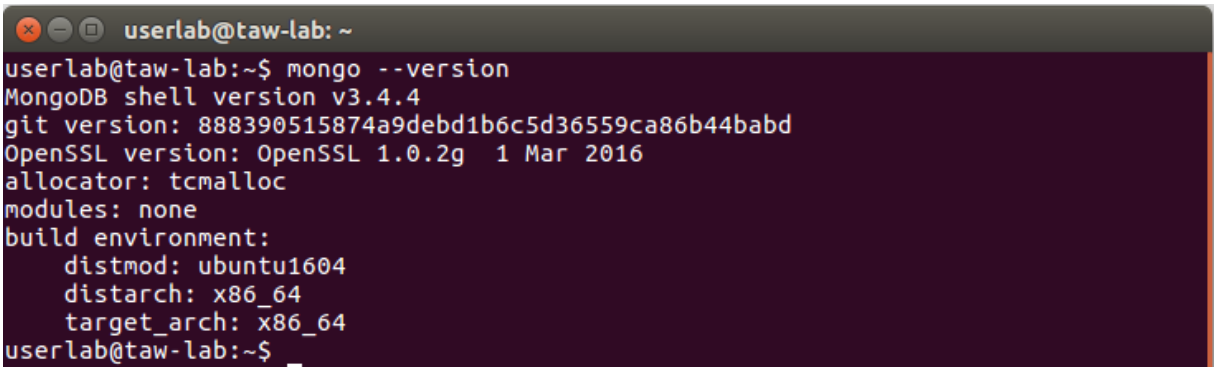
**Paso 4.** Instalación

Se ejecuta el siguiente comando para instalar la última versión de MongoDB.




**Paso 5.** Comprobar versión de MongoDB

Se verifica la versión de MongoDB instalada utilizando el comando:



**Paso 6.** Comando útiles en MongoDB

Inicia proceso mongod

```
sudo service mongod start
```

Detiene proceso mongod

```
sudo service mongod stop
```

**ANEXO D. Instalación de CouchDB 2.0.0 en Mac OS X**

La instalación de CouchDB 2.0.0 en un equipo con sistema operativo Mac OS X, se puede realizar de tres formas:

- Instalación de CouchDB como una aplicación nativa
- Instalación de CouchDB con Homebrew (Todavía en construcción)
- Instalación desde el código fuente

El procedimiento que se detalla a continuación muestra la instalación de CouchDB desde el código fuente.

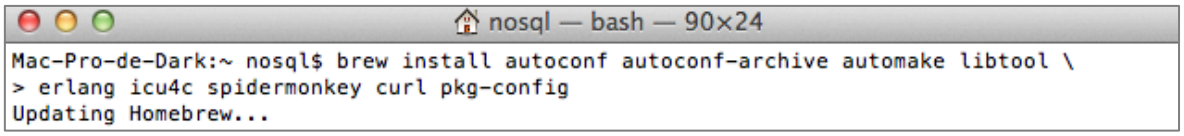
**Paso 1.** Instalación de dependencias

Se debe tener instalado las siguientes dependencias.

- `autoconf`
- `autoconf-archive`
- `automake`
- `libtool`
- `erlang`
- `icu4c`
- `spidermonkey`
- `curl`
- `pkg-config`

Para instalar las dependencias se necesita tener instalado Homebrew[16] para usar el comando `brew`. Utilice el siguiente comando para realizar la instalación de las dependencias.

[16] Más información sobre Homebrew en: https://brew.sh/index_es.html




**Paso 2.** Descargar código fuente de CouchDB

Una vez que se han instalado las dependencias necesarias, el siguiente paso es descargar el código fuente de CouchDB desde el siguiente enlace:

`https://www.apache.org/dyn/closer.lua?path=/CouchDB/source/2.0.0/apache-CouchDB-2.0.0.tar.gz`

Descomprimir en algún directorio el archivo descargado (`.tar.gz`), que contiene el código fuente de CouchDB.

**Paso 3.** Instalación

```
Mac-Pro-de-Dark:~ nosql$ cd Downloads/
Mac-Pro-de-Dark:Downloads nosql$ cd apache-couchdb-2.0.0
Mac-Pro-de-Dark:apache-couchdb-2.0.0 nosql$ ./configure
==> configuring couchdb in rel/couchdb.config
You have configured Apache CouchDB, time to relax. Relax.
Mac-Pro-de-Dark:apache-couchdb-2.0.0 nosql$ make release
```

Para empezar con el proceso de instalación, se debe acceder a la carpeta descomprimida desde un terminal y poner en ejecución el siguiente comando.

```
./configure
```

Si se ejecuta correctamente se mostrará el siguiente mensaje.

```
You have configured Apache CouchDB, time to relax.
```

Luego ejecutar el siguiente comando.

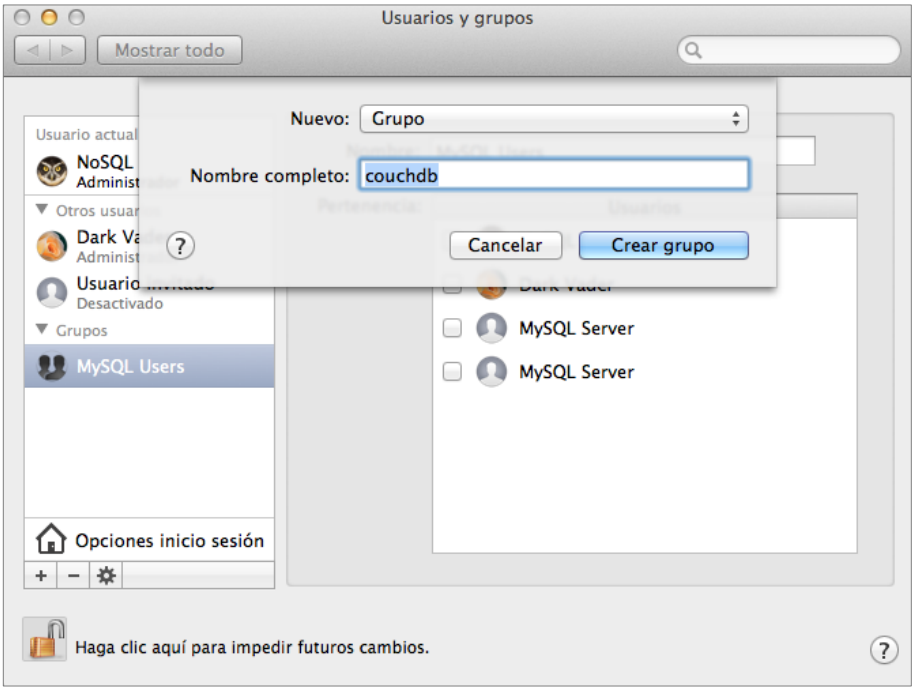
```
make release
```

Si se ejecuta correctamente, se debería mostrar el siguiente mensaje.

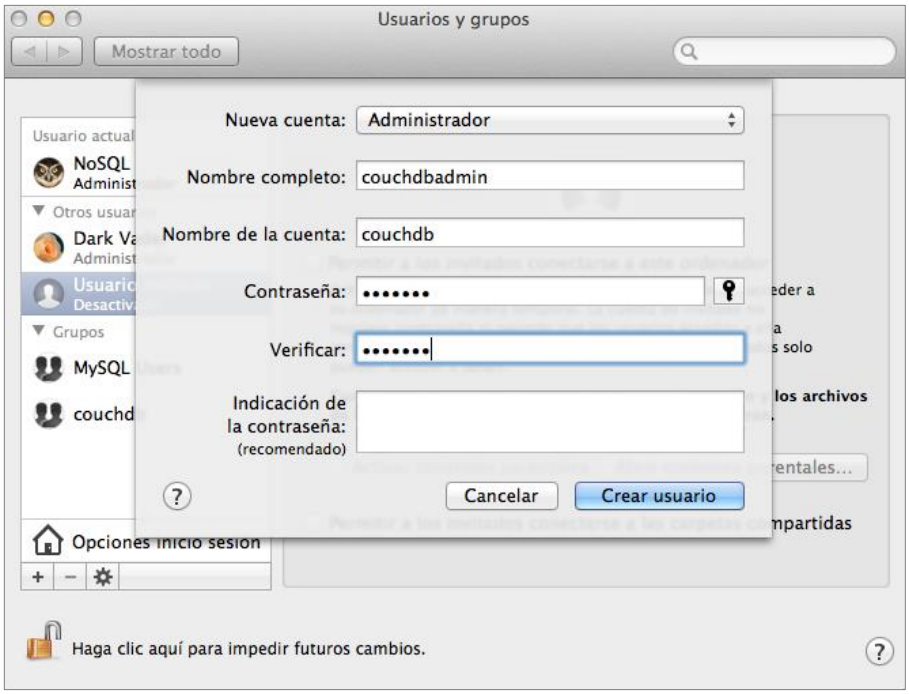
```
... done

    You can now copy the rel/couchdb directory anywhere on your system.
    Start CouchDB with ./bin/couchdb from within that directory.
```

**Paso 4.** Registro de usuario y seguridad

En este paso se debe crear un usuario especial para administrar CouchDB. En Mac OS x se puede crear un usuario desde la interfaz gráfica, específicamente en `Preferencias del Sistema > Usuarios y grupos`.




Una vez creado el grupo CouchDB, se debe crear el usuario administrador para CouchDB. Luego especificar que el usuario creado pertenece al grupo CouchDB.



En Mac OS X cuando se añade un nuevo usuario se crea automáticamente un directorio principal para ese usuario en `/Users/CouchDB`, y es allí donde se debe copiar la liberación (Paso 3). Para finalizar, ejecute los siguientes comandos:

```
Mac-Pro-de-Dark:~ nosql$ sudo cp -R Downloads/apache-couchdb-2.0.0/rel/couchdb /Users/couchdb
Mac-Pro-de-Dark:~ nosql$ sudo chown -R couchdb:couchdb /Users/couchdb/couchdb
Mac-Pro-de-Dark:~ nosql$ sudo find /Users/couchdb/couchdb -type d -exec chmod 777 {} \;
Mac-Pro-de-Dark:~ nosql$ sudo chmod -R 777 /Users/couchdb/couchdb/etc/*
```

Para copiar la liberación de CouchDB construida al directorio del nuevo usuario:

```
cp -R /path/to/CouchDB/rel/CouchDB /Users/CouchDB
```

Para cambiar la propiedad de los directorios CouchDB:

```
chown -R CouchDB:CouchDB /Users/CouchDB/CouchDB
```

Para cambiar los permisos de los directorios CouchDB:

```
find /Users/CouchDB/CouchDB -type d -exec chmod 777 {} \;
```

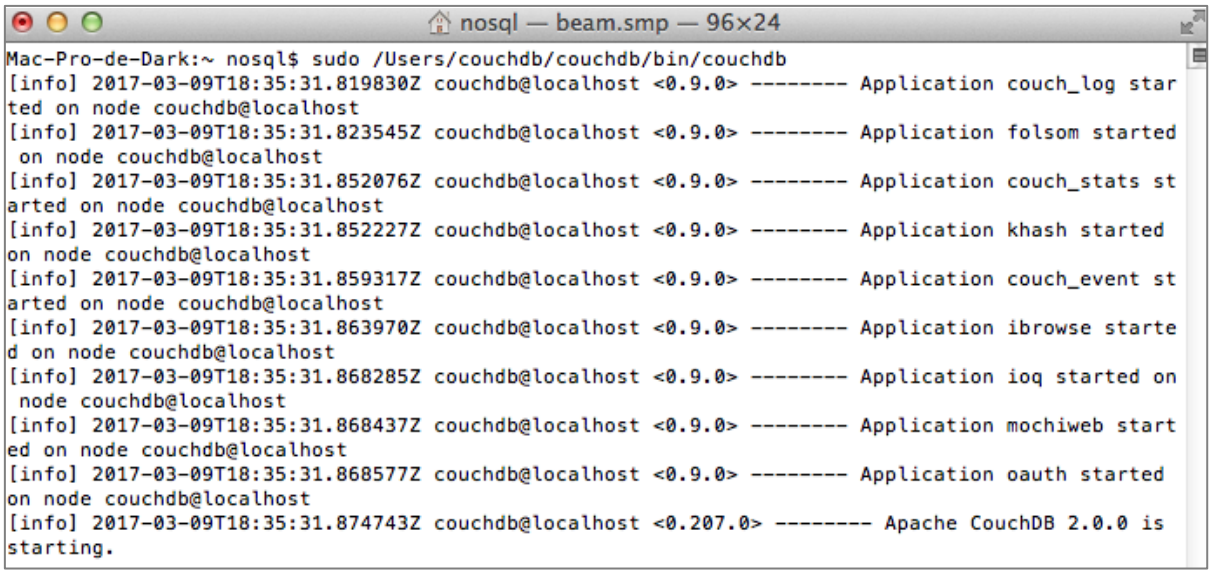
Para cambiar los permisos de los archivos `.ini`

```
chmod 777 /home/CouchDB/CouchDB/etc/*
```

**Paso 5.** Ejecución

Utilice el siguiente comando para ejecutar CouchDB.

```
sudo /Users/CouchDB/CouchDB/bin/CouchDB
```




**Paso 6.** Abrir interfaz administrativa Fauxton de CouchDB

Desde un navegador web acceder a `http://ip-addess:5984/_utils`

**Paso 7.** Configurar usuario administrador en CouchDB

Establecer un usuario administrador que escuche una dirección IP pública `(User: Admin Pass: Admin)`. Para ello, ejecutar los siguientes comandos:

```
curl -X PUT http://Admin:Admin@ip-address:5984/_node/CouchDB@ip-address/_config/admins/Admin -d '"Admin"'

curl -X PUT http://Admin:Admin@ip-address:5984/_node/CouchDB@ip-address/_config/chttpd/bind_address -d '"0.0.0.0"'
```

Si se está configurando CouchDB para varios servidores, ejecutar los mismos comandos para cada servidor únicamente cambiando `CouchDB@ip-address`, por el nombre y la IP del servidor. En todos los servidores utilizar el mismo nombre de usuario y contraseña del administrador.

### ANEXO E. Instalación de CouchDB 2.0.0 en Ubuntu 16.04

El siguiente procedimiento describe la instalación CouchDB 2.0.0 desde el código fuente, en un equipo con sistema operativo Ubuntu 16.04.

**Paso 1.** Instalación de dependencias

Se debe tener instalado las siguientes dependencias.

- `build-essential`

- pkg-config
- erlang
- libicu-dev
- libmozjs185-dev
- libcurl4-openssl-dev

Utilice el siguiente comando para realizar la instalación de las dependencias.

```
userlab@taw-lab:~$ sudo apt-get --no-install-recommends -y install \
> build-essential pkg-config erlang \
> libicu-dev libmozjs185-dev libcurl4-openssl-dev
```

**Paso 2.** Descargar código fuente de CouchDB

Una vez que se han instalado las dependencias necesarias, el siguiente paso es descargar el código fuente de CouchDB desde el siguiente enlace:

`https://www.apache.org/dyn/closer.lua?path=/CouchDB/source/2.0.0/apache-CouchDB-2.0.0.tar.gz`

Descomprimir en algún directorio el archivo descargado (`.tar.gz`), que contiene el código fuente de CouchDB. Para descomprimir por línea de comandos, utilice:

```
userlab@taw-lab:~$ cd Descargas/
userlab@taw-lab:~/Descargas$ tar -xzvf apache-couchdb-2.0.0.tar.gz
```

**Paso 3.** Instalación

```
userlab@taw-lab: ~/Descargas/apache-couchdb-2.0.0
userlab@taw-lab:~/Descargas$ cd apache-couchdb-2.0.0/
userlab@taw-lab:~/Descargas/apache-couchdb-2.0.0$ ./configure
==> configuring couchdb in rel/couchdb.config
You have configured Apache CouchDB, time to relax. Relax.
userlab@taw-lab:~/Descargas/apache-couchdb-2.0.0$ make release
```

Para empezar con el proceso de instalación, se debe acceder a la carpeta descomprimida desde un terminal y poner en ejecución el siguiente comando.

```
./configure
```

Si se ejecuta correctamente se mostrará el siguiente mensaje.

```
You have configured Apache CouchDB, time to relax.
```

Luego ejecutar el siguiente comando.

```
make release
```

Si se ejecuta correctamente, se debería mostrar el siguiente mensaje.

```
... done

    You can now copy the rel/couchdb directory anywhere on your system.
    Start CouchDB with ./bin/couchdb from within that directory.
```

**Paso 4.** Registro de usuario y seguridad

En este paso se debe crear un usuario especial para administrar CouchDB. En Ubuntu se puede crear un usuario utilizando la siguiente línea de comandos.

```
userlab@taw-lab:~$ sudo adduser --system \
>         --no-create-home \
>         --shell /bin/bash \
>         --group --gecos \
>         "couchdbadmin" couchdb
[sudo] password for userlab:
Añadiendo el usuario del sistema `couchdb' (UID 120) ...
Añadiendo un nuevo grupo `couchdb' (GID 128) ...
Añadiendo un nuevo usuario `couchdb' (UID 120) con grupo `couchdb' ...
No se crea el directorio personal `/home/couchdb'.
```

En Ubuntu cuando se añade un nuevo usuario no se crea automáticamente un directorio principal para ese usuario, por ello se debe crear una carpeta llamada CouchDB en `/home/CouchDB`, y es allí copiar la liberación (Paso 3). Para finalizar, ejecute los siguientes comandos:

```
userlab@taw-lab:~$ sudo cp -R /home/userlab/Descargas/apache-couchdb-2.0.0/rel/couchdb /home/couchdb
userlab@taw-lab:~$ sudo chown -R couchdb:couchdb /home/couchdb
userlab@taw-lab:~$ sudo find /home/couchdb -type d -exec chmod 777 {} \;
userlab@taw-lab:~$ sudo chmod -R 777 /home/couchdb/etc/*
```

Para copiar la liberación de CouchDB construida al directorio del nuevo usuario:

```
cp -R /path/to/CouchDB/rel/CouchDB /home/CouchDB
```

Para cambiar la propiedad de los directorios CouchDB:

```
chown -R CouchDB:CouchDB /home/CouchDB
```

Para cambiar los permisos de los directorios CouchDB:

```
find /home/CouchDB -type d -exec chmod 777 {} \;
```

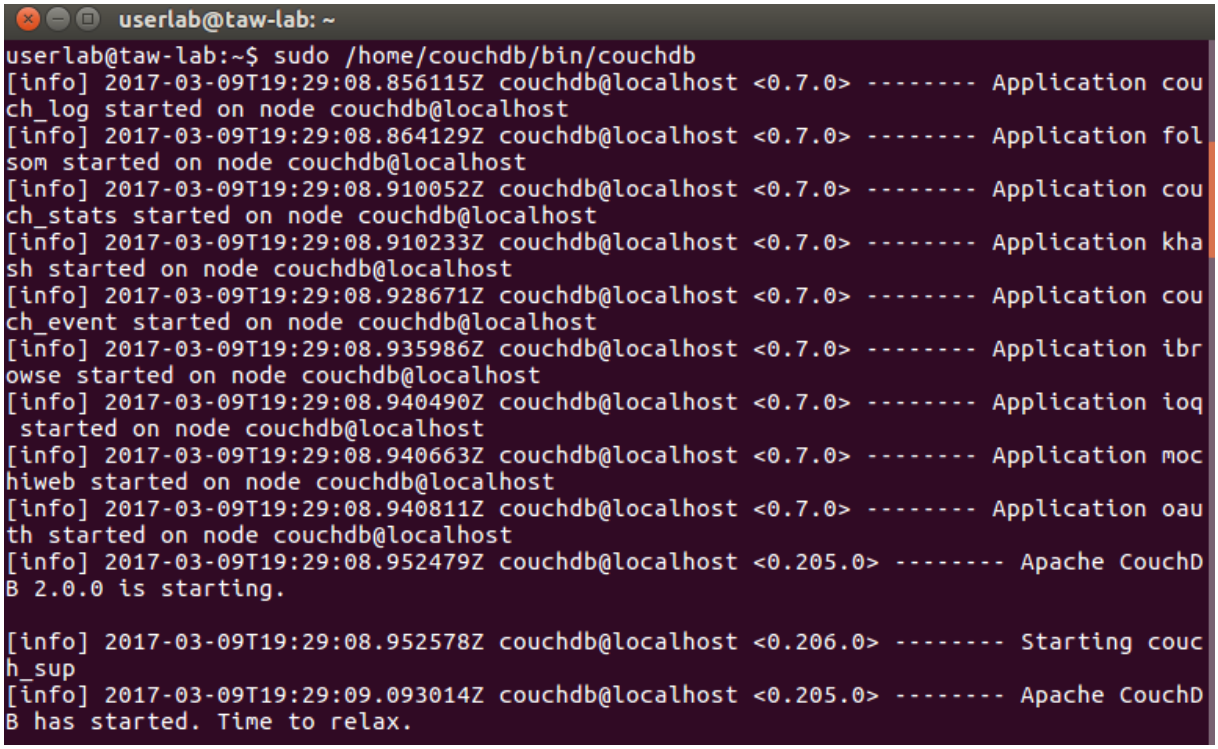
Para cambiar los permisos de los archivos `.ini`

```
chmod 777 /home/CouchDB/etc/*
```

**Paso 5.** Ejecución

Utilice el siguiente comando para ejecutar CouchDB.

```
sudo /home/CouchDB/bin/CouchDB
```




**Paso 6.** Abrir interfaz administrativa Fauxton de CouchDB

Desde un navegador web acceder a `http://ip-addess:5984/_utils`

**Paso 7.** Configurar usuario administrador en CouchDB

Establecer un usuario administrador que escuche una dirección IP pública `(User: Admin Pass: Admin)`. Para ello, ejecutar los siguientes comandos:

```
curl -X PUT http://Admin:Admin@ip-address:5984/_node/CouchDB@ip-address/_config/admins/Admin -d '"Admin"'
```

```
curl -X PUT http://Admin:Admin@ip-address:5984/_node/CouchDB@ip-address/_config/chttpd/bind_address -d '"0.0.0.0"'
```

Si se está configurando CouchDB para varios servidores, ejecutar los mismos comandos para cada servidor únicamente cambiando `CouchDB@ip-address`, por el nombre y la IP del servidor. En todos los servidores utilizar el mismo nombre de usuario y contraseña de administrador.

**ANEXO F. Instalación de Couchbase en Mac OS X**

**Paso 1.** Descargar Couchbase

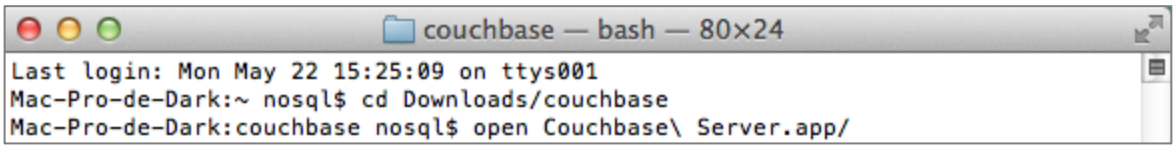
A través del enlace `https://www.Couchbase.com/downloads`, iniciar la descarga de Couchbase seleccionando como plataforma Mac Os X. En caso de haber instalación anteriormente Couchbase, se debe eliminar los archivos restantes de instalaciones anteriores utilizando los comandos:

```
rm -rf ~/Library/Application\ Support/Couchbase
rm -rf ~/Library/Application\ Support/Membase
```

Luego, en algún directorio se debe descomprimir el archivo descargado (`.zip`) que contiene Couchbase.

**Paso 2.** Instalación

Para empezar con el proceso de instalación, se debe acceder desde un terminal a la carpeta descomprimida y poner en ejecución Couchbase, con comando que se muestra a continuación.



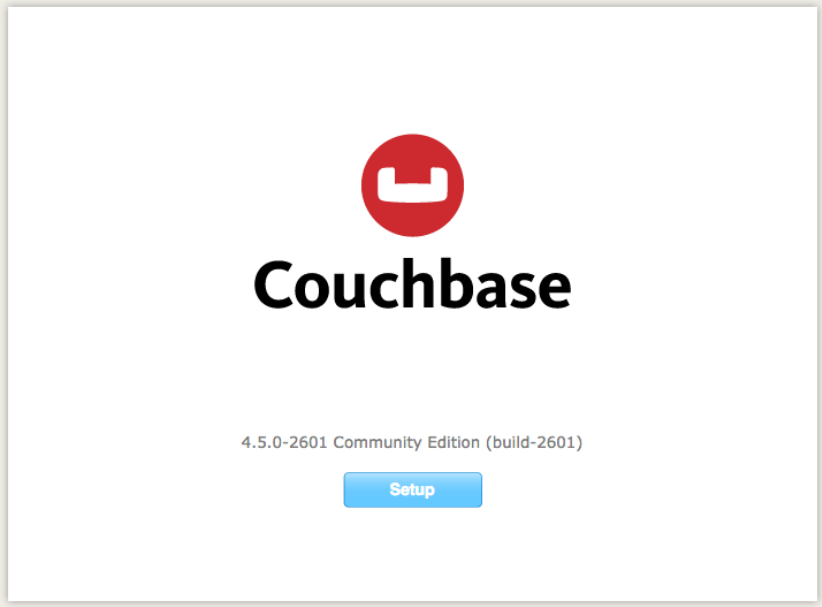

Si se ejecuta correctamente, automáticamente se abrirá la consola web de Couchbase en `http://127.0.0.1:8091/ui/index.html#/`.

Finalmente, para realizar la configuración de un clúster Couchbase se usaron los pasos que se detallan en el ANEXO G.

**ANEXO G. Configuración de un clúster en Couchbase**

Después de la instalación de Couchbase para inicializarlo como un clúster, se puede realizar a través los siguientes servicios: la consola web de Couchbase, la interfaz de línea de comandos, y la API REST de Couchbase. Los pasos detallados a continuación, muestran la configuración de un clúster en Couchbase utilizando la consola web.

La consola web de Couchbase está disponible, por defecto en el puerto `8091`. Por lo tanto, para acceder a la consola web debe accederse a `http://127.0.0.1:8091`. Una vez conectado a la consola web, aparecerá la siguiente pantalla de bienvenida.




Para iniciar la configuración de Couchbase se hace clic sobre SETUP.

**Paso 1.** Inicio de nuevo clúster en Couchbase

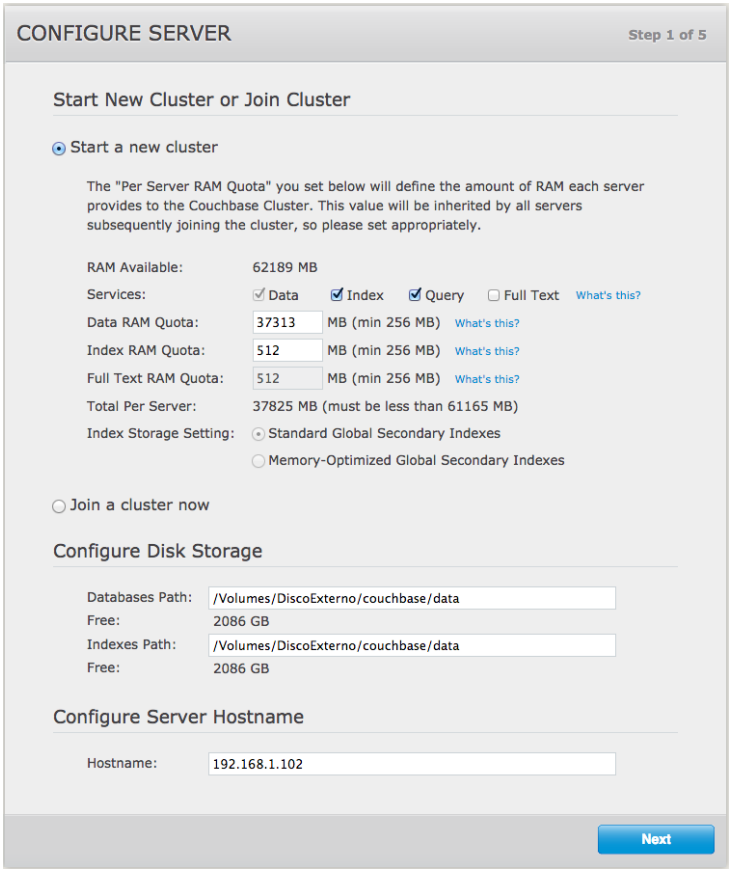
Couchbase permite iniciar un nuevo clúster o unirse como un nodo a un clúster. Para iniciar el nuevo clúster se debe completar la información de los siguientes campos asociados.

- **RAM Available:** Se refiere a la RAM disponible que será usado por el servidor Couchbase
- **Services:** Muestra los servicios de Couchbase disponibles para el servidor; que son datos, índice, consulta y texto completo. Se recomienda de 1 a 3 servicios para un entorno de desarrollo y un sólo servicio para un entorno de producción.
- **Data RAM Quota:** Es la memoria RAM asignada para el servicio de datos.
- **Full Text RAM Quota:** Es la memoria RAM asignada para el servicio de texto completo.
- **Index RAM Quota:** Es la memoria RAM asignada para el servicio de índice.

- **Total Per Server:** Es la memoria RAM total utilizada por el servidor, para un buen rendimiento, éste debe ser inferior que el 80% de la RAM total disponible para ese nodo. La RAM que se especifica aquí también se asignará a cada uno de los otros nodos.
- **Index Storage Setting:** Si se ha seleccionado el servicio de índice, en la versión 4.5.1 de Couchbase, se utilizarán por defecto los índices secundarios globales estándar.

### A. Nuevo clúster

Si se desea iniciar un nuevo clúster, se debe seleccionar la opción *Start a new clúster*.



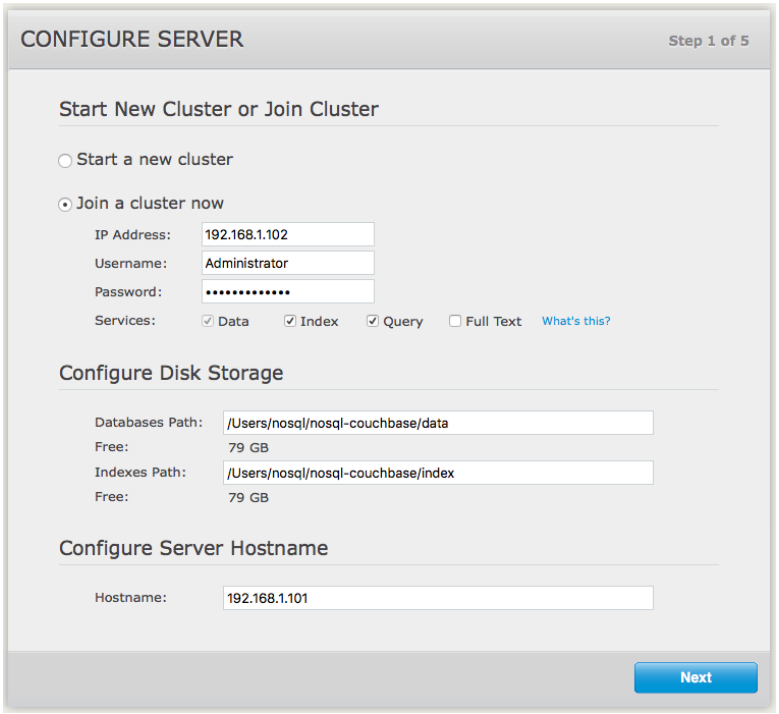

### B. Unirse a un clúster existente

Si se está uniendo a un clúster existente, en lugar de iniciar un clúster nuevo, no es necesario introducir datos en ninguno de los campos anteriores. En su lugar de ello, se selecciona la opción *Join a cluster now*.

Si se elige unirse a un clúster ahora, se deberá completar los siguientes campos:

- **IP Address:** Es la IP del servidor al que se va a unir.
- **Username:** Es el nombre del usuario administrador de Couchbase que se encarga de la administración del clúster al que se está uniendo.

- **Password:** Es la contraseña del usuario administrador de Couchbase que se encarga de la administración el clúster al que se está uniendo.
- **Services available.** Cada uno de los servicios puede seleccionarse mediante una casilla de verificación asociada.




### C. Configuración de almacenamiento en disco

Ya sea que esté iniciando un nuevo clúster o uniéndose a uno existente, es necesario configurar las direcciones de almacenamiento en el disco para la base de datos y los índices.

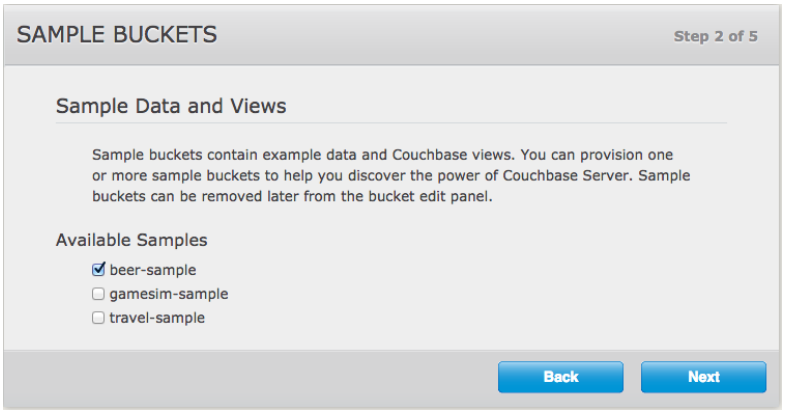
- **Database Path:** Es la ubicación en donde se almacenarán los archivos de la base de datos.
- **Indices Path:** Es la ubicación en donde se almacenarán los índices.

### D. Configuración de IP del servidor

Ya sea que esté iniciando un nuevo clúster o uniéndose a uno existente, es necesario añadir la IP del servidor.

**Paso 2.** Creación de cubos de muestra

Couchbase ofrece cubos de muestra, que contienen datos para fines de demostración y prueba; se debe elegir al menos unos de los cubos de prueba.



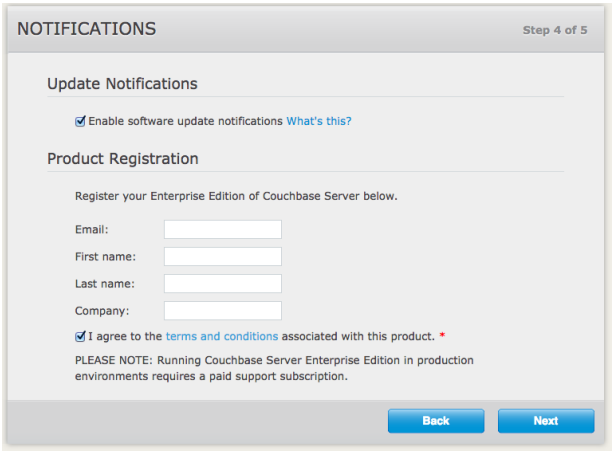

**Paso 3.** Creación de un cubo por defecto

La creación de un cubo por defecto es opcional. El cubo no contiene datos de forma predeterminada; pero puede configurarse en detalle y utilizarse para pruebas y otros fines. La creación de un cubo por defecto proporciona las siguientes opciones:

- **Bucket Settings.** Se establece el nombre y el tipo de cubo. Por defecto el nombre default se ha pre ajustado para el cubo. Sin embargo, el tipo de cubo se debe establecerse como Couchbase.
- **Memory Size.** El campo Per Node RAM Quota permite asignar la memoria RAM al cubo por defecto.
- **Replicas.** Permite activar y administrar la creación de réplicas, se puede establecer valores de 1 a 3 números de réplicas. Asegurarse de activar “View index replicas” para que los índices de vistas y los datos se repliquen.
- **Disk I/O Optimization.** Permite especificar la prioridad de E/S de disco para el cubo. Se puede seleccionar Low (Bajo) o High (Alto), con una prioridad alta se puede obtener un procesamiento más rápido.
- **Flush.** Esta sección permite habilitar flushing. Si está habilitado y se realiza la eliminación del cubo, los elementos dentro del cubo se eliminan tan pronto como sea posible.

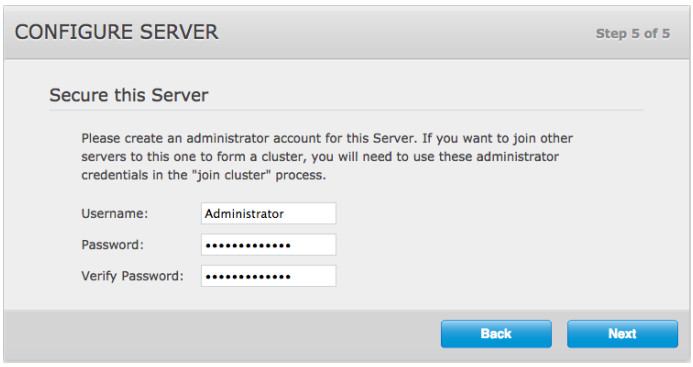

**Paso 4.** Habilitación de notificaciones

La pantalla de notificaciones, permite habilitar las notificaciones de actualización del software y el registro del producto mediante la especificación los datos correo, nombres, apellidos y compañía.



**Paso 5.** Credenciales de administración

Para finalizar la configuración de Couchbase se asignan las credenciales administrativas; estas serán utilizadas en cada uno de los nodos que se unan al clúster.




**Paso 6.** Explora Couchbase

Una vez realizado el Paso 5, el servidor Couchbase ya se encuentra funcionando. El acceso a la consola web de Couchbase debe hacerse a través de `http://ip-server:8091/ui/index.html#/`. Se accede usando las credenciales de administración.