Universidad
Carlos III de Madrid

Departamento de Informática

PROYECTO FIN DE CARRERA

BASES DE DATOS NOSQL: ARQUITECTURA Y EJEMPLOS DE APLICACIÓN

Ingeniería Informática

Autor: Raúl Herranz Gómez

Tutor: Ana María Iglesias Maqueda

Leganés, Junio de 2014

Título: BASES DE DATOS NOSQL: ARQUITECTURA Y EJEMPLOS DE APLICACIÓN
Autor: Raúl Herranz Gómez
Director: Ana María Iglesias Maqueda

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 27 de Junio de 2014 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Hay muchas personas a las que me gustaría agradecer que hayan compartido conmigo muchos momentos importantes, personas que espero y deseo de corazón que me sigan acompañando en muchos momentos más.

Primero a mi familia, quienes me quieren y apoyan en cada paso que doy. Especialmente importante para mi es el apoyo incondicional que mi madre me ha dado y me sigue dando cada día de mi vida. Me cuesta imaginar que sería de mí sin sus consejos y su ayuda. Y a mis hermanos, con los que tantas risas y peleas he tenido, y tanto nos hemos aguantado en casa en los buenos y en los malos momentos.

A Sara, por todo lo que comparto contigo, la confianza, la complicidad, los buenos momentos. También por esa seguridad que me aportas en los momentos clave. Y también por esos largos días de no ver el sol haciendo proyecto y master, sin los cuales no se si podría estar escribiendo esto.

A toda la gente de la carrera que ha compartido conmigo algún momento, y en especial a aquellos que me siguen acompañando en el día a día. Ellos son Martín, Jesús, Aitor, Mario, Marcos, Patarino, Javi, Lalla, Juanto… y mucha más gente de la UC3M.

A mis amigos de toda la vida, de Fuenlabrada y de Collado Hermoso, por haber estado ahí en todo momento, por esas quedadas, por esos veranos, esos viajes y esos fines de semana increíbles.

A Ana, mi tutora, por todo el apoyo y los consejos que me ha dado. Gracias por haberme empujado y animado hasta conseguirlo.

Y finalmente, a todo el mundo que alguna vez me ha preguntado que si ya había terminado el proyecto y que cuando iba a terminarlo. Gracias por preocuparos tanto, ya no tendréis que hacerlo más. Ya está hecho.

Resumen

En los últimos años, la cantidad de datos digitales que se genera el mundo se ha multiplicado. Las redes sociales y el cada vez más fácil acceso a Internet del que disponemos las personas hacen que el volumen de tráfico y de datos que se generan sea cada vez mayor.

Con el surgimiento de las bases de datos relacionales las empresas encontraron el aliado perfecto para cubrir sus necesidades de almacenamiento, disponibilidad, copiado de seguridad y gestión de sus datos.

Pero debido a las tendencias actuales de uso de Internet, este tipo de sistemas han comenzado a experimentar dificultades técnicas, en algunos casos bloqueantes, que impiden el buen funcionamiento de los sistemas de algunas de las empresas más importantes de Internet. Este tipo de datos que son masivamente generados reciben un nombre: BigData, y un tipo de tecnología ha surgido para tratar de poner solución a muchos de los problemas de los que adolecen los sistemas de almacenamiento tradicionales cuando intentan manejar este tipo de datos masivos. Esta tecnología se conoce como NoSQL.

En este Proyecto Fin de Carrera se profundizará en el movimiento conocido como NoSQL, analizando las características y tipos de bases de datos que han surgido como parte del movimiento. Además, se detallará la arquitectura y el modo de funcionar de seis de estos Sistemas Gestores de Bases de Datos, comparándolos entre sí y buscando en qué casos funcionan mejor unos y otros.

Finalmente, se describirá cómo instalar de dos de estos sistemas con sus configuraciones más sencillas, así como un posible diseño de base de datos para un conjunto de información concreta.

Índice General

Índice de Ilustraciones

Índice de Tablas

1 Introducción

En esta sección se explicará, en primer lugar, las razones que han motivado el desarrollo de este Proyecto Fin de Carrera. Después se expondrán los objetivos que se persiguen a lo largo del documento, y finalmente, se dará una visión general sobre cómo se ha decidido estructurar el trabajo.

1.1 Motivación

En los últimos años se ha experimentado un increíble crecimiento de las necesidades relativas a almacenar digitalmente todo tipo de información que generamos las personas como consecuencia de nuestra actividad, tanto en el ámbito personal como profesional.

Desde los inicios de la informática, el campo de las bases de datos ha sido un frente en constante estudio, cambio e innovación por parte de muchas personas y grupos de desarrolladores. Los Sistemas Gestores de Bases de Datos (en adelante, "SGBD"), se han convertido en la pieza fundamental del manejo y administración de los datos que se generan y almacenan en cualquier sistema de información actual.

Desde el nacimiento de los SGBD relacionales, se han ido cubriendo las necesidades de almacenamiento para los proyectos informáticos según han ido surgiendo los problemas. Las múltiples alternativas que, tanto libres como de pago se pueden encontrar en el mercado, aseguraban que las necesidades de almacenamiento, disponibilidad y consistencia de los datos sensibles de las empresas quedarían perfectamente cubiertos.

Pero desde el nacimiento de internet, y más concretamente, en la última década, con la explosión de las redes sociales, el volumen y la variedad de información que se genera ha puesto de manifiesto unas necesidades de almacenamiento y procesamiento de la información que no se habían tenido hasta ahora.

El acceso a internet, cada vez más rápido, barato y fácil por parte de la sociedad, la variedad de programas y aplicaciones, tales como redes sociales, páginas de noticias e información, blogs, etc., han hecho crecer exponencialmente el volumen de información que las personas generamos como fruto de nuestra actividad en internet.

Esta generación masiva de información ha sacado a relucir ciertos problemas que experimentan los SGBD tradicionales cuando se les somete a ciertas situaciones: lentitud de accesos, problemas de bloqueos, dificultad para mantener bases de datos distribuidas geográficamente, necesidades avanzadas de particionado, etc.

Las grandes empresas en internet, como Google, Facebook, Amazon, Microsoft, etc., han sido las primeras en toparse con estas limitaciones, y han sido también las primeras en comenzar a desarrollar y liberar al mundo las primeras soluciones en forma de bases de datos que surgen como alternativas a los sistemas tradicionales de almacenamiento.

A lo largo de este trabajo se tratará de explicar el origen y las fuentes de este tipo nuevo de información, los sistemas que existían hasta ahora para hacer frente a las necesidades de almacenamiento de estos datos, los problemas que experimentan, y los nuevos SGBDs que han surgido para hacer frente a las necesidades actuales.

1.2 Objetivos

El presente documento se ha realizado persiguiendo una serie de objetivos:

- Dar una visión de cómo era hasta hace unos años las necesidades de almacenamiento de datos que se tenía, así como de los sistemas SGBD que existían para cubrir estas necesidades.
- Analizar el tipo de datos que generamos actualmente las personas como consecuencia de nuestra actividad personal y profesional tanto en internet como fuera de ella.
- Concretamente, hablando de los sistemas de almacenamiento de datos, analizar qué soluciones han existido hasta hace relativamente poco tiempo. Más o menos, desde hace unos 5-6 años hacia atrás.
- Introducción a los sistemas de bases de datos NoSQL. Qué tipo de sistemas han surgido recientemente para tratar de cubrir las carencias que tienen los sistemas tradicionales en ciertos aspectos de diseño, o bajo ciertas condiciones de funcionamiento.
- Realizar una comparativa de todos estos nuevos sistemas gestores de base de datos que nos permita sacar los puntos fuertes y débiles de cada uno de ellos.
- Dado un conjunto de datos, realizar un diseño de base de datos en dos de estos nuevos sistemas. El objetivo de esto es comprobar que, efectivamente, mediante estas nuevas bases de datos se puede ser capaz de generar un modelo de datos con el que poder interactuar desde aplicaciones externas.

1.3 Estructura del documento

Este documento se estructura en distintos apartados donde se irán abordando cada uno de los aspectos claves del presente estudio.

- **Capítulo 1: Introducción**

En esta sección se expondrán las principales motivaciones que se han tenido en cuenta para la realización de este estudio. Además, se explican los objetivos perseguidos por la presente memoria, así como una visión general de los apartados desarrollados en la misma.

- **Capítulo 2: Estado del Arte**

En este capítulo se trata de poner en perspectiva el término BigData. Se verán definiciones, cuándo surge, en qué estado actual se encuentra, así como lo forma en la que afecta a los sistemas de almacenamiento de datos.

Además, se realiza un repaso de los tipos de sistemas gestores de base de datos que se pueden emplear para almacenar grandes cantidades de información.

- **Capítulo 3: NoSQL**

Estudio en profundidad del movimiento conocido como NoSQL. Aquí se analizarán los aspectos más importantes, las características que son comunes a varias de las bases de datos NoSQL, y en qué tecnología se basan. También se analizará cuáles son las críticas que se hacen a estas bases de datos.

- **Capítulo 4: Bases de Datos NoSQL Clave – Valor**

En el capítulo 4 se verá en mayor profundidad el primero tipo específico de base de datos NoSQL: las bases de datos clave-valor. Aquí se incluirán detalles de implementación y funcionamiento de dos de las bases de datos Clave-Valor más representativas, con sus características funcionales y particularidades.

- **Capítulo 5: Bases de Datos NoSQL Documentales**

En este capítulo se expondrán las principales características de las BBDD Documentales, así como el funcionamiento detallado de dos de las bases de datos de tipo documental más importantes y más utilizadas por la comunidad.

- **Capítulo 6: Bases de Datos NoSQL basadas en columnas**

Las bases de datos basadas en columnas son otro tipo de base de datos NoSQL, cuyas principales características se detallarán en este apartado. Así mismo, se profundizará en el funcionamiento y características de la que probablemente sea la base de datos más importante de este tipo.

- **Capítulo 7: Bases de Datos NoSQL Orientadas a Grafos**

El último tipo de base de datos NoSQL que se analizará en profundidad en este estudio serán las bases de datos orientadas a grafo, que son las más particulares dentro de los muchos tipos de base de datos NoSQL que existen.

- **Capítulo 8: Discusión final sobre los sistemas vistos.**

En el capítulo 8 se debatirá sobre los puntos fuertes y flacos de cada sistema, y se determinará en qué escenarios es conveniente y cuándo no utilizar cada uno de ellos.

- **Capítulo 9: Conclusiones y Trabajos Futuros**

En este capítulo final se comentan las conclusiones que se extraen tras la realización del Proyecto Fin de Carrera. También cuales podrían ser las líneas que aun permanecen abiertas, así como los siguientes desarrollos y tecnologías que pueden suceder o completar de alguna forma a las tecnologías descritas en el presente documento.

- **Siglas y acrónimos**

En este apartado se hace un repaso de términos y siglas empleados en el documento.

- **Anexo I: Instalación de Hadoop y HBase**

Aquí se muestra cómo instalar la versión mantenida por la comunidad de Hadoop, así como, una vez instalado, montar y utilizar HBase encima de Hadoop.

- **Anexo II: Descripción de los tweets que conforman el corpus**

En este anexo se describen los tweets que han utilizado para modelar el diseño de base de datos que se abordará en las siguientes secciones del documento.

- **Anexo III: Diseño de base de datos en HBase**

En este apartado se utilizará un conjunto de datos con información real para realizar un diseño de base de datos en HBase.

- **Anexo IV: Diseño de base de datos en MongoDB**

El mismo corpus de información que se ha utilizado en el apartado anterior se empleará en éste para realizar un diseño de base de datos en MongoDB.

2 Estado del arte

En el estado del arte se analizará qué es el Big Data, cómo surge, y qué características tiene. Además, se verá qué bases de datos existían hasta ahora para hacer frente a las necesidades de almacenamiento actuales, y si estas bases de datos, unidas al Big Data, presentan algún problema. Finalmente, se hará un repaso general de las distintas soluciones NoSQL que han surgido en los últimos años para tratar de cubrir los distintos tipos de necesidades.

2.1 Big Data

Definir qué es el Big Data no es tarea fácil. A lo largo y ancho de Internet y de la literatura relacionada que se puede consultar, se puede encontrar multitud de definiciones, diversos puntos de vista que encajan en mayor o menor medida con el concepto de Big Data.

En esta sección se intentará exponer de forma general una visión sobre qué es el Big Data, se detallarán los distintos conceptos que hacen de esta tecnología algo tan importante hoy en día y se someterá a análisis diferentes herramientas que toman parte importante en el Big Data tal y cómo lo conocemos hoy en día.

2.1.1 ¿Qué es Big Data?

Diversas personalidades con autoridad en el mundo de Internet han plasmado su opinión respecto a este tema.

Ed Dumbill forma parte del staff de O'Really Media, la conocida marca editorial que publica multitud de libros técnicos al año, y dice esto en el libro Big Data Now:

"*Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it*." (O'Really 2012)

En el blog oficial de Microsoft Enterprise se puede leer:

"*Big data is the term increasingly used to describe the process of applying serious computing power – the latest in machine learning and artificial intelligence – to seriously massive and often highly complex sets of information*" (HowieT 2013)

Adrian Merv, de la revista Teradata expone la siguiente definición:

"*Big data exceeds the reach of commonly used hardware environments and software tools to capture, manage and process it within a tolerable elapsed time for its user population*" (Merv 2011)

El instituto *McKinsey Global Institute* también plasmó en un artículo su opinión acerca de lo que el Big Data significa:

"*Big data refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze*" (Manyika and Chui 2011)

En el artículo "Bayes and Big Data: The Consensus Monte Carlo Algorithm" se expone la siguiente definición:

"*A useful definition of big data is data that is too big to comfortably process on a single machine, either because of processor, memory, or disk bottlenecks*" (Steven L. Scott 2013)

Esto es tan solo una pequeña parte de la cantidad de definiciones y de autores que se puede encontrar en la red.

Atendiendo a estas definiciones, se podría hacer un resumen, a grandes rasgos y proponer nuestra propia definición sobre lo que el Big Data significa:

Big Data es la ingente cantidad de información, en su mayor parte desestructurada, que hoy en día generamos toda la sociedad como consecuencia de nuestra actividad tanto en Internet como fuera de ella.

Además, debido a cantidad de datos de los que se está tratando, se necesita un nuevo y diferente enfoque a los que existen hasta ahora en lo que a la captura, almacenamiento y procesado de toda esta información se refiere. Esta es la única manera de que se pueda gestionar eficientemente toda esta información. Las bases de datos tradicionales existentes se quedan cortas, se necesitan nuevas herramientas que ayuden a llevar a cabo todo lo expuesto anteriormente

Ilustración 1. Big Data Cloud Tags[1]

[1] Figura extraída de https://www.academia.edu/6033574/BIG_DATA_GOOD_DATA.

2.1.2 ¿Para que sirve el Big Data?

El valor que el Big Data proporciona tiene que ver con los datos que se pueden extraer de esta gran cantidad de información. Pero, ¿de dónde sale toda esa información?, ¿tanta información se está generando realmente?

A continuación se exponen algunos ejemplos prácticos sobre donde se podría aplicar métodos y técnicas de recolección y procesamiento de grandes cantidades de datos (BBVA 2013):

- Más de 5.000 millones de personas poseen teléfonos móviles.
- Facebook tiene más de 900 millones de usuarios que acceden activamente a la aplicación.
- Cada día se mandan 340 millones de tweets, lo que corresponde a una media de 4.000 por segundo.
- Al día se generan 2,5 trillones de datos. El 90% de los datos que existen se han generado en los últimos 2 años.
- En el mundo se efectúan 10.000 pagos con tarjeta de crédito por segundo.

Por minuto…:

- …se envían 204 millones de emails.
- …se suben 3.000 fotos nuevas y se visualizan 20 millones.
- …se escuchan 61.141 horas de música
- …se descargan 47.000 aplicaciones móviles.
- …se visitan 6 millones de perfiles de Facebook.
- …se suben 30 horas de vídeo y se generan 1,3 millones de visualizaciones.

En líneas generales, el objetivo que persiguen las tecnologías Big Data es el de tratar de sacar el verdadero valor que guardan los datos. Esto debe lograrse tanto a través del análisis de los datos, como con la creación de nuevos modelos de negocio válidos para mantener viva este tipo de tecnología.

Aunque actualmente la comunidad se está centrando mucho en la tecnología, hay que tener presente que es necesario encontrar la forma de dar valor a los datos en forma de nuevos modelos de negocio, o de modelos de negocio adaptando los ya existentes.

2.1.3 Características del Big Data : las 3 V's

Las 3 Vs se han tomado en el universo del Big Data como unas propiedades características de esta tecnología. Las 3 Vs hacen referencia a la cantidad de información, la variedad de fuentes y formatos, y la velocidad que se necesita para manipular, almacenar y procesar la información (Soubra 2012).

La imagen que se muestra a continuación expone datos y medidas alrededor de estas definiciones que se han tomado ya como parte de la tecnología y de las características básicas del movimiento.

- Volumen

La cantidad de datos que procesan ciertas empresas ha crecido enormemente. Los sistemas tradicionales de bases de datos comienzan a tener problemas de rendimiento debido al alto volumen de información.

Por ejemplo, Google procesa 20 petabytes diariamente, y Twitter genera 8 terabytes al día.

- Variedad

Uno de los principales problemas que tiene almacenar toda la actividad que se genera es que toda esa información se encuentra principalmente desestructurada. Audio, vídeo, XML… datos de sensores, cotizaciones bursátiles, datos en streaming… Toda esta variedad en las fuentes "obliga" en cierta forma el hecho de que haya que almacenar la información sin ningún tipo de preprocesado o estructura, complicando en gran medida el posterior procesamiento y análisis general de los datos.

- Velocidad

La velocidad de procesamiento se torna de vital importancia. La captura, almacenamiento y procesado de información puede llegar a tardar días en muchos modelos de información antiguos. El Big Data intenta ser una solución a estos tiempos tan altos, mejorando los tiempos de almacenamiento y procesado para conseguir un sistema que responda "casi" en tiempo real a la demanda.

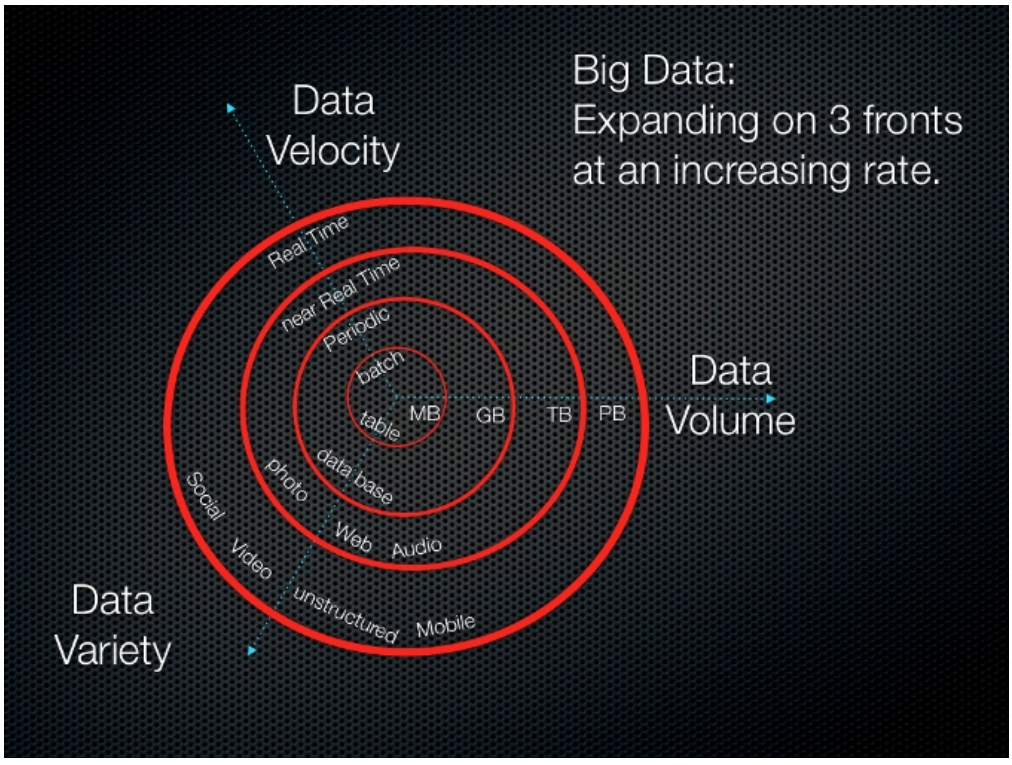

Ilustración 2. Las 3 Vs del BigData[2]

[2] Imagen extraída del artículo http://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data

Si algo se repite cada vez que alguien realiza un ejercicio comparativo entre las nuevas tendencias (Big Data, NoSQL) y los sistemas y bases de datos más tradicionales, y que es importante tener en cuenta, es que las tecnologías relacionadas con el Big Data son unas soluciones que atienden a una serie de problemas muy concretos, y por lo tanto, sería un error pretender generalizar en ellas.

Cada problema ha de ser analizado y comprendido antes de poder darle una solución. Dar el salto automáticamente y proponer una tecnología de este tipo para abordar cualquier tipo de problema puede costarnos muy caro el día de mañana.

Hay que tener claro que **el Big Data es una tecnología en constante cambio** y crecimiento. De igual forma, el mercado y sus necesidades también se encuentran en continuo cambio, por lo que es necesario mantener un proceso de actualización para no quedarse atrás.

No existe la gallina de los huevos de oro en este campo. Si estás tratando de solucionar un problema con una tecnología Big Data, tendrás primero que analizar en profundidad tu problema para descubrir las necesidades que se te están planteando, y a continuación, bucear en el enorme mercado de soluciones que se ofrecen para cubrir el amplio espectro de posibilidades que existen.

A continuación se analizará más en profundidad, como ejemplo de ello, uno de los sistemas más utilizados hoy en día para almacenar y procesas estas grandes cantidades de datos, viendo en profundidad su arquitectura y modo de funcionar.

2.1.4 Un ejemplo de sistema de almacenamiento para Big Data: Hadoop

Apache Hadoop es un proyecto *open source* que está acogido como proyecto de primer nivel en la Apache Software Foundation.

Hadoop es una herramienta que permite desarrollar aplicaciones distribuidas, proporcionando una capa entre el sistema operativo y dichas aplicaciones distribuidas de forma que se tenga un visión de conjunto sobre los recursos de todas las máquinas. Como se verá posteriormente en este mismo apartado. Hadoop se compone de dos elementos que proporcionan un servicio de sistema de ficheros distribuido y redundante, así como un paradigma de procesamiento de los datos que se hayan volcado previamente a dicho sistema de ficheros.

Además, existen diferentes herramientas desarrolladas por terceros (muchas de ellas liberadas como *open source* y otras con objetivos comerciales) orientadas a ofrecer servicios aprovechando las características principales de Hadoop. Entre estas herramientas se pueden ver bases de datos, librerías de aprendizaje automático (*machine learning*), lenguajes de scripting, servicios de coordinación de configuraciones, etc.

Debido al creciente interés de las empresas y de la comunidad en general por las aplicaciones distribuidas en los últimos años, Hadoop se ha convertido rápidamente en uno de los proyectos más activos y que más interés genera en la comunidad.

Hadoop está escrito en Java, y está diseñado para escalar desde un único nodo hasta clústers (conjuntos de máquinas que funcionan como una sola compartiendo los recursos) del orden de miles de máquinas, cada una de ellas compartiendo capacidad de cómputo y almacenamiento.

Esta parte del documento, así como las imágenes han sido obtenidas directamente de la web oficial del proyecto, mantenido por la Apache Software Foundation (Apache Software Foundation 2014).

2.1.4.1 Componentes

El proyecto se subdivide en varios módulos, a saber:

- Hadoop Common.

Este módulo contiene una serie de librerías comunes que permiten la integración del resto de módulos. Entre otras, proporciona librerías que permiten acceso a los sistemas de archivos soportados por Hadoop. También contiene código fuente y documentación acerca del proyecto.

- Hadoop Distributed File System (*HDFS*).

Un sistema de ficheros distribuido, escalable y portátil que Hadoop utiliza para leer y escribir la información. Se trata del sistema de ficheros propio de Hadoop, diseñado para ser tolerante a fallos y que se ajustara a las necesidades derivadas del manejo de enormes cantidades de información.

Más adelante se profundizará más en los detalles de diseño e implementación de este sistema de ficheros.

- Hadoop MapReduce

Hadoop *MapReduce* es un entorno de trabajo (*framework*) que permite escribir aplicaciones sencillas que puedan procesar grandes cantidades de información, del orden de muchos gigabytes o terabytes.

La forma que tiene de realizar este trabajo es, de forma similar al *HDFS*, apoyarse en el clúster tan grande que ofrece Hadoop, y dividir las los trabajos (*Jobs*) en tareas más pequeñas (*Tasks*), que puedan ser repartidas a lo largo del clúster. Para ello, utiliza 2 procesos independientes que gestionan cada una de estas tareas, el *JobTracker* y el *TaskTracker*

Más adelante se le dedicará un apartado para profundizar en el diseño y en la forma de trabajar de *MapReduce*.

- Hadoop *YARN*

YARN ha llegado al ecosistema Hadoop para sustituir a *MapReduce* a partir de la versión 2.0 de la plataforma.

YARN es una revisión de *MapReduce*, que enfoca de manera diferente ciertos aspectos que éste implementa. Así, la idea fundamental de Hadoop *YARN* es la de dividir las 2 funcionalidades principales del *JobTracker*: la gestión de recursos y la programación, y la monitorización de trabajos, creando con esto 2 procesos independientes.

La arquitectura pasaría por tener un *ResourceManager* global y un *ApplicationMaster* por cada una de las aplicaciones, entendiendo "aplicación" como un trabajo del antiguo *MapReduce*.

2.1.4.2 HDFS

El sistema de ficheros *HDFS* es el principal sistema de ficheros que utilizan las aplicaciones de Hadoop para almacenar y recuperar la información.

Como se verá más adelante, *HDFS* gestiona el almacenamiento de varias máquinas que se comunican entre sí formando parte de una misma red. Esto es lo que comúnmente se conoce como sistema de ficheros distribuido.

La idea original de la que se partió para crear un sistema de ficheros escalable, robusto y redundante proviene de la arquitectura que presentó Google en 2004 en uno de sus artículos más famosos y consultados desde entonces (Ghemawat, Gobioff and Leung 2003).

2.1.4.2.1 Diseño

HDFS está diseñado para almacenar ficheros muy grandes (idealmente del orden de gigas o teras) y está pensado para ejecutarse en servidores estándar, sin necesitar una inversión elevada en máquinas con discos rápidos o duraderos, ni con una CPU especialmente potente o con más módulos centrales (*core*) de lo habitual.

Esto es así porque la posibilidad de que un nodo falle de entre todos los del clúster es bastante elevada cuando el número de máquinas que pertenecen al clúster es grande.

De igual manera, existen ciertos casos de uso en los que los clústers de Hadoop no se comportan de manera óptima. Hoy en día, estos son los casos en los que es bueno revisar si se está haciendo un buen uso del sistema:

- **Acceso a datos de baja latencia**. Las aplicaciones que requieren acceso a datos con muy poca latencia, del orden de decenas de milisegundos, no funcionarán bien con *HDFS*. *HDFS* está optimizado para ofrecer alto rendimiento en la entrega de datos, y esto es a costa de la latencia.
- **Multitud de archivos pequeños**. Debido a que el *NameNode* mantiene los metadatos del sistema en memoria, el límite de archivos que éste puede manejar es directamente proporcional a la memoria que tenga este servicio disponible. El tamaño habitual de cada fichero, bloque y directorio es de unos 150 bytes. De este modo, si por ejemplo se tienen un millón de archivos, cada uno de ellos ocupando un bloque, se está ocupando 300 megas de memoria. El coste de almacenar millones de ficheros puede que no sea algo demasiado descabellado, pero la cosa se complica cuando se empieza a tratar con miles de millones o incluso de billones de archivos.
- **Múltiples escrituras y escrituras en posiciones aleatorias del fichero**. Los archivos en *HDFS* deben ser escritos por un único proceso de escritura, y deben ser siempre efectuadas al final del fichero.

Un **bloque**, en el ámbito de los discos duros tradicionales, representa la cantidad mínima de datos que el disco puede leer o escribir. Generalmente los bloques son de varios bytes, o incluso unos pocos kilobytes.

HDFS toma el mismo concepto de bloque, pero utiliza en cambio un tamaño mucho mayor: 64 megabytes por defecto, aunque como muchos otros puede ser parametrizable.

Abstraer y dividir los ficheros a nivel de bloque tiene varias ventajas. Una de ellas es poder contar con ficheros de tamaños más grandes que cualquier disco duro del clúster. Al poder repartir los diferentes bloques de un fichero por distintos discos duros el tamaño del fichero podría ser incluso el de la suma de todos los discos duros del clúster.

Además, hacer las cosas lo más sencillas posible es en general una buena práctica, y más cuando se está tratando un sistema distribuido, donde se complica en gran medida el hacer debug y hacer seguimiento de los errores. Es por eso que la simplificación que se obtiene al dividir los ficheros en conjuntos de "trozos" más pequeños y de igual tamaño parece ser, a priori, una buena idea.

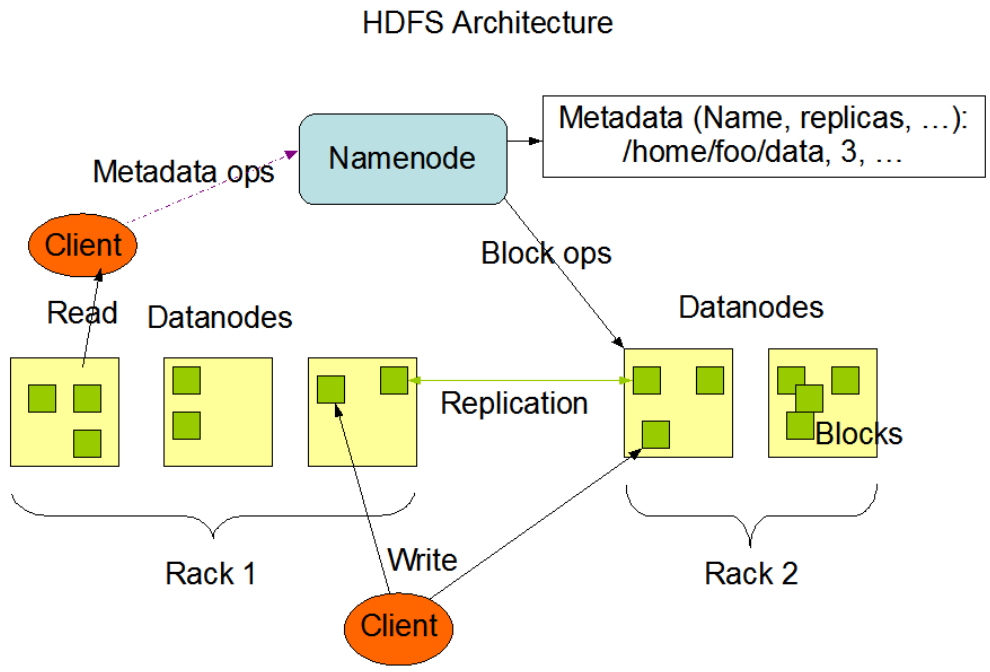

Ilustración 3. Arquitectura de funcionamiento del *HDFS*

Además, los bloques encajan perfectamente con la necesidad de tener la información replicada y siempre disponible. La forma de conseguir esto con los ficheros es sus bloques a través de las distintas máquinas de la red. Los bloques que se quedan no accesibles por cierto tiempo porque están corruptos o por problemas con el servidor que los aloja, pueden ser replicados desde sus otras localizaciones a nuevos servidores para mantener el factor de replicación.

2.1.4.2.2 Arquitectura

Un clúster de *HDFS* está formado por:

- Un ***NameNode***.
- Uno más ***DataNodes***.

- Un ***SecondaryNameNode***.

El *NameNode* es el encargado de gestionar los metadatos para todos los ficheros y directorios del *HDFS*. Este nodo será el que conecte con todos los *DataNodes* para que almacenen la información que entra y sale del clúster, mientras que mantiene una relación de *chunks* por fichero y de donde están alojados esos *chunks*.

HDFS posee una interfaz similar a la que implementa el sistema de ficheros POSIX, de manera que un programa o un cliente que quiera manejar información que se encuentre dentro del sistema de ficheros puede abstraerse de toda la funcionalidad que implementan por debajo el *NameNode* y los *DataNodes*.

Los *DataNodes* son los nodos encargados de almacenar y devolver los ficheros cuando es necesario, bien porque lo pide el *NameNode* o lo pide algún programa cliente. Además, reporta periódicamente al *NameNode* con listados de bloques que mantiene bajo su control.

Existe un tercer actor en este escenario, y el nodo que desempeña el rol de *SecondaryNameNode* (*SNN*). Este, al contrario de lo que pueda parecer, no es otro *NameNode*, ni puede sustituirlo si tuviera que ser necesario.

El *SNN* se encarga de obtener una "foto" de la imagen del *NameNode* cada cierto tiempo, y efectúa ciertas tareas de mantenimiento en los logs del nodo maestro. Esto sirve para que, en caso de error o pérdida en el maestro, se pueda recuperar un estado anterior que se conozca estable, de forma que se pueda crear un nuevo *NameNode* a partir de estos datos.

2.1.4.3 MapReduce

MapReduce es un modelo de programación además de una implementación para procesado y tratamiento de grandes cantidades de información. Es una de las 2 piezas más importantes de Hadoop, junto con el *HDFS*, y está basado en un artículo publicado por Google en el año 2004 (Dean and Ghemawat 2004).

El modo de funcionar de *MapReduce*, a grandes rasgos, consiste en tomar una entrada idealmente de un tamaño del orden de gigabytes o terabytes de tamaño, dividirla en trozos de un tamaño más pequeño, y realizar una serie de operaciones que operan sobre esos datos de forma paralela en multitud de ordenadores que trabajan en conjunto.

En las siguientes secciones se darán más detalles sobre esto.

Arquitectura

MapReduce tiene dos procesos que se encargan de dividir y procesar los Jobs que un usuario o un agente externo encola al sistema. Estos son:

- El *JobTracker*.
- El *TaskTracker*.

El modo de funcionamiento de estos dos procesos es análogo al ya explicado para *HDFS*, puesto que en arquitectura son muy parecidos.

El ***JobTracker*** harías las veces de maestro (similar al *NameNode* del sistema de ficheros) y sería el encargado de recibir los nuevos trabajos (o *Jobs*) y dividirlos en tareas (o *tasks*). También será el encargado de recibir los resultados tanto del *map* como del *reduce* (se verán más adelante) y de presentarle los resultados al agente que pidió la ejecución del *job*. Además, el *JobTracker* realizará tareas de monitorización básicas sobre los *TaskTrackers*, de manera que detectará cuando alguno de ellos queda inaccesible (el nodo muere) para dejar de enviarle tareas, o bien para repartir las tareas que ese nodo tenía asignadas entre el resto de nodos vivos.

Los ***TaskTrackers*** son los agentes encargados de recibir esas tareas en las que se ha subdividido el trabajo y de procesarlas. El *TaskTracker* recibirá una porción de los datos que necesitan ser procesados y el código que tiene que ejecutar. Una vez que haya finalizado su trabajo, su misión es devolver el resultado al maestro, el *JobTracker*.

2.1.4.3.1 Modelo de programación

De forma genérica, *MapReduce* toma una tupla (par clave – valor) por entrada, y genera un conjunto de tuplas como salida. *MapReduce* divide su ejecución en 2 funciones inspiradas en la programación funcional: la función *map* y la función *reduce*.

2.1.4.3.2 Map

La fase *map* no es más que un conjunto de valores o datos, a los que debe aplicarse, de forma independiente, una función definida por el usuario. Como resultado de esta operación se obtendrá un conjunto de nuevos pares clave – valor, que serán ordenados convenientemente según su clave antes de presentarse a la salida de la función.

Map(k1,v1) -> list(k2,v2)

2.1.4.3.3 Reduce

La fase *reduce*, por su parte, recibe un conjunto de tuplas a las que aplica, en conjunto, una función definida por el usuario. Esta función puede ser extremadamente simple (una suma, o una concatenación) hasta tan complicada como se desee. Así, generará a su vez una salida que corresponderá a una lista de nuevos valores para la clave inicialmente dada.

Reduce(k2, list (v2)) -> list(v3)

Así, globalmente se puede resumir el proceso *MapReduce* como un conjunto de valores en formato clave valor que son tratados y transformados, dando como resultado del proceso una lista con los valores finales.

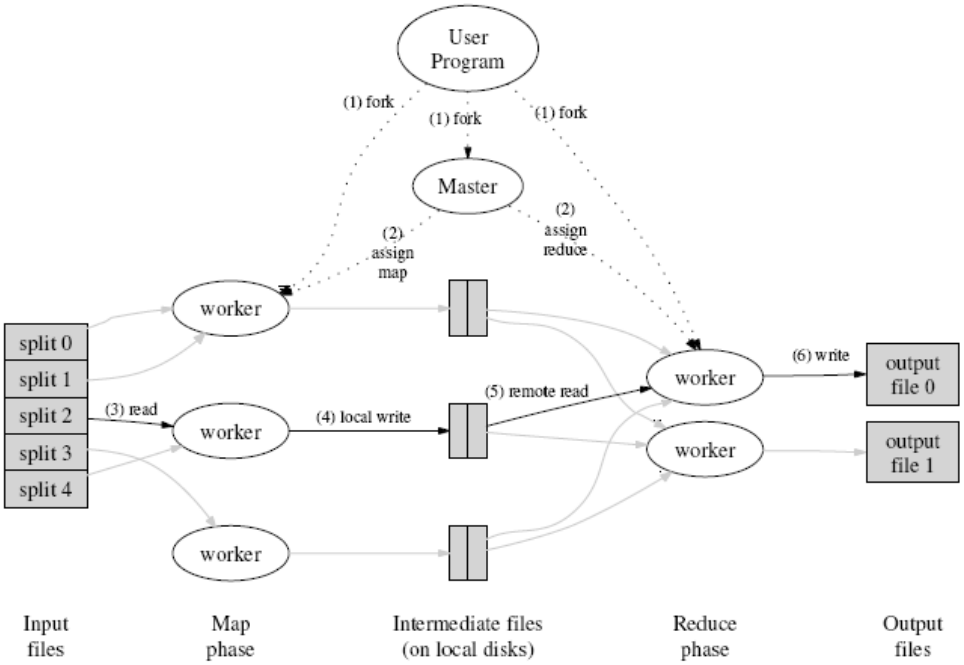

Ilustración 4. Diseño de una aplicación *MapReduce*

2.1.4.3.4 Usos habituales

El framework *MapReduce* se suele utilizar para procesar en paralelo grandes cantidades de información, del orden de gigas/teras de datos. Procesar cantidades pequeñas de datos puede tener una penalización elevada debido al uso en gran medida de un almacenamiento distribuido en red. Al tener que recuperar la información de otras máquinas, el tiempo de latencia sube, al igual que el tiempo de procesamiento general.

Un uso bastante común es el del la interpretación y procesado de logs. Comúnmente, los servicios web o de aplicaciones generan una gran cantidad de información sobre el uso que se le está dando a las aplicaciones que mantienen alojadas. Estos logs suelen crecer enormemente con el paso del tiempo, especialmente si son frecuentemente accedidas por muchos usuarios simultáneamente.

Otro de los usos habituales es el de ***Machine Learning*** o Aprendizaje automático. Existen adaptaciones concretas de varios algoritmos de regresión, clusterización, redes de neuronas, etc. al ecosistema Hadoop. Contar con un sistema de almacenamiento y procesado de grandes cantidades de información es sin lugar a dudas un escenario perfecto para este tipo de sistemas que, habitualmente, requiere de mucha información venida de diferentes fuentes para ser capaz de tomar decisiones relevantes.

Google fue la compañía que redactó los artículos originales en los que plasmó las ideas que dieron forma a las distintas piezas de las que se compone Hadoop. Es por este motivo por lo que se considera a la compañía de Mountain View todo un referente a la hora de abordar problemas con este tipo de tecnología.

Hasta hace unos años, el uso más importante que le dan a sus clústers de *GFS* (Google File System, el sistema de ficheros en el que se basa *HDFS*) y de *MapReduce* tal y

como se ha presentado en este documento era el de la creación de los índices que utiliza el propio buscador. Esto es debido a la ingente cantidad de información que Google procesa e indexa diariamente. Actualmente, estas tecnologías se han sustituido por el sucesor natural del *GFS*, una revisión del sistema de ficheros original, como consecuencia del proyecto *Caffeine* (Google 2010).

2.1.4.4 YARN

YARN (las siglas de *Yet Another Resource Negotiator*) es el sustituto del componente *MapReduce* en las versiones más actuales de Hadoop.

La idea es que el *JobTracker* hasta ahora se estaba encargando de muchas tareas, a veces demasiadas: llevar el rastreo de cientos de servidores, trabajos, tareas de *map* y de *reduce*… Como se ha comentado anteriormente, el principal objetivo de *YARN* es el de lograr desacoplar el gestor de recursos que tiene *MapReduce* del manejador de trabajos y tareas que son encoladas al sistema. De esta manera se puede incluso incrementar la escalabilidad, al tener muchos más servidores que, como se verá más adelante, se encarguen de tareas más específicas.

Para realizar estas tareas, *YARN* propone una arquitectura un tanto distinta a la que estipula *MapReduce*. Además, *YARN* es totalmente compatible con el modelo *MapReduce*, de forma que se pueden ejecutar los trabajos que han sido programados en *MapReduce* en el nuevo modelo de arquitectura *YARN*.

2.1.4.4.1 Arquitectura

YARN realiza una monitorización activa de los nodos del clúster. Esto le sirve para poder realizar un mejor reparto de los trabajos cuando son añadidos al sistema, de manera que se puedan optimizar los recursos que se poseen en el clúster en función de las necesidades de cómputo.

Para realizar esta tarea, *YARN* utiliza dos procesos propios: el ***ResourceManager*** (*RM*) y los ***NodeManagers*** (*NM*).

El *RM* es el servidor maestro, que se encarga de recolectar todas las métricas de cada uno de los nodos. Estas métricas son enviadas por los *NMs* que corren en cada máquina del clúster.

2.1.4.4.2 Procesamiento de trabajos

Por otro lado, existe otro servicio que hace las funciones de maestro para los trabajos. Es decir, el trabajo que antes desempeñaba el *JobTracker*. Se llama ***ApplicationMaster*** (*AM*), y a diferencia del *JobTracker*, puede correr en cualquiera de los nodos de forma independiente para cada aplicación.

Además, se introduce el concepto de contenedor (conocido como "*container*" en su voz inglesa). Los contenedores son conjuntos de recursos (CPU, memoria, disco, etc) que son creados por los *NMs* y que son utilizados para ejecutar las tareas que les encargan los *AMs*.

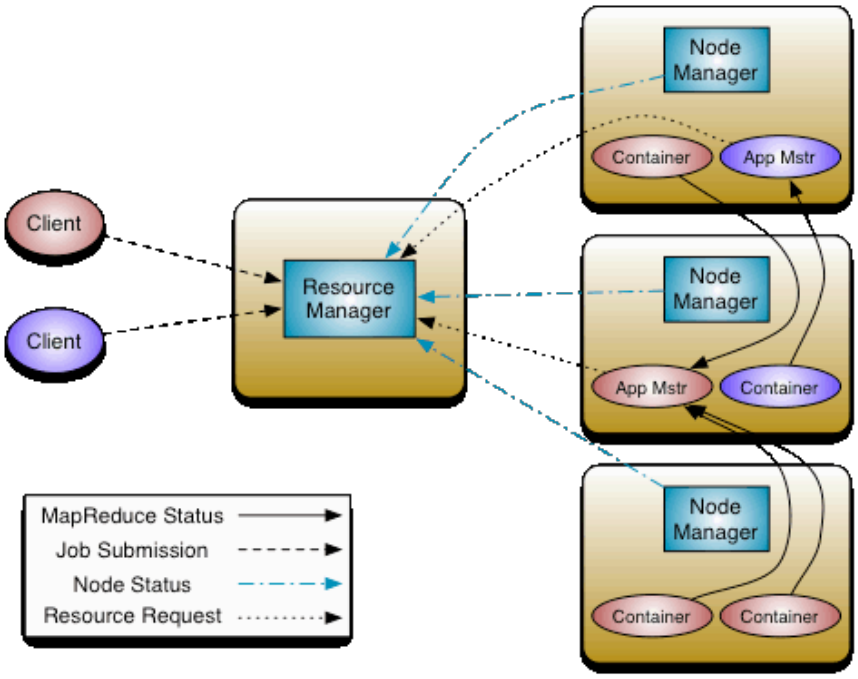

Ilustración 5. Arquitectura YARN

En resumen, por cada nuevo trabajo que se encola al sistema se genera un *AM*, y este puede ser alojado en cualquiera de los nodos esclavos del clúster. Además, por cada uno de los trabajos se utilizan los *NMs* como esclavos, los cuales cuando se les asigne la ejecución de una tarea, crearán un nuevo contenedor. Este contenedor proporcionará la base de recursos necesarios para poder ejecutar finalmente la tarea.

2.2 Sistemas Tradicionales de Almacenamiento de Información

A lo largo de la historia han surgido multitud de sistemas de almacenamiento de información (bases de datos, ficheros, etc.) para cubrir necesidades que se iban presentando en materia de almacenamiento. Así, se pueden encontrar las **bases de datos jerárquicas**, que estructuran su información en forma de árbol, y sirven por ejemplo para mantener el espacio de nombres que manejan los servidores de nombres de dominio o DNSs (*Domain Name Servers*). O bien las **bases de datos en red**, que son un modelo de base de datos ampliado de las bases de datos jerárquicas, formadas por un conjunto de datos que están interconectados entre sí mediante conexiones de red.

Se hará especial hincapié en este apartado en las bases de datos relacionales, una de las opciones más utilizadas hoy en día, con la finalidad de comprobar si éstas son útiles para el almacenamiento eficiente de *Big Data*.

2.2.1 Bases de Datos Relacionales

El modelo relacional ha sido desde principios de los 70 y aun hoy en día continúa siendo la forma habitual de estructurar y almacenar la información en cualquier tipo de aplicación informática, sobre todo en las transaccionales (aquellas aplicaciones con numerosos usuarios que permiten actualizaciones de la BBDD de forma continuada y consultas pocos complejas).

Las bases de datos relacionales son aquellas bases de datos que cumplen con el modelo relacional, que se fundamenta en el uso de relaciones. Una relación no es más que un vínculo entre dos entidades de la base de datos. La información contenida en las bases de datos relacionales y las relaciones de dependencia que se establecen entre sí, interpretados, persiguen describir el mundo real.

La persona que estructuró y definió los conceptos de las bases de datos relacionales fue el Edgard Frank Codd, ingeniero inglés que trabajaba para IBM. Publicó sus teorías sobre modelado de datos en el año 1970 (E. F. Codd 1970).

También postuló las tres formas normales que aplican a la normalización de bases de datos, y junto a Raymond Boyce creó la Forma Normal Boyce-Codd (a veces abreviada como FNBC), normalización que lleva los apellidos de ambos en su honor.

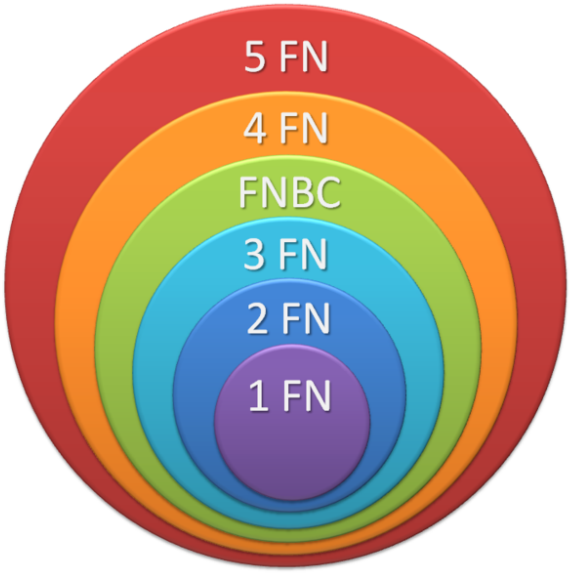

Ilustración 6. Formas normales[3]

Por aquella época, Larry Ellison tomó como referencia las ideas de Codd y fundó la base de datos con nombre **Oracle**, la primera base de datos relacional con licencia comercial. Además Codd acuñó las siglas **OLAP** (On Line Analytical Processing), una solución para gestionar grandes cantidades de datos para el ámbito del *Business Intelligence* (Codd, Codd and Salley, Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate 1993), y escribió 12 reglas que cualquier sistema de base de datos debía cumplir para ser considerado un Sistema Gestor de Bases de Datos (SGBD) relacional, que actualmente se conocen por L*as 12 Reglas de Codd* (E. F. Codd, The Relational Model for Database Management: Version 2 2000).

Se puede decir, por tanto, que Edgard Codd es la personalidad más importante en el ámbito de las bases de datos relacionales desde su concepción hasta la actualidad.

[3] Imagen extraída del libro (de Miguel and Piattini 1997)

2.2.1.1 Características del modelo de datos relacional

El modelo de datos relacional es el modelo actualmente más utilizado para diseñar bases de datos. Se considera una base de datos como base de datos relacional cuando cumple con el modelo relacional.

El modelo de datos relacional es un modelo matemático de datos. El componente principal del modelo relacional son las relaciones. Una **relación** es una interconexión entre los datos, que a su vez se almacenan en tablas. Así, en sentido inverso, se dice que dos tablas se relacionan entre sí por la relación que existe entre sus datos.

Los componentes básicos de una base de datos relacional son:

- **Tablas**. Una base de datos relacional puede contener una o varias tablas donde se estructurarán los datos. Las tablas no pueden repetirse dentro de la misma base de datos

- **Columnas**. Subdivisiones lógicas que conforman una tabla.

- **Registros**. Una tabla está formada por el conjunto de registros (también conocidos como **filas** o **tuplas**), es decir un conjunto de valores definidos para las columnas definidas en una tabla concreta.

- **Relaciones** entre tablas. Las relaciones entre tablas se establecen mediante claves primarias y claves ajenas.

- **Clave primaria**. Cada registro posee una única clave primaria, y son claves que identifican unívocamente a un registro, (es decir son irrepetibles en toda la tabla).

- **Clave ajena**. Las claves ajenas son referencias colocadas en las tablas hijas de las relaciones entre tablas. Contienen el mismo valor que la clave primaria del registro padre.

Además, el modelo relacional es un modelo de datos basado en la **lógica de predicados** y en la **teoría de conjuntos**. Esto significa que se pueden utilizar los principios del álgebra relacional y el cálculo relacional para manipular y operar con los datos almacenados.

Finalmente, se puede decir que una base de datos relacional es un conjunto de una o más tablas, estructuradas en **registros** o **tuplas** y que se relacionan entre ellas mediante diversas claves. Además, los principios del álgebra relacional y la teoría de conjuntos nos permitirán tratar la información que se quiera almacenar.

2.2.1.2 ¿Son eficientes para el almacenamiento de Big Data?

Este tipo de bases de datos presentan una serie de problemas cuando se utilizan como sistema de almacenamiento para la cantidad de datos que se manejan en los sistemas Big Data.

Además, al tener un esquema fijo, son poco flexibles a la hora de modificar la estructura de la base de datos una vez prefijada, y una migración de datos, dependiendo del volumen que se tenga puede llegar a ser un auténtico problema. Esto unido a la tendencia actual por parte de los equipos de utilizar metodologías ágiles, en los que se

realizan demos y entregas a veces semanales, en las que se necesita estar modificando constantemente el modelo de datos, hace que estas bases de datos sean poco queridas por parte de estos equipos.

Actualmente, las redes sociales tienen una presencia enorme en la vida de internet y en los hábitos diarios de muchas personas. Los sitios web en tiempo real y aplicaciones móviles, habitualmente requieren de cierto nivel de escalabilidad y flexibilidad que a las bases de datos relacionales les cuesta proporcionar.

Particularmente interesante es el artículo impulsado por los investigadores del MIT (Stonebraker, et al. 2007) En el que se expone la necesidad de plantearse el modelo de bases de datos que se utiliza en la actualidad. Aquí analizan la utilización de recursos de las máquinas, así como el uso que se hace de estas en temas como la computación distribuida o la alta disponibilidad.

Parece razonable tratar de optar por una solución pensada y diseñada para almacenar información sin unas restricciones tan grandes a nivel de seguridad, de acceso y de esquema fijo, y sí haciendo especial hincapié en la facilidad de escalado, la rápida inserción y recuperación de los datos y la disponibilidad o consistencia de los datos. Por ello, a continuación se verá más en detalle un conjunto de bases de datos que, al igual que el resto de sistemas de almacenamiento, tiene sus pros y sus contras y serán más eficientes dependiendo del caso de uso.

2.2.2 Bases de Datos NoSQL

En los últimos años, una gran variedad de bases de datos NoSQL han salido a la luz, creadas por compañías principalmente para cubrir sus propias necesidades. Temas como escalabilidad, rendimiento, mantenimiento, etc. que no encontraban en ninguna solución que existía en el mercado.

Debido a la variedad de enfoques que existe entre requisitos y funcionalidades que debe cumplir una base de datos NoSQL, es bastante difícil mantener una visión general de la situación actual de las bases de datos no relacionales.

Se puede decir que las bases de datos NoSQL son una categoría independiente dentro del conjunto de bases de datos. Más adelante se expondrán diversos conceptos y características comunes de las bases de datos NoSQL. Ahora se hará una clasificación general de las bases de datos NoSQL más importantes atendiendo a su modelo de datos.

2.2.2.1 Bases de Datos Clave – Valor

Las bases de datos clave – valor (también llamadas a veces como *tablas hash* o *simplemente tablas*) se estructuran almacenando la información como si de un diccionario se tratara. El diccionario, en este caso, es un tipo de datos que contienen tuplas clave valor. Los clientes añaden y solicitan valores a partir de una clave asociada que conocen de antemano.

Los sistemas modernos de almacenamiento clave – valor se caracterizan por tener una elevada escalabilidad y un rendimiento muy bueno para volúmenes de datos muy grandes. A cambio, su estructura es muy sencilla, y sacrifica ciertas funcionalidades como la consistencia inmediata, la verificación de la integridad de datos o las referencias externas.

En el caso concreto de las referencias externas y la integridad referencial, deberá ser la propia aplicación que hace uso de la base de datos la que se encargue de actualizar los valores correspondientes cada vez que exista un borrado o una modificación sobre el conjunto de datos.

En los sistemas de bases de datos relacionales existen agrupaciones o componentes llamados "*base de datos*" (o su voz inglesa: "*databases")*, y dentro de ellas existen contenedores llamados tablas Por el contrario, en los sistemas clave valor existen los "contenedores" (o su voz inglesa: "*cabinets*"), y dentro de ellos se pueden almacenar tantos pares clave-valor como se desee.

Existen distintos tipos de *contenedores*, algunos permiten pares duplicados, otros pares admiten valores nulos, etc.

Muchas de estas bases de datos funcionan almacenando la información en memoria principal, de manera que aumenta aun más la velocidad de respuesta ante nuevas lecturas / escrituras. Otras también realizan un copiado periódico a disco para persistir los datos, y poder recuperarlos más adelante si se establece un punto de restore, o bien la maquina cae y se desea recuperar una imagen concreta de cierto momento.

Mas adelante se analizará en profundidad las bases de datos de tipo clave – valor más utilizadas, como Memcached, Redis o DynamoDB.

2.2.2.2 Bases de Datos Documentales

Las bases de datos documentales o bases de datos orientadas a documento, son otro tipo de base de datos NoSQL con un grado de complejidad y flexibilidad superior a las bases de datos clave – valor.

En las bases de datos documentales el concepto principal es el de "documento". Un documento es la unidad principal de almacenamiento de este tipo de base de datos, y toda la información que aquí se almacena, se hace en formato de documento.

Las codificaciones más habituales de estos documentos suelen ser XML, YAML o JSON, pero también se pueden almacenar en formato Word o PDF.

Habitualmente, los documentos se estructuran dentro de una serie de contenedores llamadas colecciones (o su voz inglesa: "collections") que son proporcionados por el sistema gestor documental.

Los documentos son almacenados dentro de la base de datos con una clave única dentro del almacén, por la cual son también recuperados posteriormente. También existirá un índice principal por esta clave primaria. Se pueden generar índices que afecten a otros campos, pero hay que evaluar bien su construcción, ya que aceleran y mejoran los tiempos de carga por ese campo pero a cambio tienen necesidades de espacio y de mantenimiento a tener en cuenta.

2.2.2.3 Bases de Datos Orientadas a Grafos

Las bases de datos orientadas a grafos tienen la particularidad de representar la información como si de un grafo se tratara. La información viene representada por los nodos, y las relaciones entre los datos por las aristas. De este modo, se puede emplear la

teoría de grafos para recorrer la base de datos y así gestionar y procesar la información (Velasquez 2013).

Una base de datos orientada a grafos, de forma generalizada, es cualquier sistema de información donde cada elemento tiene un puntero directo hacia sus elementos adyacentes, es decir, no sería necesario realizar consultas mediante índices.

Como se expone a continuación, existen varios casos en el que el uso de una base de datos orientada a grafos es más eficiente que utilizar cualquier otro tipo de base de datos, tanto relacional como NoSQL.

A diferencia de la mayoría de bases de datos, el **rendimiento** de una base de datos orientada a grafos no se deteriora con el crecimiento de la base de datos, ni con consultas o procesamientos muy intensivos de los datos. Esto es debido a que el rendimiento será siempre proporcional al tamaño y el rendimiento que tenga el recorrer la parte del árbol que esté implicada, y no el tamaño global del grafo.

El modelo de grafos dota al desarrollador de aplicaciones de una elevada **flexibilidad** para manipular el conjunto de datos, ya que le permite conectar los datos de forma que posteriormente sea sencillo realizar consultas o actualizaciones desde una aplicación.

Este tipo de bases de datos nos permiten realizar un mantenimiento progresivo y **ágil** de los sistemas, a medida que la aplicación va creciendo, de forma que se puede adaptar a las nuevas necesidades del negocio.

2.2.2.4 Bases de Datos Orientadas a Objetos

En las bases de datos orientadas a objetos la información se representa como objetos. Esta manera de representar la información es análoga a la que hace la programación orientada a objetos, que define y representa la información en un conjunto de datos y de operaciones que se pueden realizar sobre esos datos.

Cuando un sistema aúna una base de datos de este tipo y un lenguaje de programación orientado a objetos, el resultado es un sistema gestor de bases de datos orientado a objetos.

La clave de este tipo de almacenamiento reside en la potencia de los lenguajes de programación orientados a objeto, junto con la capa de persistencia que le proporciona una base de datos que ha sido diseñada específicamente para trabajar con aplicaciones desarrolladas con lenguajes orientados a objetos.

Los vendedores de bases de datos relacionales tradicionales se dieron cuenta de esto, y poco a poco fueron implementando capas de integración con los diferentes lenguajes de programación más populares, en lo que se conoce como sistemas de mapeo objeto–relacional (también conocido por sus siglas ORM). Sin embargo, esta traducción de objetos al modelo relacional no siempre se puede llevar a cabo, puesto que siempre se perderá algo de algún lado. O bien el modelo orientado a objetos perderá para adaptarse al 100% al modelo relacional, o bien el modelo relacional asociado al modelado orientado a objetos no cumplirá al 100% el paradigma relacional.

Se puede decir que algunas de las ventajas que tienen las bases de datos orientadas a objetos con respecto al modelo relacional son:

- Deja de emplearse el SQL como lenguaje de base de datos. En su lugar, es el propio código de aplicación la que realiza esta funcionalidad.
- Se elimina la doble representación del modelo de datos: el modelo de los objetos y el relacional. Ahora el diagrama de clases de la aplicación es el que construye el modelo de persistencia de la base de datos.
- Mejora el rendimiento cuando se trata con objetos muy complejos.
- Cualquier cambio en los objetos se aplican directamente en base de datos, no es necesario reconfigurar ni migrar nada.

Las bases de datos orientadas a objetos son, probablemente, las más antiguas de todas las bases de datos NoSQL, ya que el pulso que han mantenido con las bases de datos relacionales data de antes de que Google liberara los artículos de su GFS (Google File System) y Bigtable, la base de datos NoSQL creada por Google, y desde antes de que el movimiento NoSQL empezara a coger fuerza.

2.2.2.5 Bases de Datos Orientadas a Columnas

Las bases de datos orientadas a columnas son otro caso particular de la enorme familia de las bases de datos NoSQL. En este tipo de almacenes, en contraposición con el modelo relacional, la información se estructura en columnas en lugar de en filas.

Algunas de las bases de datos NoSQL más importantes y con una mayor aceptación pertenecen a este grupo. Bigtable, la solución NoSQL de Google, HBase, la base de datos de Hadoop, Cassandra, impulsada por Facebook y ahora bajo los brazos de Apache Software Foundation, son solo algunos ejemplos de este tipo de bases de datos.

Las comparativas de rendimiento entre los sistemas de gestión por filas (especialmente RDBMS) y los basados en columna, vienen principalmente de la mano de la eficiencia de los accesos que realizan a disco. Los accesos a posiciones consecutivas se producen en una cantidad de tiempo sensiblemente menor que los accesos que se producen a posiciones aleatorias a disco. Tener un sistema con multitud de accesos aleatorios a disco puede lastrar en gran medida el rendimiento del sistema de almacenamiento.

Algunas de las ventajas de utilizar bases de datos orientados a columna son:

- Los sistemas de almacenamiento basados en columnas proporcionan un mejor rendimiento cuando es necesario realizar una transformación de datos que involucra a todas o gran parte de las filas, pero solo a una o una pequeña parte de las columnas. El acceso columnar, en este caso, produce un rendimiento mayor.
- Los sistemas de base de datos basados en columnas son más eficientes cuando hay que sustituir un valor que afecta a una o más columnas en todas las filas de la colección.
- Los sistemas basados en filas son más eficientes en los casos en los que es necesario obtener a la vez varias columnas de una misma fila, siempre que la fila sea lo suficientemente pequeña (esto es, que se pueda obtener con una única búsqueda aleatoria a disco).
- Los sistemas de almacenamiento basados en filas son más eficientes cuando hay que escribir una fila nueva, se proporcionan todos los valores referentes a las columnas, y además la escritura se puede realizar con una única búsqueda en el disco.

El resumen que se puede hacer finalmente es que, las bases de datos basadas en columnas se emplean con mayor rendimiento cuando se utilizan para agilizar la consulta de grandes cantidades de información. Se ven beneficiadas técnicas como la Minería de Datos (en inglés: *Data Mining)*, la inteligencia de negocio (en inglés *Bussiness Intelligence*), informes de marketing (en inglés *Marketing Reports*), ventas, etc.

Por otro lado, los sistemas basados en filas proporcionan habitualmente mejor rendimiento cuando la aplicación realiza frecuentes transacciones de base de datos, especialmente en arquitecturas cliente-servidor. El comercio electrónico, la banca o cualquier servicio que requiera la interacción con un usuario o cliente se verá beneficiado de este tipo de base de datos.

Tabla 1. Tabla comparativa de almacenamiento en filas vs almacenamiento en columnas

	Sistemas basados en columnas	Sistemas basados en filas
Útil para	Consultas de grandes cantidades de información.	Frecuentes transacciones
Usos habituales	Data Mining Bussiness Intelligence Informes de marketing	Comercio electrónico Banca Arquitectura cliente-servidor

2.2.2.6 Bases de Datos Multivalor

Las bases de datos multivalor conforman otra familia de bases de datos NoSQL, y sus principios son casi tan antiguos como los de las bases de datos relacionales.

Las bases de datos multivalor persiguen la idea de la propiedad multivalor, que es aquella propiedad que puede estar representada por varios valores al mismo tiempo (OpenQM 2014).

Para entender el principio de estas bases de datos es necesario entender las bases de datos relacionales, y más concretamente las Reglas de la Normalización. Las más importantes son las Tres Formas Normales, que dicen:

- **Primera forma normal**: Eliminar datos repetidos creando una tabla independiente para cada conjunto de datos duplicados, o lo que es lo mismo, tener una única tupla por cada valor de clave primaria
- **Segunda forma normal**: Tablas independientes para conjuntos de datos que aplican a múltiples registros. Esto es, evitar la redundancia de información, almacenando los datos comunes en tablas aparte y referenciándolas desde donde tenga que ser referenciada.
- **Tercera Forma Normal**: Eliminar los campos que no dependan de la clave primaria.

Según los principios de las bases de datos multivalor, la segunda y tercera formas normales son buenas prácticas, pero la primera forma normal no debería ser aplicada, es decir, la información duplicada forma parte del registro original, y no debería ser extraída y llevada a otras tablas.

Un buen ejemplo para entender este modelo sería el de la gestión que realiza un supermercado de las compras que realizan sus clientes. Un ejemplo de bases de datos con datos multivaluados sería el siguiente, donde, por ejemplo el pedido "12001" tiene dos posibles valores para la columna "Producto":

Tabla 2. Tabla de compras de un supermercado

Pedido	Fecha	Cliente	Producto	Cantidad	Precio
12000	3 Junio	1001	101	3	3.50
12001	3 Junio	1002	107	1	1.75
			108	2	4.00
12002	6 Junio	1003	201	8	8.30

En este ejemplo el pedido 12001 está reflejado en una única fila, mientras que, si se siguiera la normalización que nos propone la primera forma normal, habría que separar los campos que se repiten (producto, cantidad y precio) en una tabla nueva.

Este ejemplo es muy sencillo, pero si se imagina el crecimiento de la tabla a medida que se van realizando más y más pedidos, se podrá observar la ganancia de rendimiento de este modelo de datos. Además, de este modo se pueden reducir el número de tablas totales de la base de datos.

2.2.2.7 Ejemplos de tecnología asociada a bases de datos NoSQL

A continuación se darán ejemplos de cada una de las clasificaciones proporcionadas en el apartado anterior, así como algunas nuevas existentes pero de menos calado en la comunidad (Edlich 2014).

No es extraño que, mientras se investiga en la literatura de la materia, se pueden encontrar la misma base de datos clasificada en distintas categorías. Por ejemplo, Cassandra habitualmente se suele agregar a la categoría de las bases de datos basadas en columnas, pero también se puede encontrar clasificado como un contenedor Clave-Valor. Esto es así porque dependiendo del uso que se haga de la misma, y de las configuraciones que se le apliquen, pueden desempeñar unas funcionalidades u otras.

2.2.2.7.1.1 Bases de datos Clave-Valor.

- DynamoDB
- Memcached[4]
- Project Voldemort[5]
- Scalaris[6]
- Tokyo Cabinet[7]
- Dynomite[8]
- Redis
- Riak[9]

2.2.2.7.1.2 Bases de Datos Documentales.

- MongoDB
- SimpleDB[10]
- CouchDB[11]
- Terrastore[12]

2.2.2.7.1.3 Bases de Datos de Grafo.

- Neo4j
- AllegroGraph[13]
- FlockDB[14]
- Titan[15]
- Sparksee[16]
- InfiniteGraph[17]
- InfoGrid[18]

2.2.2.7.1.4 Bases de Datos Orientadas a Objetos

- db4o[19]
- GemStone[20]

---

[4] http://memcachedb.org/
[5] http://www.project-voldemort.com/
[6] https://code.google.com/p/scalaris/
[7] http://fallabs.com/tokyocabinet/
[8] https://github.com/moonpolysoft/dynomite/
[9] http://docs.basho.com/
[10] http://aws.amazon.com/es/simpledb/
[11] http://couchdb.apache.org/
[12] https://code.google.com/p/terrastore/
[13] http://franz.com/agraph/
[14] https://github.com/twitter/flockdb
[15] https://github.com/thinkaurelius/titan/
[16] http://www.sparsity-technologies.com/
[17] http://www.objectivity.com/infinitegraph
[18] http://infogrid.org/trac/
[19] http://db4o.com/
[20] http://gemtalksystems.com/

- InterSystems Caché[21]
- NeoDatis ODB[22]
- ObjectDatabase++[23]
- ObjectDB[24]
- Objectivity/DB[25]
- ODABA[26]

2.2.2.7.1.5 Bases de Datos Orientadas a Columnas

- Google Bigtable
- HBase
- Hypertable[27]
- Cassandra
- Accumulo[28]

2.2.2.7.1.6 Bases de Datos Multivalor

- OpenQM[29]
- Rocket U2[30]
- OpenInsight[31]
- InfinityDB[32]

Bases de datos Multimodelo

- ArangoDB[33]
- OrientDB[34]
- Datomic[35]
- AlchemyDB[36]

Bases de datos XML

- EMC Documentum[37]

---

[21] http://www.intersystems.com/
[22] http://neodatis.wikidot.com/
[23] http://www.ekkysoftware.com/ODBPP/
[24] http://www.objectdb.com/
[25] http://www.objectivity.com/
[26] http://www.odaba.com/content/start/
[27] http://hypertable.org/
[28] http://accumulo.apache.org/
[29] http://www.openqm.com/
[30] http://www.rocketsoftware.com/brand/rocket-u2
[31] http://www.revelation.com/index.php
[32] http://boilerbay.com/
[33] http://www.arangodb.org/
[34] http://www.orientechnologies.com/
[35] http://www.datomic.com/
[36] https://code.google.com/p/alchemydatabase/

- eXist[38]
- Sedna[39]
- BaseX[40]

Bases de datos multidimensionales.

- GlobalsDB[41]
- Intersystem Cache[42]
- GT.M[43]
- SciDB[44]

Bases de datos orientadas a Eventos

- Event Store[45]

3 NoSQL

Como ya se ha visto en apartados anteriores, los sistemas de bases de datos relacionales llevan mucho tiempo siendo el modelo informático más empleado del mundo para almacenar y recuperar la información.

La famosa frase "one size fits all" tiene que ver con el diseño de estas bases de datos relacionales porque a su vez tiene que ver con las necesidades de las empresas y las comunidades de usuarios para con las bases de datos.

Desde hace unos años esto ha cambiado. Ahora las necesidades son diferentes, y el BigData tiene buena parte de culpa. Revisando detenidamente los puntos que se han ido comentando a lo largo del trabajo, hay que darse cuenta de que los datos no son los mismos, ni están estructurados de igual manera. También ha cambiado la manera de almacenar la información, así como la de consumirla.

Esta sección pretende centrarse en estos nuevos conceptos y razones que han motivado un cambio de dirección enorme en el tratamiento de las bases de datos, dando lugar a multitud de nuevas formas de almacenamiento, con partes y estructuras comunes, y características propias y únicas que las diferencia de las demás. En este apartado se esbozarán las características generales de las bases de datos NoSQL.

---

[37] http://www.emc.com/products/detail/software2/documentum-xdb.htm
[38] http://exist-db.org/
[39] http://www.sedna.org/
[40] http://basex.org/
[41] http://globalsdb.org/
[42] http://www.intersystems.com/our-products/cache/cache-overview/
[43] http://www.fisglobal.com/products-technologyplatforms-gtm
[44] http://www.scidb.org/
[45] http://geteventstore.com/

3.1 El Movimiento NoSQL

El término NoSQL fue inicialmente utilizado en el año 1998, y fue para denominar una base de datos relacional que no utilizaba el lenguaje SQL para funcionar. A partir de aquí, el término fue rescatado en 2009 en unas charlas por defensores de las bases de datos no relacionales.

3.1.1 Motivaciones principales del movimiento NoSQL

Los principales motivos con los que un proyecto debe encontrarse para que se pueda comenzar a valorar la posibilidad de dejar atrás las bases de datos relacionales y comenzar a fijarse en las soluciones que se tratarán más adelante en este documento son las siguientes (Strauch 2011):

Evitar la complejidad innecesaria. Las bases de datos relacionales proporcionan gran cantidad de funcionalidades y restricciones para mantener la consistencia de los datos, en ciertos casos, mucho más de lo necesario.

Esto hace que, a nivel global, las operaciones en base de datos tarden más tiempo en ayudarían a incrementar el rendimiento.

Alto rendimiento. Muchas bases de datos NoSQL proporcionan un rendimiento superior al que ofrecen los sistemas RDBS convencionales. Para ilustrar esto, tan solo hay que ver ejemplos sobre los tiempos que tardan las grandes compañías en insertar información en sus bases de datos NoSQL. Por ejemplo, en el caso de Google y su Bigtable. Es capaz de procesar hasta 20 petabytes diarios (Lai 2009).

Escalabilidad horizontal y hardware de bajo coste. Al contrario que las bases de datos relacionales, los sistemas NoSQL han sido diseñados para escalar horizontalmente. El software está pensado para poder agregar o eliminar máquinas de forma sencilla sin tener un coste operacional realmente elevado.

Cuando tu CPD se compone del orden de cientos o miles de máquinas funcionando 24h al día 7 días a la semana, la probabilidad de que una de las máquinas tenga un fallo y haya que sustituir uno de sus componentes (o la maquina por completo) es muy elevada. Es por esto que es preferible contar con un montón de servidores comerciales, de coste moderado, a tener unos cuantos realmente caros y potentes.

Complejidad y coste de levantar un clúster de base de datos. Tiene que ver con el punto anterior, en el se exponía la facilidad y sencillez con la que estos sistemas son capaces de añadir y quitar nodos del sistema.

Comprometer la fiabilidad a cambio del rendimiento. La fiabilidad en los datos es un tema muy importante, pero existen ciertos momentos en los que se puede demandar un aumento del rendimiento a cambio de un nivel menos de fiabilidad.

La frase "One size fit's it all" ha sido y es incorrecta. Actualmente, existe un elevado número de problemas que no puede ser abordado a través de una visión tradicional en bases de datos. Muchas compañías, sobre todo relacionadas con internet han adoptado soluciones NoSQL en la empresa, seguramente animados por el creciente uso y aceptación de la tecnología. Pero muchas de estas tecnologías no eran lo

suficientemente maduras (muchas siguen sin serlo hoy en día), por lo que les ha tocado ir viéndolas crecer y estabilizarse versión a versión.

Bien es cierto que, muchas compañías, especialmente startups y empresas orientadas especialmente al mundo web han abrazado la tecnología NoSQL con fuerza, hay que tener cuidado a la hora de decidir implementar este tipo de tecnologías. Actualmente existen multitud de soluciones y de tipos de bases de datos, así que hay que evaluar todas las opciones convenientemente, familiarizarse con sus puntos fuertes y sus puntos débiles, etc.

Movimientos en lenguajes de programación y Frameworks de desarrollo. De un tiempo a esta parte, se han venido popularizando el hecho de dejar, funcionalmente hablando, las capas de acceso a bases de datos independientes del resto del código. Esto, que ya de por si es una buena práctica para bases de datos relacionales, adquiere un importante significado para el movimiento NoSQL, que se ha dado prisa en desarrollar los conectores para sus bases de datos para los distintos lenguajes de programación.

Requisitos de Cloud Computing. En una entrevista a Dwight Merriman de 10gen (la compañía que desarrolla y mantiene MongoDB) menciona los 2 principales requisitos de las bases de datos en entornos cloud computing: la alta escalabilidad, especialmente horizontal, y que los tiempos de administración sean lo mínimo posible.

Bajo su punto de vista, las siguientes bases de datos funcionarían bien en un entorno cloud.

- Bases de datos específicas para almacenamiento de datos por lotes y operaciones *MapReduce*.
- Almacenamiento clave/valor.
- Bases de datos cuya lógica se encuentra más cerca de los sistemas de bases de datos tradicionales que de los almacenamientos clave / valor, pero sin renunciar al las características de rendimiento y escalabilidad propias de estos.

Necesidades de ayer ante necesidades de hoy. En los 60 y 70, las bases de datos fueron diseñadas para correr en un único servidor muy potente, al contrario que la tendencia actual de muchas empresas actuales, sobre todo las orientadas a web, que tienen varias máquinas más baratas ya que se prevé que fallarán y habrá que reemplazarlas. También las aplicaciones han de ser diseñadas en consecuencia. Esto último es algo con lo que Amazon con su servicio AWS tiene que lidiar diariamente.

En todo momento, Amazon explica que todo puede fallar, y ha de ser tu aplicación la que esté preparada para hacer frente a una posible perdida de parte del hardware (DeCandia, et al. 2007).

Por otro lado, mientras que en las aplicaciones que utilizan sistemas de bases de datos relacionales el sistema trata de ocultar ciertos aspectos del modelo, los sistemas modernos dejan que sea la aplicación la que gestione enteramente qué y cómo se almacenará la información en la base de datos NoSQL.

Un caso de estudio curioso son las 8 falacias de la computación distribuida, las cuales fueron enunciadas por varias personalidades del mundo de la informática, entre ellos Peter Deutsch, fundador de Aladdin Enterprises (actualmente SafeNet[46]), y James Gosling, creador del lenguaje de programación Java. Estas leyes pretenden dar una serie de pautas para todo programador que se inicie en el mundo del desarrollo de aplicaciones distribuidas, y sus leyes son las siguientes.

1. La red es confiable.
2. La latencia es despreciable.
3. El ancho de banda es infinito.
4. La red es segura.
5. La topología de red no cambia nunca.
6. Existe un solo administrador.
7. El coste del transporte es despreciable.
8. La red es homogénea.

Posteriormente se demostró que la mayoría de estos preceptos no se cumple en casi ningún sistema distribuido de tamaño medio / grande.

3.1.2 Crítica general y oposición

A continuación se detallan los argumentos se esgrimen en contra del uso de las bases de datos NoSQL

3.1.2.1 Escepticismo desde el punto de vista del negocio

La mayoría de las tecnologías NoSQL tiene su base en el software open source, tan apreciado por muchos desarrolladores, que no tienen que preocuparse por temas relacionados con licencias o cuestiones de soporte. Sin embargo, la gente de negocio no suele compartir esta opinión, especialmente en caso de posible catástrofe, donde no existiría ningún tipo de soporte al que acudir.

3.1.2.2 El NoSQL como un producto con mucho "hype"

Ciertas empresas toman precauciones en lo que al movimiento NoSQL se refiere. Éstas argumentan que las ganas con las que la comunidad ha adoptado esta nueva tecnología podría ser contraproducente, ya que las ganas de que este tipo de almacenamiento funcione podrían ir en contra de que pueda responder ante todas las promesas que tiene que cumplir.

Habitualmente, cada avance por parte de un individuo o un grupo de individuos es muy bien recibido, tanto por el resto de compañeros informáticos más cercanos, como por parte de la comunidad. Esta pequeña euforia por continuar alcanzando éxitos, independientemente de su utilidad real, es esgrimida por quienes creen que los objetivos de este tipo de software deben centrarse más en intentar solucionar los problemas que se pueden presentar en un entorno profesional.

Por ejemplo, el uso de una base de datos NoSQL para abordar problemas sencillos o a pequeña escala, sería fruto de esa confianza que se deposita actualmente en este tipo de

[46] http://www.safenet-inc.com/

bases de datos de forma ciega. Los responsables de este tipo de decisiones son profesionales que deciden implementar estas soluciones tan solo porque el resto del mundo lo está haciendo, sin pararse a analizar si realmente las necesita.

El NoSQL no es algo nuevo.

Varios de los enfoques, e incluso varias de las bases de datos que se verán en este documento, llevan años existiendo. Algunas de ellas tienen tantos años de vida incluso como el propio modelo relacional, y llevan siendo una alternativa válida y estable desde entonces.

Si esto es así, ¿porque hasta ahora no se las ha empezado a tener en consideración?

3.1.2.3 El NoSQL entendido como un rechazo al SQL

En los primeros años, el NoSQL ha sido entendido de muy diferentes maneras, especialmente con Internet como medio de difusión.

Algunos blogs anunciaban y vaticinaban el fin de las bases de datos tan y como se conocen, criticando duramente en modelo de bases de datos relacional. Algunos incluso cambiaron las siglas de NoSQL, que significa "Not Only SQL" por la expresión "Not To SQL".

El rechazo frontal que debe existir ante este tipo de actitudes y pensamientos por cualquier profesional, unido al sector más defensor de las bases de datos tradicionales y del modelo relacional, hizo que el movimiento NoSQL estuviese mal viso por un sector importante de la industria informática en sus inicios.

3.1.2.4 Necesidad de administradores de sistemas y operadores de hardware.

Cuando se habla de NoSQL normalmente es desde el punto de vista de la funcionalidad, de las capacidades y de las ventajas e inconvenientes cuando lo comparas con otro tipo de bases de datos. Normalmente es la gente de desarrollo la que se dedica a inventar y explorar nuevas opciones NoSQL, y a hablar sobre ellas con otros desarrolladores, pero existe otro tipo de perfil que también es importante cuando se habla de un desarrollo NoSQL. Este es el operador de sistema.

Este rol, a veces dejado de lado en compañías dedicadas a computación en la nube, es cierto que cada vez va quedando poco a poco más relegado a un segundo plano (al menos así ocurre con el rol de administrador de sistemas clásico), pero es importante en este caso.

Estas personas serán las encargadas de solucionar problemas que puedan suceder con las colecciones de datos, así como diagnosticar el estado del clúster, preparar planes de sharding, particionado, migraciones de datos, y otro tipo de tareas diferentes a las que un desarrollador esta acostumbrado.

3.1.2.5 No todos los SGBD se comportan como MySQL

Existe una fuerte tendencia a hablar de MySQL como la única base de datos relacional que existe cada vez que se va a realizar una comparativa entre un SGBD y un motor NoSQL.

También existe un pensamiento generalizado de que las bases de datos NoSQL son las sucesoras de MySQL+Memcached.

La realidad es que existen multitud de sistemas de base de datos relacional. MySQL es tan solo uno más, puede que se utilice más o menos, pero al igual que existe una amplia oferta en la mayoría de campos de la informática (se está precisamente analizando el campo del NoSQL), las bases de datos relacionales no son una excepción.

Se han expuesto las características generales del movimiento NoSQL. Ahora se analizará más en profundidad como se aplica todo lo anteriormente expuesto al mundo de las bases de datos, dando lugar a las bases de datos NoSQL.

3.2 Bases de Datos NoSQL

3.2.1 Conceptos básicos, técnicas y patrones

Antes de pasar a ver más en profundidad los tipos de bases de datos, así como las propias bases de datos en sí, se analizarán los conceptos fundamentales, técnicas y patrones que son comunes a este tipo de bases de datos.

3.2.1.1 Consistencia

3.2.1.1.1 El Teorema CAP

El teorema CAP ha sido ampliamente adoptado por las grandes compañías de internet, al igual que por la comunidad NoSQL.

Las siglas CAP hacen referencia a:

- **Coherencia**. En sistema distribuido, habitualmente se dice que se encuentra en un estado consistente si, después de una operación de escritura, todas las operaciones de lectura posteriores son capaces de ver las actualizaciones desde la parte del sistema desde la que están leyendo.
- **Disponibilidad (Availability).** La alta disponibilidad se produce cuando el sistema ha sido diseñado e implementado de modo que se pueda continuar operando (lecturas, escrituras), incluso después de que un nodo quede indispuesto, o que algunas partes de hardware tengan que ser retiradas, debido a errores o actualizaciones.
- **Tolerancia a Particiones (Partition Tolerance).** Entendido como la habilidad de un sistema de tener diferentes regiones o divisiones lógicas en la red, y de ser capaz de seguir funcionando aunque una de estas partes quede inaccesible durante un tiempo.

La teoría CAP (también se conoce como teorema de Bewer (Bewer 2012)) expone que es imposible que un sistema distribuido pueda garantizar simultáneamente estas 3 características. Sin embargo, el teorema de CAP también dice que sí puedes garantizar 2 de estas 3 propiedades .

Los SGBD NoSQL cumplen dos de estas 3 características, y por tanto, se puede realizar una clasificación de dichos SGBDs en función de esto.

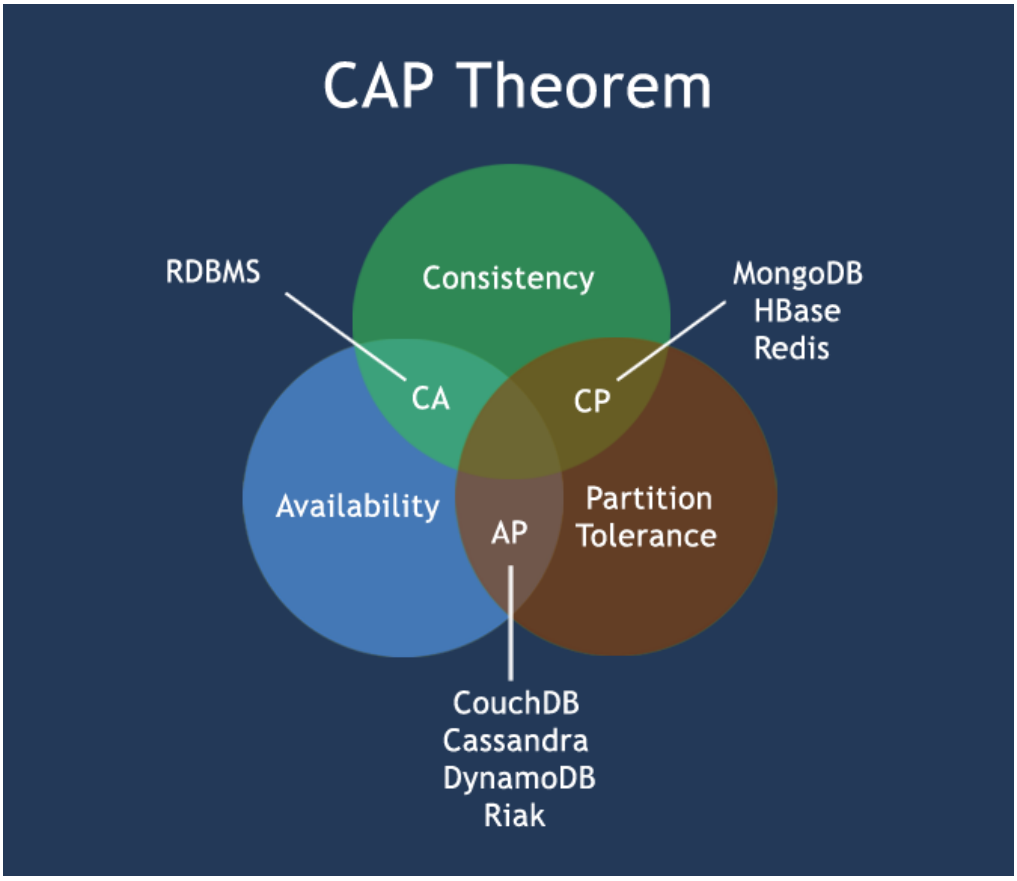

Ilustración 7. Teorema de CAP[47]

De este modo, atendiendo lo que se acaba de exponer en el punto anterior, podemos deducir que se puede "elegir" dos de estas 3 características par nuestro sistema de datos compartidos. De esta manera se tiene:

- Sistemas coherentes y disponibles, pero con dificultades para funcionar en caso de que haya muchas particiones.
- Sistemas coherentes y tolerantes a particiones, pero con ciertas carencias en temas de disponibilidad.
- Sistemas disponibles y tolerantes a particiones, pero no estrictamente coherente.

El siguiente cuadro ejemplifica lo anteriormente expuesto.

[47] Imagen extraída de (Internetria 2013)

Tabla 3. Teorema CAP

Elección	Características	Ejemplos
CA. Consistencia y Disponibilidad (se pierde la capacidad Particiones	Confirmaciones dobles Protocolos de validación de caché	Relacionales (Oracle, Mysql, SQL Server), Neo4J
CP. Consistencia y Particiones (se pierde la capacidad Disponibilidad)	Bloqueos "pesimistas" Ignorar las particiones más pequeñas	MongoDB, HBase, Redis
AP. Disponibilidad y Particiones (se pierde la capacidad Consistencia)	Invalidaciones de caché Resolución de conflictos	DynamoDB, CouchDB, Cassandra

Algunas bases de datos de las que se acaba de exponer son configurables, de forma que podrían pasar de una categoría a otra.

Elegir un tipo de base de datos u otra dependerá enteramente del problema a resolver. Hay que tener en cuenta las ventajas e inconvenientes de cada tipo de base de datos, de forma que se adapte lo máximo posible, no solo a nuestro modelo de datos, sino también a las características del servicio que se desea ofrecer.

3.2.1.1.2 ACID vs BASE

El mundo de las bases de datos relacionales está familiarizado con las transacciones ACID.

Las transacciones que se producen en el lenguaje SQL, sea cual sea el sistema gestor de base de datos cumplen siempre las propiedades ACID. Este tipo de transacciones se llaman así porque garantizan la **A**tomicidad, la **C**onsistencia, el aislamiento (**I**solation) y la **D**urabilidad.

- **Atomicity** (Atomicidad). Las transacciones han de ejecutarse por completo o no ejecutarse, pero la transacción no puede quedar a medias.
- **Consistency** (Consistencia o Integridad). Los datos que se guardan tras la transacción han de ser siempre datos validos.
- **Isolation** (Aislamiento). Las transacciones son independientes y no se afectan entre sí.
- **Durability** (Durabilidad). Una vez finalizada una operación, esta perdurará en el tiempo.

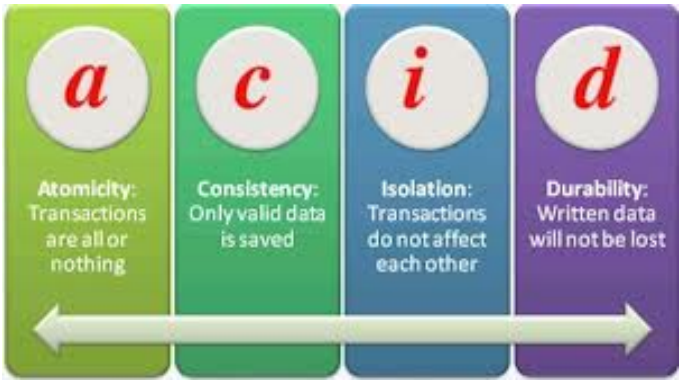

Ilustración 8. Transacciones ACID

El modelo BASE es un enfoque similar al ACID, aunque perdiendo la consistencia y al aislamiento, a favor de la disponibilidad, la degradación y el rendimiento.

El modelo **BASE** toma su nombre de:

- **Basic Availability**. El sistema funciona incluso cuando alguna parte falla, debido a que el almacenamiento sigue los principios de distribución y replicación.
- **Soft State**. Los nodos no tienen porque ser consistentes entre si todo el tiempo.
- **Eventual Consistency**. La consistencia se produce de forma eventual.

Para que un sistema gestor de bases de datos relacional pueda ser considerado tal, debe cumplir el modelo ACID.

Tabla 4. ACID vs BASE

ACID	BASE
- Fuerte coherencia - Aislamiento - Enfocado a commits - Transacciones anidadas - Más conservador - Evolución complicada (esquema)	- Coherencia débil - Disponibilidad - Mejor esfuerzo - Respuestas aproximadas - Más optimista - Más sencillo y rápido - Evolución más sencilla

Por otro lado, el modelo no relacional se ciñe más al enfoque BASE aquí expuesto, basándose en que las aplicaciones debe funcionar la mayoría del tiempo (disponibilidad, no tienen que ser siempre consistentes (soft state) y ser eventualmente coherentes (consistencia eventual).

3.2.1.1.3 Particionado

Suponiendo que los datos en los sistemas escalables superan la capacidad de almacenamiento de una sola maquina, y también que deben ser replicados para garantizar la fiabilidad y permitir el balanceo de carga, se deben planificar las tareas de particionado de datos.

Dependiendo del tamaño del sistema existen diferentes enfoques en este tema.

Caches de memoria como memcached se pueden ver como bases de datos en memoria directamente particionadas, de la forma en la que sacan los datos más frecuentemente accedidos a la memoria principal. De esta manera, son capaces de servir esta información más rápidamente a los clientes.

Hacer **clustering** de servidores es otro enfoque válido de particionamiento. La agrupación de servidores, y su modo de trabajar deben ser transparentes al cliente que realiza las peticiones, de forma que no debería notar que existen un grupo de servidores respondiendo las peticiones en lugar de una sola máquina.

Esta técnica es criticada ya que no es una solución que haya sido diseñada originalmente como base de datos no relacional, sino que ha sido pensada para funcionar sobre un sistema SGBD, mejorando de esta manera sustancialmente su rendimiento.

Separación de lecturas y escrituras. Esta técnica se logra manteniendo uno o más servidores dedicados para operaciones de lectura, así como varios servidores replica que ejercen de esclavos y atienden las peticiones de lectura. Si los maestros replican a los esclavos de forma asíncrona no tiene porque haber retardos en la escritura, pero si el maestro replica los datos de forma síncrona y sucede algún problema, puede haber pérdida de información. Además, si el maestro replica de forma síncrona, y por cualquier motivo existe retraso con cualquiera de los esclavos, las lecturas de esa información se deben retrasar todo el tiempo que sea necesario hasta que los datos estén disponibles en todos los esclavos.

Por norma general, la replicación maestro esclavo funciona bien si el ratio lectura / escritura es alto. La replicación de datos puede suceder tanto por transferencia de estado (copia de la información completa o de los deltas) como con el envío de las operaciones que se deben aplicar, atendiendo al estado de cada esclavo.

El **Sharding** se basa en particionar los datos de forma que la información que es frecuentemente accedida a la vez se encuentre en el mismo nodo. Además, se deberá tener en cuenta que la carga debe quedar distribuida entre todos los nodos del sistema de forma más o menos homogénea. Estos "shards" o fragmentos pueden ser, a su vez, replicados por razones de fiabilidad o balanceo de carga

3.2.1.1.4 Consistent Hashing

La idea de los hashes consistentes es bastante antigua y fue introducida por David Karger, en un paper sobre protocolos para cachés distribuidas (Karger, et al. 1997).

Esta idea surgió como solución a un problema que existe en las bases de datos basadas en clave valor. Durante el funcionamiento normal del sistema, el número de claves que se necesita almacenar, se van distribuyendo de forma más o menos equitativa entre todos los servidores que confirman en clúster de almacenamiento. El problema ocurre cuando es necesario añadir uno o más nodos al esquema y aproximadamente la totalidad de claves han de ser redistribuidas.

El consistent hashing viene a cubrir este problema de redistribución de claves, así tomar las claves de los nuevos servidores de los nodos de caché que antes las tenían asignadas, y no directamente de la base de datos principal.

Existen varias aproximaciones reales, varios enfoques distintos que parten de esta base y proponen una solución para minimizar el impacto de añadir o eliminar servidores de la ecuación, y para saber exactamente que servidor debe realizar las funciones de cacheo para la entrada que se está tratando.

Todas estas soluciones pasan por simular en forma de anillo el conjunto total de valores que pueden tomar nuestros hashes. Es decir, si se está empleando MD5 como función hash, el rango de posibles valores que puede tomar nuestra clave serán [0, 2^{160}-1], y se dispondrán en forma de anillo, es decir, los valores se repartirán de forma continua a lo largo de una línea, de tal manera que el valor 2^{160}-1 y el 0 sean consecutivos.

Ahora queda repartir los servidores por el anillo que se acaba de crear. Existen dos enfoques ampliamente utilizados para llevar a cabo esta tarea:

- **Marcar cada servidor con un único punto sobre el anillo** que se acaba de generar, de forma que, inicialmente, queden repartidos equitativamente entre las claves. En el caso de que haya que meter un nuevo servidor, habrá que calcular las claves que están contenidas entre cada servidor, para añadir el nuevo nodo donde más concentración de claves haya, y así "aliviar" la parte del clúster donde se concentren más "hotspots". Cuando un cliente tenga una clave y quiera resolver el servidor donde tiene que ir a preguntar, únicamente deberá resolver el hash que conforma su clave, ubicar ese punto sobre el anillo y recorrerlo en sentido horario hasta que encuentre un servidor. Ese será el nodo que contiene o debería contener el valor que está buscando.

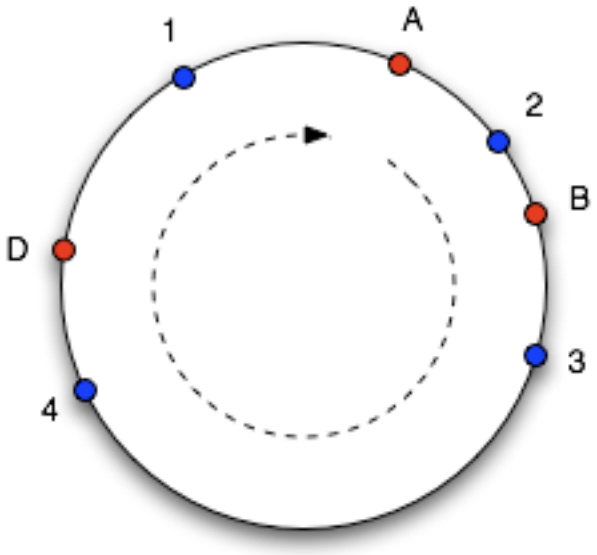

Ilustración 9. Consistent Hashing 1

- **El anillo generado se dividirá en particiones**. Se deberá elegir un número bastante elevado, puesto que será un numero que no cambiará nunca en la vida del clúster de almacenamiento. Cada nodo se dividirá en subnodos, y se hace

corresponder cada subnodo a una porción del anillo. Así cada nodo tendrá repartidos subnodos a lo largo de todo el anillo, y cada uno de esos subnodos se hará cargo de una porción del circulo, es decir, de un conjunto reducido de claves. Reducir el clúster en un nodo significa simplemente significa que el resto de nodos deben añadir subnodos en los huecos que quedan libres. Por otro lado, para añadir un nuevo nodo al clúster, los nodos que están ocupando el anillo deben ceder un número de subnodos al nuevo nodo que entra a formar parte del almacenamiento, dándole las claves y la administración de esas partes del anillo de hashes.

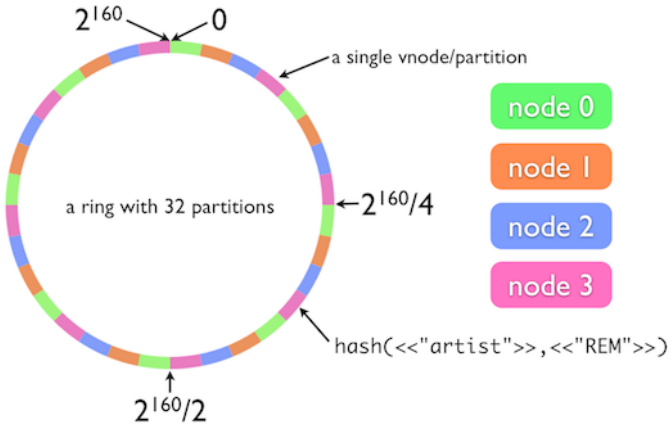

Ilustración 10. Consistent Hashing 2

3.2.1.1.5 Lectura y escritura sobre datos particionados

En un sistema con datos particionados en el que se introducen replicas de estas particiones, de forma que se incrementa la escalabilidad, se tiene que tener ciertas consideraciones si se quiere tener lecturas y escrituras consistentes.

Para empezar, tal vez sea buena idea anular el balanceo de carga en las operaciones de lectura cuando existen varias versiones del mismo conjunto de datos conviviendo en los nodos.

El equipo del Project Voldemort teoriza que existen tres parámetros importantes en las operaciones de lectura / escritura (Kreps 2010).

- **N**: El número de nodos replica que existen para los datos que se van a leer.
- **R**: El número de nodos de los que se va a leer.
- **W**: El número de nodos que se bloquean en una operación de escritura.

Si se cumple la fórmula **R + W > N**, y siempre que no haya errores en las operaciones de escritura, entonces se puede asegurar que se puede leer lo que se escribe de forma consistente.

3.2.1.1.6 Cambios de nodos

En un sistema en el que los nodos pueden caerse y ser reemplazados por otros miembros en su lugar, la comunicación en todo momento entre los nodos es importante (Ricky 2009).

Estas son las acciones que deben suceder cuando un nuevo nodo se une al sistema:

1. El nuevo nodo anuncia al resto de nodos su presencia mediante brodcast.
2. Los vecinos directos del nuevo nodo en el anillo reajustan sus límites para que el nuevo nodo toma control de su parcela.
3. El nuevo nodo copia los datos de la región que le ha sido asignada de los nodos que anteriormente tenían asignada esa zona.

De igual manera, cuando un miembro del clúster abandona el mismo, también se toman una serie de medidas.

1. Los vecinos se dan cuenta de que el nodo ha dejado de responder las peticiones de keep alive.
2. Los vecinos avisan al resto de nodos y se reparten y toman el control de las partes del anillo que tenía el nodo que ha caído.

3.2.1.2 Almacenamiento

La capa de almacenamiento depende en gran medida del tipo de base de datos que se esté tratando. En esta sección se dará una visión general de cómo almacena los datos cada modelo de base de datos.

3.2.1.2.1 Almacenamiento basado en filas.

Este almacenamiento se basa en juntar de forma secuencial todas las filas de una tabla y escribirlas juntas a disco. La ventaja de este tipo de almacenamiento es que si se están realizando operaciones de lectura / escritura de varias filas seguidas, la operación se realiza en un solo acceso a disco.

3.2.1.2.2 Almacenamiento basado en columnas.

Almacenamiento basado en la serialización de las columnas, y la escritura conjunta a disco. Por consiguiente, las operaciones que involucran accesos completos a los datos de las columnas quedan favorecidos. Por otro lado, se penalizan los accesos a información que se encuentra consecutiva dentro de una misma fila.

3.2.1.2.3 Arboles LSM (Log Structured Merge)

Los Arboles LSM son un tipo de estructuras que facilitan la organización y el almacenamiento de los sistemas de bases de datos, especialmente de aquellos que poseen una tasa muy alta de inserciones.

En los árboles LSM existen dos tipos de estructuras en forma de árbol. El primer componente, más pequeño, reside exclusivamente en memoria, mientras que el segundo persiste directamente a disco. El proceso consiste en que cada vez que hay que escribir un nuevo registro, este se escribe primero en el memoria, en el primer componente, de tal forma que cuando este primer componente alcanza un tamaño determinado, se realiza un volcado de una parte de los registros aquí contenidos a disco, al segundo componente, y se liberará la memoria que estos estaban ocupando.

Este tipo de almacenamiento de la información presenta ventajas muy interesantes, como el uso muy eficiente que hace de la memoria principal, por lo que es capaz de

responder peticiones muy rápidamente, además de optimizar el uso del disco de la máquina, haciendo volcados a disco sólo cuando es necesario para volcar una cantidad importante de información.

Por todo ello, es utilizado por sistemas gestores de bases de datos tan importantes cómo son SQLite, HBase o Cassandra.

3.2.1.3 Consultas

Existen varias diferencias con respecto a cómo los distintos tipos de bases de datos permiten a los usuarios / aplicaciones realizar consultas. Desde las consultas más básicas por clave primaria, como por ejemplo, los almacenes clave – valor, pasando por otros que ofrecen un acceso a la información algo más complejo. En este terreno se encontrarían las bases de datos documentales.

En general, la flexibilidad y la riqueza de las querys no son demasiado elevadas, puesto que lo que se prima por encima de las consultas es el rendimiento y la escalabilidad, por lo que es habitual que se delegue a la aplicación el implementar opciones más avanzadas en este terreno.

Un método de consulta llamado **Companion SQL database** consiste en tener una base de datos auxiliar (que puede ser una base de datos SQL o una TextDB) de forma que se utilice esta secundaria para almacenar ciertos metadatos importantes para realizar la búsqueda, y se empleen para facilitar la búsqueda posterior en el contenedor NoSQL.

Búsqueda local dispersa. Otra forma de realizar consultas consiste en, puesto que se tiene el conjunto de datos repartido entre los distintos servidores, repartir de igual modo la consulta, de forma que cada servidor ejecute localmente cada consulta y reenvíe los resultados a un nodo maestro, que sería el encargado de juntar todos los resultados y presentárselos a la aplicación.

Arboles B+ Distribuidos.

Una forma eficiente de acelerar las búsquedas consiste en mantener un árbol B+ que forme un índice de entradas a la base de datos NoSQL (Aquilera, Golab and Shah 2008).

El procedimiento consistiría en sacar los valores hash de los atributos que nos interese indexar, y construir con ellos el árbol B+. Cuando se quiera realizar una consulta, se comenzará desde la raíz y se irá descendiendo en orden hasta llegar a la hoja correspondiente, que nos dará la entrada concreta donde se encuentra el registro que se está buscando.

Al tratarse de un árbol B+, se debe tener en cuenta las particularidades de este tipo de estructura de datos a la hora de realizar los mantenimientos necesarios, las inserciones y borrados que puedan hacer redimensiones en el árbol, etc.

3.2.1.4 Procesamiento con MapReduce

En secciones anteriores se ha hecho una introducción al framework de programación y procesamiento en paralelo, Hadoop. Se ha expuesto también que, entre otras cosas, una de las principales características que posee este sistema de almacenamiento y procesado de grandes cantidades de información es que ha adoptado *MapReduce* como modelo de

programación para la plataforma, para dar soporte a la computación en paralelo de los datos almacenados en el sistema.

Pues bien, uno de los motivos por los que este modelo de programación cuadra tan bien con el modelo de Hadoop es porque está especialmente pensado para grandes cantidades de información. Las bases de datos NoSQL también comparten parte de este modelo de datos, por lo que el procesamiento con un framework basado en *MapReduce*.

Además, tanto la función *map* como la función *reduce* han de ser entendidas de una manera realmente funcional e independiente de la máquina donde se ejecuten, por tanto, ante la misma entrada, en el mismo entorno de ejecución, se producirá exactamente la misma salida. Esto significa que el particionado de la entrada en trozos más pequeños y el reparto de éstos a través del clúster, no afectará al resultado.

Por último, juntando algunos conceptos vistos anteriormente, se puede utilizar el Consistent Hashing como forma de orquestación para el paradigma *MapReduce*. Así, se puede repartir el conjunto de datos de la entrada de igual modo que se hace con el Consistent Hashing, por lo que los nodos comenzarán a procesar los datos en sentido horario. Después, la muestra de información intermedia volverá a ser consistentemente hasheada para que los nodos finalicen con la segunda parte del proceso, el *reduce*, y produzcan el resultado final.

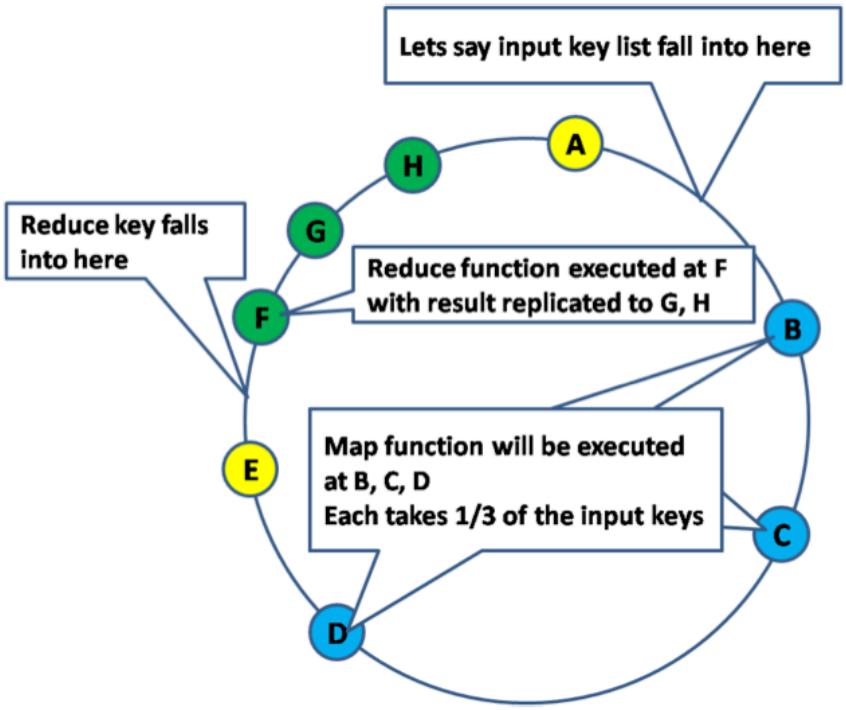

Ilustración 11. Ejemplo de *MapReduce* con Consistent Hashing

4 Bases de datos clave-valor

Las bases de datos clave – valor será el primer tipo de bases de datos NoSQL que se abordará en este estudio. Cuando se habla de bases de datos clave – valor se hace referencia a DynamoDB y a Redis (expuestas a continuación), y también a Bigtable, de Google, a Voldemort por parte de Linkedin, a Tokyo Cabinet y a muchas otras.

Como se verá en esta sección, este tipo de base de datos posee la estructura de datos más sencilla, lo cual proporcionará ventajas a la hora de particionar y escalar el sistema a lo largo de clústers de decenas e incluso cientos de nodos.

5 DynamoDB

5.1 Introducción

Amazon DynamoDB es un SGBD NoSQL distribuido multi maestro con modelo de datos clave valor. Pero es un SGBD especial, puesto que lo que Amazon ofrece realmente es el servicio de base de datos de manera totalmente gestionado (Amazon 2012)

DynamoDB es un SGBD propietaria desarrollada internamente por Amazon y lanzada allá por 2007, que se incluye dentro de la oferta de servicios que se ofrecen como parte de los AWS (Amazon Web Services).

5.2 Características principales

El servicio de Amazon permite únicamente crear nuevas tablas en las que alojar los datos, e interactuar con ellas mediante tus aplicaciones a través de APIs (interfaces de programación que proporciona Amazon para poder añadir, actualizar, borrar y consultar información de la base de datos). Es decir, no se tiene acceso al código fuente, ni al hardware ni a los servicios que hacen funcionar a la base de datos.

5.2.1 Aprovisionado

Que la base de datos sea totalmente gestionada significa que Amazon se encarga de toda la configuración y el mantenimiento de la infraestructura física, mientras que el usuario debe encargarse únicamente del aprovisionado de la base de datos, entendiendo por "aprovisionado" la especificación mediante una interfaz de los umbrales de peticiones de lectura / escritura que debe alcanzar cada una de las tablas.

DynamoDB es capaz de repartir los datos y el tráfico a través de múltiples servidores de forma que se puedan cubrir las necesidades de aprovisionado que realiza el usuario. Es precisamente estas necesidades las que son facturadas a posteriori por Amazon. Por ejemplo, un administrador estima que una tabla concreta va a recibir un promedio de 10 peticiones de lectura y 2 de escritura por segundo, por lo que las labores de aprovisionado de la base de datos consistirá en indicar estas cifras a DynamoDB. Ésta, a su vez, se encargará de que el sistema esté preparado para recibir la carga que se le ha especificado.

Pero ¿qué ocurre si el administrador se equivoca e indica números más bajos de los que realmente se producen? En caso de que el sistema reciba un nivel de carga mayor al indicado en el aprovisionado, el sistema comenzará a degradar su nivel de servicio

progresivamente. Será necesario una intervención manual que incremente los umbrales de peticiones que la tabla es capaz de soportar. Si por ejemplo, nuestra aplicación va a tener que realizar 100 lecturas y 20 escrituras por segundo, habrá que indicárselo a DynamoDB para que gestione sus recursos y permita soportar esa carga.

Además, DynamoDB es capaz de realizar este cambio de configuraciones (lo que se conoce como escalar hacia arriba o hacia abajo) sin ningún tipo de pérdida de servicio, es decir, aceptando el 100% de las peticiones que se envían, incluso durante un cambio de valores del aprovisionado.

5.2.2 Regiones y Zonas de Disponibilidad

Amazon ofrece sus servicios a nivel mundial. Esto lo consigue teniendo CPDs en países de todos los continentes. A día de hoy, la oferta de ubicaciones geográficas de los AWS es la siguiente:

Tabla 5. Regiones y Zonas de Disponibilidad Globales de Amazon

Continente	Región	Zonas de Disponibilidad
América del Norte	EEUU Este (Virginia)	5
	EEUU Oeste (Oregón)	3
	EEUU Oeste(California)	3
	AWS GovCloud	2
América del Sur	Sao Paulo	2
Europa	Irlanda	3
Asia Pacífico	Singapur	2
	Sydney	2
	Tokyo	3

Amazon cuenta con un total de **9 regiones**. Una región es una ubicación auto contenida para tus datos. Amazon asegura que la información que alojes en una de las regiones no saldrá de ella.

Cada una de las regiones cuenta con un número de zonas de disponibilidad (ZdD), que habitualmente varía entre 2 y 3. Estas zonas de disponibilidad físicamente corresponden con CPDs geográficamente alejados entre sí, de manera que un corte de suministro o un aislamiento total o parcial de un CPD no afecte a los demás. Algunos servicios, como

por ejemplo EC2, permiten elegir en que zona de disponibilidad funcionan sus servicios. Otros, como es el caso de DynamoDB, son servicios que funcionan a nivel de región, haciendo uso de todas las zonas de disponibilidad dentro de la misma región. De esta manera, si existe algún problema con cualquiera de las ZdD, el servicio es capaz de seguir funcionando en las demás.

Amazon DynamoDB almacena sus datos en discos de estado sólido SSD que se distribuyen a lo largo de todas las regiones y zonas de disponibilidad propias de la arquitectura de Amazon (Amazon Web Services 2013), lo que mejora la latencia y reduce los tiempos de acceso a la información.

5.2.3 Influencias

Desde el inicio del movimiento NoSQL, muchas han sido las bases de datos clave valor que han ido saliendo a lo largo del tiempo. Y muchas de ellas han sido fuertemente influenciadas por DynamoDB, como por ejemplo Cassandra (Lakshman and Malik 2009) y Riak. (Basho Technologies 2013)

5.3 Arquitectura

DynamoDB ha sido implementado como un sistema particionado con replicación y ventanas muy concretas de consistencia (DeCandia, et al. 2007). Está hecho de esta manera para poder ofrecer una alta disponibilidad a cambio de una consistencia eventual.

Además, DynamoDB ha sido diseñado para estar **siempre disponible para escribir**, es decir, en cualquier momento en cualquier nodo se puede realizar una escritura en base de datos. Ya se encargará más adelante el sistema en hacer consistente esa lectura con el resto de nodos.

El diseño de la base de datos ha sido pensando en un esquema de replicación que incluso permita escribir teniendo errores de comunicaciones entre las máquinas, **dejando la consistencia inmediata entre servidores en un segundo plano**. Por eso es que se dice que DynamoDB es una base de datos altamente disponible para escrituras.

DynamoDB implementa una parte muy básica de resolución de conflictos. Para este apartado concreto, ha de ser la aplicación la que gestione todo lo relacionado con la capa de negocio, la que se hará cargo del modelo de datos y la que tendrá que decidir, llegado el caso, qué método de resolución de conflictos es más apropiado para cada caso en particular.

Por otro lado, en DynamoDB todos los nodos son exactamente iguales, todos poseen el mismo rol y desempeñan las mismas funcionalidades. Su diseño favorece las pequeñas sincronizaciones “peer to peer” entre nodos, en lugar de disponer de un servidor o un grupo de servidores que hagan la labor de “maestros” centralizados del clúster.

DynamoDB tiene en cuenta que las máquinas que conforman su sistema tienen un hardware heterogéneo, y por lo tanto, tiene que saber gestionarlos en función de las capacidades de trabajo que tiene cada uno.

Puesto que los servidores que sostienen Dynamo operan directamente desde los servidores de la infraestructura de Amazon, tanto el entorno como los nodos son

considerados elementos de confianza, por lo que no tienen ningún mecanismo de seguridad implementado a esos niveles.

5.3.1 Interfaces del sistema

DynamoDB proporciona una serie de operaciones hacia el exterior (Amazon Web Services 2012), entre las cuales están las correspondientes a recuperar e insertar o actualizar nuevos elementos:

- `GetItem ( clave )`

La operación `get` sirve para, dada una clave, obtener el valor asociado a la misma. Puede devolver más de un resultado si existieran varias versiones de la misma información. En ese caso, en lugar de devolver un único resultado devolvería una lista de objetos, que se corresponderían con los valores asociados a esa clave.

Otro dato importante que devuelve es el contexto, que será necesario después en la operación de escritura.

- `PutItem ( clave, valor )`

El método `putItem` determina, en función de la clave, donde alojar el valor pasado por parámetro.

Todas estas invocaciones al API de DynamoDB viajan por http(s) y están codificadas en formato JSON.

5.3.2 Algoritmo de particionado

DynamoDB confía en el Consistent Hashing como método para distribuir la carga entre múltiples nodos de sistema (DeCandia, et al. 2007). Esto es así debido a que una de los puntos fuertes del diseño de DynamoDB es que debe escalar incrementalmente, por lo que se requiere de un mecanismo que sea capaz de particional los datos de forma dinámica entre el conjunto de máquinas que haya disponibles en cada momento en cada uno de los clústers.

Puesto que el algoritmo de particionado ya ha sido ya explicado con anterioridad en este mismo documento, no se entrará en detalles sobre qué es o cómo funciona el Consistent Hashing. Pero sí vale la pena destacar algunas particularidades que Amazon ha realizado sobre el método original (Strauch 2011).

Los dos principales problemas que encuentra Amazon en el algoritmo del Consisten Hashing son:

1. Las asignaciones aleatorias que realiza el algoritmo para cada nodo del anillo, ya que desembocan en una distribución desigual tanto de la carga como de los datos
2. Este algoritmo de particionado no tiene en cuenta la naturaleza heterogénea de los nodos, de forma que se varíe la carga de trabajo que se asigna a cada nodo de cara al rendimiento que éste puede proporcionar al sistema.

Para solucionar estas carencias, Amazon ha creado su propia variante del Consistent Hashing. Amazon crea el concepto de "**nodo virtual**". Esto es, que en lugar de mapear

sobre el anillo el conjunto de máquinas que se tienen disponibles, se repartirían los nodos virtuales. Un nodo virtual es como un nodo más del sistema, solo que cada máquina sería responsable de uno o más de estos nodos virtuales.

El número de nodos virtuales que cada máquina es capaz de asimilar es directamente proporcional a las características físicas de su hardware.

5.3.3 Replicación

Para lograr la alta disponibilidad y la durabilidad que son necesarias en un sistema que roza el 100% de disponibilidad, DynamoDB utiliza replicación de datos entre nodos, más concretamente, entre nodos consecutivos.

El mismo esquema de Consistent Hashing que se utiliza para particionar los datos del sistema también es utilizado a la hora de hacer la replicación.

Cada rango de claves están coordinadas por un nodo. Estos coordinadores están al cargo de la replicación de todo el rango, y lo harán a las N instancias siguientes, donde N es un número positivo parametrizable por instancia. Habitualmente se pone 3 como número por defecto. De esta manera, una clave k que se almacene en DynamoDB elegirá los siguientes N-1 nodos en sentido horario, y serán esos los nodos virtuales que tendrán almacenada una copia del valor.

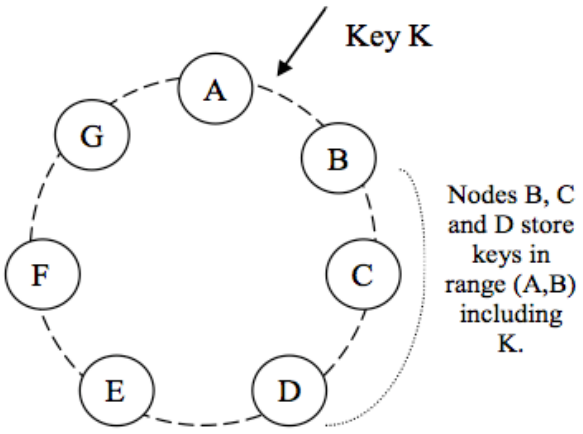

Ilustración 12. Replicación de los N-1 nodos virtuales consecutivos

5.3.4 Versionado de Datos

En DynamoDB, cada vez que se agrega un nuevo dato o se actualiza uno ya existente, esta modificación se realiza únicamente en el nodo donde está realizando la escritura. Este nodo, posteriormente, informará a los nodos restantes del nuevo estado de ese dato concreto, pero lo hará de forma asíncrona. Así, pueden darse casos en los que un método `putItem()` realiza una modificación de un dato de la base de datos, y consecuentes `getItem()` sigan devolviendo el valor desactualizado durante un

tiempo, hasta que finalmente se propaguen estos cambios. Como se ha comentado anteriormente, prioriza escritura sobre consulta de datos.

Sin embargo, existen ciertos escenarios con los que hay que tener especial consideración. Errores en la red, o problemas de comunicación pueden aislar una parte de la red y parecer que los datos están en su última versión pero realmente no ser así. Para evitar esto, DynamoDB permite que varias versiones del mismo objeto estén conviviendo simultáneamente en el clúster. Cada vez que se actualiza un objeto, DynamoDB crea una nueva versión inmutable de los datos. De esta manera, el propio sistema, aunque tenga actualizaciones retardadas, al final siempre será capaz de discernir cual es la información más actualizada y cual la antigua.

Sin embargo, debido a la naturaleza distribuida del sistema, es posible que ciertas actualizaciones o ciertos errores de comunicación entre partes del clúster, generen versiones conflictivas de los mismos datos. En ese caso, cuando la base de datos no es capaz de discernir qué datos deben ser los más actuales y cuáles los que deben ser relegados, es necesario intervención del cliente o de la aplicación que está realizando las modificaciones en la base de datos.

Es decir, es la aplicación externa creada por el usuario y que hace uso de los recursos de DynamoDB la que debe ser capaz de resolver explícitamente que convivan diversas versiones de la misma información, marcadas como versiones conflictivas entre sí.

DynamoDB utiliza lo que se conoce como *Vector Clocks* (Lamport 1978) para aplicar causalidad entre las diferentes versiones del mismo objeto.

En la Ilustración 12, un cliente primero crea un objeto y la petición de actualización es manejada por el host Sx, de forma que el Vector Clock queda asociado con él ([Sx, 1]).

A continuación, el cliente actualiza la información, cambio que es gestionado de igual manera por el nodo Sx. Como resultado, se genera el Vector Clock ([Sx, 2]).

De esta manera, se puede establecer una relación de causalidad entre el primer suceso y el segundo: el host que gestiona el cambio es el mismo, y se puede percibir un contador ascendente positivo que nos dice qué evento sucede primero y cual sucede a continuación. Incluso, puesto que ([Sx, 2]) es el sucesor único de ([Sx, 1]), se puede eliminar esté último del Vector Clock.

Ahora un cliente solicita una actualización partiendo de la versión 2 de los datos, y el host que gestiona el cambio será Sy. Un nuevo Vector Clock es generado con la versión D3: ([Sx, 2],[Sy, 1]). Aquí no se podrá eliminar ([Sx, 2]) puesto que los datos aun no han sido replicados entre todos los nodos afectados, y por el momento es necesario a la hora de reconstruir la cadena de causalidad.

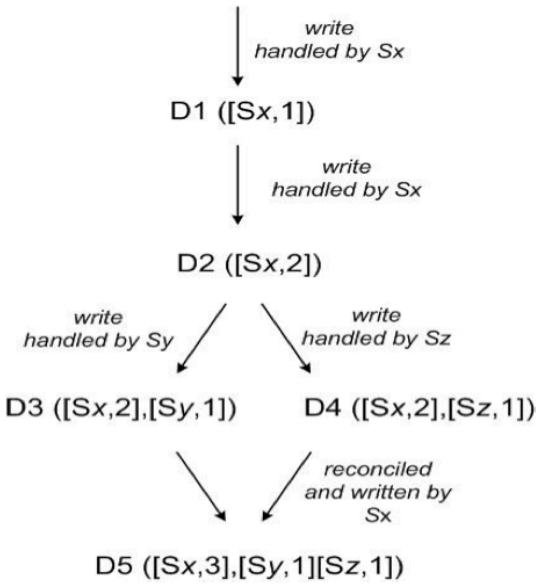

Ilustración 13. Esquema de la evolución de un objeto a lo largo del tiempo

Lo mismo sucede con la siguiente actualización. Otro cliente accede a otro nodo, esta vez el Sz y modifica el objeto partiendo de su versión ([Sx, 2]). ¿Porque no parte de la versión más actual en este momento, esto es, la versión D4 ([Sx, 2],[Sy, 1])?. Pues porque el nodo Sz desconocer que en otro lado del clúster, este mismo fichero ha sido modificado y hay una versión superior a la ([Sx, 2]).

El resultado es la versión marcada por el Vector Clock ([Sx, 2], [Sz, 1]).

A continuación, un cliente lee el valor del objeto, y le es enviado el valor en conflicto, es decir, la combinación de las versiones conflictivas D3 y D4 como combinación de ambos Vector Clocks, y que no reflejan una sucesión limpia de actualizaciones.

Antes de que se pueda realizar cualquier otro cambio sobre esta información, el software que hace las funciones de cliente de la base de datos debe resolver el conflicto. En el caso, por ejemplo, que lo hace desde el nodo Sx. Una vez que finalice el proceso, el nuevo valor que tendría la nueva versión del objeto que se está tratando sería la marcada en D5, es decir, ([Sx, 3],[Sy, 1],[Sz, 1]).

Ante el problema de las distintas versiones, mantener un versionado empleando los Vector Clocks es, en general, una buena idea. Ahora bien, en los sistemas con multitud de escrituras por segundo, es frecuente que las distintas versiones de los objetos, así como las sucesiones de Vector Clocks crezcan de manera desmesurada. Para ello, DynamoDB permite implementar un sistema de truncado de la cadena de versiones, al cual se le especifica un número máximo de versiones que se desea almacenar, y cuando llega al límite, elimina la versión más antigua de la que tenga constancia.

5.3.5 Ejecución de operaciones `getItem()` y `putItem()`

DynamoDB permite que cualquier nodo del clúster pueda responder peticiones `getItem` y `putItem`, sin importar la clave solicitada, sin importar si el nodo recibe la petición aloja físicamente el dato o no.

Existen principalmente dos maneras de poder contactar con los servidores de almacenamiento de DynamoDB: bien mediante encaminadores o balanceadores genéricos, de forma que cada vez que se realice una petición se resuelva el nodo que atenderá siguiendo algún método de balanceo, bien mediante alguna librería que tenga en cuenta la manera en la que DynamoDB tiene repartidos los datos, y nos pueda poner en contacto con algún servidor que tenga los datos que se le están pidiendo al clúster. En el primer caso, el reparto de carga es más equilibrado y el sistema es independiente por completo del clúster de DynamoDB, y en el segundo caso se obtiene una latencia y unos tiempos de acceso menores.

Para proporcionar una visión de coherencia general a las aplicaciones cliente, DynamoDB aplica un protocolo de coherencia basado en *quórum* (método por el cual un conjunto de participantes votan o deciden por una cuestión, en este caso, cada nodo miembro del clúster será un miembro del *quórum*), que contiene principalmente dos valores configurables, R y W. R representa el número mínimo de nodos que debe participar en una operación de lectura exitosa, y W será el número mínimo de nodos que deben participar en una operación de escritura exitosa.

Idealmente, para poder formar el *quórum*, $R + W > N$. Pero hay que tener en cuenta que, en este caso, el nodo más lento en responder será el nodo que marque la respuesta de todo el *quórum*, por lo que habitualmente el *quórum* se configura de tal forma que $R + W < N$.

La modificación de estos valores puede ayudar en según qué situaciones. Por ejemplo, En los casos en los que se necesite un elevado rendimiento de lectura pero una elevada consistencia en escritura, podrían configurarse los valores del *quórum* con R=1 y W=N.

Por otro lado, mantener ambos valores en un umbral excesivamente bajo, puede incrementar el riesgo de inconsistencias, permitiendo a los clientes escribir y sobrescribir objetos antes de que a los nodos les dé tiempo a replicar toda la información que están procesando.

Las aplicaciones que hacen uso de DynamoDB pueden sobrescribir los valores N, R y W para poder conseguir el rendimiento que desean en cada caso.

5.3.6 Gestión de errores

Para gestionar errores en el sistema provocados por indisponibilidad de nodos, DynamoDB no realiza ningún control ni toma ninguna medida exhaustiva, al contrario. El enfoque que toma es el de delegar la lectura o escritura de la operación a N nodos, más concretamente los N primeros nodos en buen estado que encuentre, y si todos devuelven el mismo resultado, tomará por buena la operación. Estos nodos no tienen por qué ser los primeros que encuentre en sentido horario siguiendo el anillo marcado por el Consistent Hashing.

Otra medida que la base de datos toma cuando no se puede escribir en un nodo concreto del clúster debido a que no responde correctamente, es el de designar un nodo encargado que se encuentre en disposición de realizar escrituras. Este nodo se hará cargo de la actualización de los datos, y tan pronto como el nodo original vuelva a estar disponible, se le hará llegar la actualización.

Un caso interesante es el del fallo total, o situaciones de error general a nivel de CPD (Centro de Proceso de Datos). DynamoDB ha sido configurado también para asegurar el funcionamiento incluso cuando se dan este tipo de fallos. Básicamente, en la configuración de los clústers de DynamoDB viene predeterminado la inclusión de nodos en distintos CPDs, incluso en distintas zonas de disponibilidad (grupos de CPDs). Los CPDs de Amazon están conectados por líneas de alta velocidad, por lo que las comunicaciones, replicación, etc. no implican un retardo importante en tiempos de respuesta.

Finalmente, para gestionar los casos de inconsistencias entre replicas, DynamoDB utiliza árboles Merkle. Esto, como se verá en la siguiente sección, permite minimizar la cantidad de datos que se transfieren en el sistema.

5.3.6.1 Arboles Merkle

Una inconsistencia se produce cuando un nodo posee nuevos datos que aun no ha enviado a otro, o cuando un nodo tiene un dato más actualizado que otro nodo. Es deseable resolver las inconsistencias lo antes posible para que el sistema devuelva siempre la información más actualizada posible.

Los arboles Merkle son una estructura de datos que le permite a DynamoDB comprobar cuándo dos nodos del clúster están sincronizados y cuando no.

Un árbol Merkle es un árbol hash que cada nodo genera y actualiza, cuyas hojas mantienen los hashes de cada una de las entradas individuales de la base de datos. Los nodos que no son nodos hoja, a su vez, mantienen actualizado un hash de sus respectivos hijos.

De esta forma, es sencillo comprobar los nodos raíz de dos árboles. Si son iguales, ambos nodos contendrán exactamente la misma información, por lo tanto, estarán sincronizados. Además, mediante este mismo procedimiento, es sencillo comparar ramas completas de distintos árboles sin tener que descargar los datos que contienen dichas ramas en la base de datos ni en el propio árbol.

Los árboles Merkle reducen considerablemente la cantidad de datos que son transferidos por operaciones de sincronización, así como las lecturas a disco.

En DynamoDB, cada nodo de almacenamiento mantiene un árbol Merkle por cada rango de claves, esto es, por cada una de las divisiones creadas en el anillo del Consistent Hashing. Además, será responsable tanto de mantenerlo actualizado como de compararlo con el resto de nodos que se estén encargando de ese mismo rango de claves.

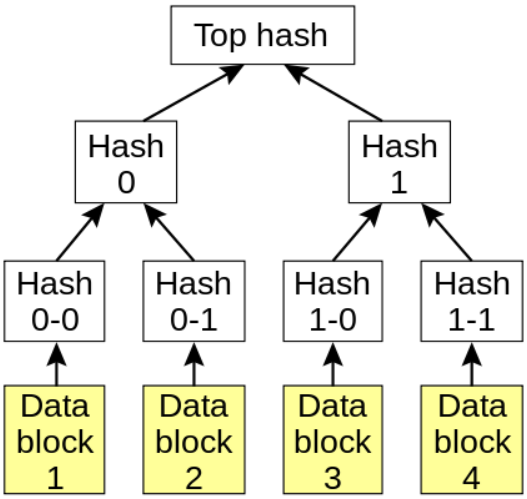

Ilustración 14. Estructura de un Árbol Merkle

5.3.7 Gestión de Nodos

DynamoDB implementa un mecanismo explícito para agregar y eliminar nodos al sistema. Es decir, no existe ningún mecanismo implícito que detecto y/o sustituya nodos inaccesibles, recalcule rangos del Consistent Hashing, etc.

Sin embargo, en DynamoDB sí que se pueden gestionar los nodos problemáticos de forma explícita. Amazon proporciona herramientas a los administradores de sistemas al cargo para que puedan agregar y eliminar nodos cuando consideren.

Cuando un nodo arranca por primera vez, escoge los nodos virtuales de los que se va a hacer cargo dentro del anillo del Consistent Hashing. El nuevo nodo persistirá estos rangos de claves que de los que se va a hacer cargo, e irá informando al resto de nodos de estas nuevas asignaciones. El número de nodos virtuales de los que se elegirá hacerse cargo tendrá que ver con la configuración hardware con la que haya sido creado.

Agregar nodos al clúster siempre cambia los propietarios de ciertos rangos de claves. Los nodos que eran propietarios de estos rangos de claves, ahora son notificados mediante el protocolo de propagación de miembros de que han perdido la responsabilidad de esos rangos.

5.4 Modelo de Datos

El modelo de datos propio de DynamoDB el siguiente:

5.4.1 Tablas

Las tablas son, al igual que en muchos SGBDs, los elementos donde se almacenan las filas o elementos donde se guardan los datos propiamente dichos de la base de datos.

Las tablas en DynamoDB son libres de esquema, esto es, que dos datos de una misma tabla no tienen porque tener los mismos atributos, ni siquiera el mismo número de atributos.

Lo que sí deben tener todos los elementos de una tabla es un atributo obligatorio que haga las funciones de clave primaria, puesto que esta clave identifica unívocamente a cada elemento dentro de los elementos de la tabla.

Cada tabla puede contener un número infinito de elementos.

5.4.2 Elementos

Un elemento es información con una estructura que se almacena en la base de datos. Los elementos son los homónimos de las filas en el modelo relacional. Cada elemento en DynamoDB es el resultado de proporcionar valores a un conjunto de atributos que se definen en el momento de la inserción.

Cada elemento, en conjunto con sus atributos, tiene una restricción de tamaño máximo de 64KB.

5.4.3 Atributos

Los atributos son pares clave – valor o asociaciones de clave – conjunto de valores. Estos atributos no tienen restricción de tamaño máximo.

Tabla 6. Ejemplo de tabla en DynamoDB[48]

ID (clave primaria)	Atributos

101	{
	Title = "Book 101 Title"
	ISBN = "111-1111111111"
	Authors = "Author 1"
	Price = -2
	Dimensions = "8.5 x 11.0 x 0.5"
	PageCount = 500
	InPublication = 1
	ProductCategory = "Book"
	}

201	{
	Title = "18-Bicycle 201"
	Description = "201 description"
	BicycleType = "Road"
	Brand = "Brand-Company A"
	Price = 100
	Gender = "M"
	Color = ["Red", "Black"]
	ProductCategory = "Bike"
	}

5.5 Implementación y Optimizaciones

Los nodos de DynamoDB tienen principalmente los siguientes elementos software:

---

48 Ejemplo extraído de http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html

1. Motor de persistencia local
2. Coordinador de peticiones
3. Detector de miembros y fallos
4. Controlador de Admisión

Todos estos componentes están escritos en Java.

1. El motor de persistencia está preparado para gestionar diferentes formas de almacenar la información. Es capaz de alojarla tanto en otro tipo de bases de datos: MySQL, BDB (Berkeley Database) o BDB Java Edition, así como mantener la información directamente sobre la memoria principal en buffers con un sistema de persistencia a disco cada cierto tiempo.

A pesar de que la mayoría de instancias de producción utilizan Berkeley DB, la librería de manejo de base de datos, como componente de persistencia, la realidad es que DynamoDB se ha diseñado de forma que se le pueda conectar uno u otro sistema, de forma que este componente de almacenamiento se pueda adaptar lo máximo posible a los patrones de utilización de la aplicación.

2. El coordinador de peticiones está construido sobre una pieza de intercambio de mensajes basado en eventos. Todas las comunicaciones entre nodos se realizan empleando las librerías Java NIO.

En el momento en el que un cliente lanza una petición de lectura o de escritura, éste abre una vía de comunicación con uno de los nodos del clúster, el cual hará la función de coordinador entre el cliente y el clúster. El nodo crea una máquina de estados a partir de la cual será capaz de dirigir al cliente hasta el nodo que es responsable del conjunto de claves que ha pedido leer o escribir el cliente. Cada máquina de estados manejará una y solo una petición.

Si un coordinador de peticiones, ante una petición de cliente recibe un valor obsoleto de un objeto, como parte del proceso de devolverle la petición al cliente, actualizará el valor del objeto en el nodo que lo tiene desactualizado. A esto se le llama *read-repair* (lectura y reparación).

3. Una forma de optimización que se puede lograr a nivel de latencia, tanto para lecturas como para escrituras consiste en que sean los clientes los que sean los coordinadores de las operaciones. Es decir, la máquina de estados que es creada por el coordinador de peticiones es alojada localmente en la parte del cliente.

Para obtener información más detallada del estado del clúster, de vez en cuando el cliente lanzará peticiones a nodos aleatorios del clúster. De esta manera, el cliente va siendo capaz de determinar qué nodos están en posesión de qué rangos de claves, para reducir accesos y acudir a preguntar directamente a los nodos encargados en cada caso de las claves que correspondan.

Habitualmente, las peticiones de escritura que finalizan con éxito suelen derivar en operaciones de lectura satisfactorias, especialmente en un pequeño espacio de tiempo. DynamoDB implementa una optimización en este sentido, y es que el nodo que responda más rápido ante una petición de lectura será reenviado directamente al cliente, de forma que posteriores peticiones de escritura sean lanzadas directamente contra él.

Así, se podrá asegurar que el que está recibiendo las operaciones de escritura es exactamente el nodo que está al cargo del rango de claves que nos afecta, y se consigue una consistencia muy elevada cuando alguien lee lo que se escribe en ese nodo.

Aparte de toda la funcionalidad que ya se ha comentado, también existen en los nodos una serie de tareas que se ejecutan en segundo plano. Estas tareas son directamente gestionadas por otro componente llamado **controlador de admisión.**

Este controlador se encarga básicamente de monitorizar el funcionamiento a nivel de hardware del sistema: impacto de las operaciones `get / put` en el rendimiento de las máquinas, latencias de entrada / salida a disco, bloqueos, timeouts, etc.

A través de estos sistemas de monitorización que incorpora el controlador será el encargado de ir asignando rodajas de tiempo para acceder a los recursos y los tiempos de utilización de los mismos por parte de dichas tareas que suceden en segundo plano.

5.6 Ejemplos reales de uso del SGBD

En esta sección se expondrán casos reales de uso de la base de datos. El sitio web DynamoDB Testimonials (DynamoDB 2014) alberga una gran cantidad de casos de éxito de esta base de datos. Entre ellos se puede destacar.

5.6.1 IMDB

El **IMDB** (Las siglas de Internet Movie DataBase) es el sitio de internet líder en información sobre películas y series, manteniendo información actualizada también de actores, personal de producción, videojuegos, y demás áreas relacionadas con el medio audiovisual. Actualmente, es propiedad de Amazon.

El IMDB tiene más de **110 millones de usuarios** únicos al mes, y aparte de albergar todo tipo de información relacionada con películas, aloja una gran cantidad de material audiovisual, como fotografías y trailers.

IMDB migró su sistema de puntuaciones (un sistema mediante el cual los usuarios puede valorar en una escala del 1 al 10 las películas que decidan) a DynamoDB.

Según cuenta H.B.Siegel, director técnico de IMBD: "*A fin de garantizar que podemos ajustar la escala con rapidez, hemos migrado el sistema de evaluación popular de 10 estrellas de IMDb a DynamoDB. Hemos evaluado varias soluciones tecnológicas y hemos elegido DynamoDB porque es un sistema de bases de datos de alto rendimiento que ofrece una escalabilidad perfecta y que está totalmente gestionado. Esto nos ahorra un montón de tiempo en tareas de desarrollo y nos permite centrar nuestros recursos en crear mejores productos para nuestros clientes, mientras mantenemos nuestra confianza en la capacidad de gestionar el crecimiento que experimentamos*".

5.6.2 SmugMug

SmugMug es una empresa que permite a los usuarios y a las empresas crearse su propio portfolio fotográfico y de videos. Además, tiene muchas opciones avanzadas como crear un sitio web a para las fotografías, crear galerías, obtener estadísticas de acceso a las fotografías, etc.

En palabras de Don MacAskill, director general del SmugMug: "*Me encanta que DynamoDB nos permita ofrecer el rendimiento deseado, así como una baja latencia y una escalabilidad perfecta, incluso con nuestras cargas de trabajo sujetas a un crecimiento constante. Aunque nos avalan años de experiencia con arquitecturas grandes y complejas, nos complace poder dejar por fin de tener que gestionarlas por nuestra cuenta y poder utilizar DynamoDB para obtener incluso mayor rendimiento y escalabilidad de los que podríamos obtener nosotros*"

6 Redis

6.1 Introducción

Redis es un Sistema Gestor de Bases de Datos de base de datos en memoria de conjuntos clave – valor, que además permite persistir los datos que residen en memoria a disco.

El origen del nombre de la base de datos, Redis, es el acrónimo de **RE**mote **DI**ctionary **S**erver. Redis funciona de forma similar a como lo hacen el resto de los SGBD clave - valor, presentando una serie de ventajas sobre éstas.

Memcached es un tipo de base de datos similar en el sentido de que es un sistema de almacenamiento clave – valor, que almacena datos únicamente en memoria. Esto significa que cuando un nodo que tiene información del clúster almacenada se apaga, esta información se pierde.

Redis, a diferencia de Memcached, extiende la funcionalidad de éste permitiendo persistir los datos que aloja a disco, de tal manera que si un nodo se apaga, siempre podría recuperar la última información que haya escrito previamente a disco. Otra de las funcionalidades comunes para las que se emplea esta base de datos es la de gestionar su almacenamiento como si fuera un gestor de colas.

Un **gestor de colas** es un sistema que permite controlar y planificar el uso de recursos de un sistema. Por ejemplo, en el caso de una aplicación muy pesada como pueda ser un servicio que grabe videos y los procese: los redimensione, realice copias, etc. Este proceso puede llevar minutos o incluso horas. Mediante un gestor de colas, el agente encargado de grabar los videos encolarían los trabajos una vez filmados y los servidores dedicados a realizar el trabajo irían abordándolos a medida que van quedando libres.

Redis es utilizado por algunas de las empresas más importantes del panorama actual de internet, como Twitter, GitHub, Weibo, Digg, Hulu, StackOverflow, Instagram o Tumblr, y forma parte de muchos proyectos de software libre como una pieza que aporta un componente de colas o persistencia en, por ejemplo, el gestor de logs centralizado Logstash[49] o el sistema de monitorización Sensu[50].

6.2 Características principales

6.2.1 Tipos de datos

Redis es capaz de reconocer y manejar cinco tipos de datos diferentes, lo cual es una ventaja con respecto a otras bases de datos que únicamente trabajan con cadenas de caracteres (*String*). Esto le da ventaja en lo que a manejo de los datos y almacenamiento de los mismos se refiere.

Los tipos de datos son:

- String

[49] http://logstash.net/docs/1.4.1/tutorials/getting-started-with-logstash
[50] http://sensuapp.org/docs/0.12/overview

- Set
- Hash
- List
- Sorted Set

En la sección Modelo de Datos en la que se describe el modelo de datos con mayor profundidad, se expondrán distintos ejemplos de utilización de estos tipos de datos.

Poder manejar varios tipos de datos de forma nativa otorga al sistema diversas ventajas. Cada uno de ellos ofrece una serie de métodos para acceder y manipular la información que almacena. Dependiendo del tipo de uso que se le vaya a hacer a esos datos. Redis da la opción de elegir que estructura se adapta mejor a las necesidades.

6.2.2 Usos del SGBD

Redis es un SGBD con multitud de aplicaciones prácticas. Se puede hacer uso de sus distintos tipos de datos para gestionar un ranking, listas de contactos, un sistema de puntuaciones, etc. Se puede utilizar como sistema de cachés puro en memoria sin ningún tipo de opción de persistencia, o con ella activada pero únicamente desempeñando funciones de cacheado intermedio.

También se puede utilizar como sistema de colas para desacoplar las llamadas del front de una aplicación hacia la capa de servicios, para encolar trabajos pesados y no tener que esperar una respuesta síncrona, etc.

6.3 Arquitectura

6.3.1 Almacenamiento en memoria y persistencia asíncrona

La funcionalidad básica de Redis es el de servir como motor de base de datos en memoria. Esto le confiere una velocidad y unos tiempos de respuesta que quedan lejos de los sistemas de bases de datos tradicionales y de otro tipo de sistemas NoSQL.

Que sea una base de datos en memoria significa que cuando un dato es añadido, modificado o consultado, el lugar donde ir a añadir, modificar o leer esta información directamente a disco, lo que hará será utilizar la memoria principal como elemento de almacenamiento principal.

La persistencia de estos datos es un proceso automático que recoge los datos que se están utilizando en memoria y realiza un volcado a disco, de tal forma que en caso de apagado de la máquina, esos datos no se pierdan y el tiempo de recuperación del servidor para el clúster sea menos costoso.

Principalmente en Redis existen dos tipos de persistencia: la **persistencia RDB** y la **persistencia AOF** (Redis Community 2014).

La persistencia RDB realiza snapshots (copias instantáneas de la base de datos) cada cierto tiempo del almacenamiento que tiene alojado en memoria y lo copia a disco para que sea almacenado de forma persistente y recuperado posteriormente en caso de que ocurra algún problema con el servidor. La frecuencia con la que se realiza la persistencia RDB es configurable. El comportamiento que Redis tiene por defecto es el de observar el número de claves que cambian en un cierto tiempo, y si esos valores

superan los valores que tiene configurados, realiza la llamada de persistencia. Por defecto, Redis copia la base de datos a disco si en un periodo de 60 segundos se han modificado 1000 claves o más.

La persistencia AOF, por el contrario, basa su comportamiento en dejar constancia de todas las operaciones de escritura que se realizan en la base de datos. Redis escribirá en un log la operación de escritura solicitada por parte de cliente, de manera que, llegado el caso, se pueda volver a reconstruir la base de datos en cualquier punto del tiempo ejecutando todas las operaciones registradas en ese log de forma ordenada.

En general, la recomendación que Redis hace a sus usuarios es el de implementar ambos tipos de persistencia. El RDB es especialmente recomendable cuando se quiere recuperar la base de datos en un momento concreto del tiempo, y el AOF para recuperar los últimos datos en caso de que el servidor se apague de forma no controlada. Pero esto conlleva unos costes de ocupación en disco y de sincronización de los logs.

En caso de desastre, habrá que evaluar la criticidad de tener una pérdida de información correspondiente a los últimos minutos, se podrá utilizar únicamente RDB.

Por otro lado, utilizar solo AOF no es muy recomendable, puesto que, idealmente, se debe partir de una snapshot RDB relativamente reciente para recuperar una base de datos. Restaurar un sistema que lleva meses funcionando mediante el log que escribe la persistencia AOF puede ser realmente costoso.

Por otro lado, el sistema puede ser configurado de forma que no existe persistencia ninguna, en cuyo caso Redis funcionará como una base de datos clave valor en memoria sin más. El comportamiento en este caso se asemejará más, en lo que a persistencia se refiere, a sistemas como puedan ser Memcached.

6.3.2 Replicación y Consistencia

Redis utiliza un modelo clásico de replicación tipo maestro-esclavo. De este modo, Redis consigue tener un servidor maestro que será el que mantiene y actualiza la base de datos o una parte de ella, y una serie de servidores esclavos que se encargan de tener copias de la base de datos de ese maestro para su consulta.

Algunos hechos destacables de la replicación de Redis son (Redis Community 2014):

- Redis usa replicación asíncrona.
- Los esclavos aceptan conexiones de otros esclavos, formando así una estructura en forma de grafo.
- La replicación es no bloqueante tanto para el maestro como para el esclavo. Es decir, el ambos puede estar en medio de un proceso de replicación y atender al mismo tiempo operaciones de lectura (o escritura en caso de que sea el maestro) en la base de datos. En el caso concreto de los esclavos, esto es configurable, es decir, si se desea que el comportamiento del esclavo sea no aceptar lecturas hasta haber finalizado el proceso de replicación, se puede especificar de tal modo.

La replicación tiene lugar cuando el maestro y el esclavo establecen una conexión y el esclavo envía el comando `SYNC` hacia el maestro. Lo habitual es que, tanto la primera vez como cada vez que se produce un error en las comunicaciones, la sincronización

que se produzca sea completa. En versiones más actuales se puede dar lugar una re-sincronización parcial dependiendo del escenario.

Cuando el maestro recibe la petición de sincronización, comienza un proceso de volcado de los datos a fichero, y una vez que el volcado finaliza, es enviado a través de la red. En el caso de que entren operaciones que modifiquen los datos mientras está teniendo lugar el volcado de datos, el maestro almacenará estas operaciones para después aplicarlas al conjunto una vez obtenido el fichero.

Si un dato se escribe en el maestro, hasta que el esclavo no ha recibido estos datos y no los ha volcado en su copia de la base de datos no estará disponible. Por tanto, si un dato es leído antes de una operación de sincronización, mantendrá el dato antiguo hasta que se ha sincronizado la nueva información.

Las escrituras son atómicas. Es decir, mientras una operación de escritura está teniendo lugar, ningún otro proceso puede sobrescribir ese mismo dato.

6.3.3 Particionado

El particionado es la capacidad del sistema de dividir su base de datos en múltiples instancias, de manera que cada una de las instancias sea responsable de un subconjunto de los datos.

El particionado es una ventaja a la hora de escalar, puesto que permite repartir la carga a través de múltiples nodos del clúster, y también en lo que a almacenamiento se refiere, porque la base de datos puede ser relativamente grande en tamaño, y la división y reparto de datos en varias instancias permiten alojar una mayor cantidad de información.

Redis utiliza básicamente dos tipos de particionado, **por rango** y **por hash** (Redis Community 2014).

- **Particionado por rango**. Es un tipo de particionado muy sencillo. Consiste en mapear rangos de objetos a instancias concretas. Un ejemplo sería dividir el ID de los elementos entre el número de instancias disponibles, y asignar un rango a cada nodo.
- **Particionado por Hash**. El particionado por hash consiste en aplicar una función resumen a cada uno de los elementos de la base de datos. De esta manera, cuando se va a escribir un nuevo objeto, se calculará su función hash a la cual, posteriormente, se le aplicará la función módulo N donde N será el número de nodos disponibles en el clúster, obteniendo así el servidor concreto al que deberá ir este dato.

Redis también tiene disponibles diferentes modos de realizar este particionado:

- Existe el **particionado por parte del cliente**, en el cual es el cliente el que realiza las operaciones necesarias para determinar en qué nodo debe ser almacenado el dato que está intentando escribir.
- También existe el **particionado mediante proxy**, en el que el cliente contacta con un nodo que hace las funciones de proxy y reenvía el dato al nodo que corresponda.

- Por último, está el **particionado por enrutamiento** en el cual los clientes pueden escribir en cualquier nodo del clúster, y será ese mismo nodo el encargado de determinar a la instancia a la que reenviar la información.

A la hora de implementar un tipo de particionado u otro, existen algunas alternativas.

La forma en la que el equipo de Redis recomienda realizar esta tarea es mediante el proxy llamado Twemproxy[51], que utiliza el particionado mediante proxy. Básicamente, Twemproxy es una capa intermedia entre los clientes y el software de Redis, que gestiona los servidores con los datos ya particionados. Además, en este sistema no existe un punto único de fallo (del inglés SPoF, Single Point of Failure) ya que se pueden levantar varias instancias con este software y hacer que los clientes tengan una lista de servidores a los que conectar.

Otra manera de implementar el particionado es mediante un clúster de Redis. El clúster de Redis por el momento está en fase experimental pero está llamado a ser la solución de particionado y alta disponibilidad por defecto de Redis. Esta solución es una solución híbrida entre el particionado por enrutamiento y el particionado por parte de cliente.

Para finalizar, comentar que Redis soporta Consistent Hashing, una de las formas más extendidas y estandarizadas de realizar particionado, pero únicamente si es el cliente el que realiza esta labor, es decir, se utiliza un particionado por parte de cliente

6.3.4 Gestión de colas: paradigma de mensajes generador - consumidor

El paradigma generador - consumidor es una protocolo de intercambio de mensajes que otorga una serie de ventajas que en un sistema de eventos ofrece ciertas ventajas.

Esta arquitectura permite que las aplicaciones desacoplen las llamadas de envío y recepción de eventos, dando a la base de datos un aspecto de sistema de colas en el que la entidad que publica y la entidad que consume no tienen por qué estar sincronizados.

Este sistema de envío y recepción de mensajes es utilizado por las aplicaciones que hacen uso de los diferentes comandos que dispone Redis para tal efecto.

Un ejemplo de uso de este tipo de sistema es el clásico chat o IRC en el que los usuarios pueden hablar entre ellos, con la posibilidad de suscribirse a varios canales, etc.[52]

```
connectionA> SUBSCRIBE room:chatty
 ["subscribe", "room:chatty", 1]
connectionB> PUBLISH room:chatty "Hello there!"
 (integer) 1
connectionA> ...
 ["message", "room:chatty", "Hello there!"]
```

[51] https://github.com/twitter/twemproxy

[52] Ejemplo extraido del Redis Cookbook http://www.rediscookbook.org/pubsub_for_asynchronous_communication.html

```
connectionA> UNSUBSCRIBE room:chatty
 ["unsubscribe", "room:chatty", 0]
```

6.3.4.1 *Formato de los mensajes*

Los diferentes mensajes que maneja Redis son:

- **Recibir (SUBSCRIBE)**. Confirmación de suscripción a un canal. En el mensaje se especifica el canal al que ha sido correctamente suscrito y el número actual de canales a los que se está suscrito.
- **Dejar de recibir (UNSUBSCRIBE)**. Confirmación de que se ha desvinculado de un canal. En el mensaje se especifica el canal al que se ha sido correctamente desvinculado y el número actual de suscripciones.
- **Publicar (PUBLISH)**. Cuando un mensaje se ha recibido como consecuencia de un envío de mensaje por parte de una aplicación, se envía un mensaje con el contenido original y el canal del que procede.

6.3.5 Transacciones y Atomicidad

Una transacción en Redis es un conjunto de comandos ejecutados uno tras otro, de manera que se asegure que ningún otro comando ordenado por otro cliente externo sea ejecutado entre medias de éstos.

Se dice que una operación es atómica cuando un conjunto de comandos se ejecutan todos secuencialmente de forma satisfactoria o ninguno. Es decir, no puede suceder que se ejecute satisfactoriamente tan solo una parte del conjunto de comandos.

En Redis existen 4 comandos que intervienen en lo que tiene que ver con las transacciones y la atomicidad del SGBD. Para entender como funcionan las transacciones en Redis es equivalente a entender como funcionan estos 4 comandos. Estos son MULTI, EXEC, DISCARD y WATCH.

La transacción comienza con el comando MULTI. Una vez escrito, el cliente procederá a insertar los comandos que se desee que formen parte de la transacción. Cuando la transacción haya sido escrita por completo, se escribirá el comando EXEC.

```
> MULTI
 OK
> INCR foo
 QUEUED
> INCR bar
 QUEUED
> EXEC
 1) (integer) 1
 2) (integer) 1
```

De esta manera se asegurará que ambos comandos se ejecutan secuencialmente. Si se desea cancelar una transacción en curso se utilizará el comando DISCARD.

```
> SET foo 1
 OK
> MULTI
 OK
> INCR foo
 QUEUED
> DISCARD
 OK
> GET foo
 "1"
```

Redis obtiene la atomicidad en las transacciones a través del comando WATCH. Mediante este comando, se pueden "vigilar" variables, de manera que si su resultado cambia debido a una operación externa en el transcurso de una transacción, se emite un resultado erróneo de la transacción.

```
WATCH mykey
val = GET mykey
val = val + 1
MULTI
SET mykey $val
EXEC
```

6.3.6 Consistencia

La consistencia de Redis cuando se trata de un sistema de un único nodo es inmediata, puesto que las escrituras serán inmediatamente visibles a posteriores lecturas.

La forma que tiene Redis de manejar la consistencia es bloqueando las escrituras en caso de que haya muchos esclavos que tengan un lag de replicación de varios segundos.

Para los casos de replicación maestro-esclavo, Redis es un sistema que implementa replicación asíncrona. Esto significa que el sistema permite realizar escrituras sin realizar la operación de replicado en ese mismo momento (de forma síncrona), sino que es delegado para un momento posterior (asíncrona). Por tanto, entre que la escritura es efectuada con éxito, y el esclavo escribe esa información en su base de datos, existe una ventana de tiempo en el que se produce una inconsistencia (Redis Community 2014).

El sistema funciona como se describe a continuación:

- Los esclavos contactan con el maestro cada segundo.

- El maestro recuerda cuando fue la última vez que cada uno de los esclavos contactó con él.
- EL usuario especifica un número de esclavos que no deben tener un lag de más de cierto número de segundos. Por ejemplo, no puede ser que haya más de 3 esclavos que tengan un lag de replicación de más de 4 segundos.

Si existen como mucho N esclavos que tienen como mucho M segundos de retraso con respecto al maestro, se aceptará la escritura. En caso de que no sea así, y haya más esclavos retrasados, se devolverá un error y no se permitirá realizar la escritura.

Esto asegura que los datos serán consistentes con al menos las escrituras que se realizaron anteriormente a M segundos.

6.4 Modelo de Datos

En este apartado se analizarán específicamente los 5 tipos de datos que maneja Redis, y que le confieren ciertas ventajas en comparación con otros sistemas de base de datos que no realizan estas distinciones.

6.4.1 Cadenas de texto

Las cadenas de texto son el tipo de datos más sencillo que se puede manejar en Redis.

Además, existen múltiples comandos para facilitar el manejo y la gestión de datos de tipo String, como los que permiten incrementar números, añadir otras cadenas de texto al final de otro texto, sacar longitudes, etc.

Ejemplo de uso y comandos:

```
> set mykey somevalue
 OK
> get mykey
 "somevalue"
```

`Set` y `get` son comandos de asignan y obtienen valores respectivamente de los pares clave valor del sistema.

6.4.2 Diccionarios

Los diccionarios (o hashes) son otro de los tipos básicos de Redis. Este es el tipo de datos ideal para representar objetos, entre otras cosas, dentro de la base de datos, puesto que más que pares clave valor, lo que este tipo de objeto almacena es conjuntos de pares clave valor.

Así, se podría mantener una base de datos de usuarios para una aplicación, que contenga información como apellidos, edad, fecha de nacimiento, etc.

Ejemplo de uso y comandos:

```
> hmset user:1000 username antirez birthyear 1977 verified
1
 OK
```

```
> hget user:1000 username
 "antirez"
> hget user:1000 birthyear
 "1977"
```

`Hmset` es un comando que efectúa inserciones de múltiples atributos en un hash. `Hget` extrae la información de uno de los atributos.

6.4.3 Listas

Las listas en Redis se utilizan para asignar varios valores asociados a una misma clave.

Las funciones que nos proporciona Redis para la gestión de listas son múltiples y variadas. Se podrá, por ejemplo, insertar al inicio o al final de la lista, recuperar un valor por posición, obtener el tamaño total, obtener los valores comprendidos dentro de un rango, etc.

Además, en las listas se pueden alojar referencias a otras claves, de modo que desde dentro de la propia lista se puede acceder a los valores de otros objetos de la base de datos.

Ejemplo de uso y comandos:

```
> rpush mylist A
 (integer) 1
> rpush mylist B
 (integer) 2
> lpush mylist first
 (integer) 3
> lrange mylist 0 -1
 1) "first"
 2) "A"
 3) "B"
```

`Rpush` y `lpush` son comandos que insertan elementos en la lista por la derecha y por la izquierda, respectivamente, y `lrange` es un comando que lista los elementos de una determinada lista especificando los rangos deseados.

6.4.4 Conjuntos

Los conjuntos son colecciones de cadenas de texto. Los conjuntos han de cumplir una serie de requisitos referente a los elementos que contiene. Por ejemplo, los elementos de un conjunto no se pueden repetir, y no están ordenador por ningún criterio.

Un ejemplo de uso para este tipo de datos es una lista de personas o de contactos. Otros ejemplos serían la implementación de búsquedas de las IPs que visitan un sitio web, para llevar contabilizados el número de tags que se generan en un blog,

Ejemplo de uso y comandos:

```
> sadd myset 1 2 3
 (integer) 3
> smembers myset
 1. 3
 2. 1
 3. 2
```

`Sadd` es un comando que añade elementos aun conjunto determinado. `Smembers` obtiene todos los miembros del conjunto especificado.

6.4.5 Conjuntos ordenados

Los conjuntos ordenados de Redis son similares a los conjuntos normales explicados anteriormente, pero además añaden un campo por cada elemento en el que almacenan un peso asociado al campo. De esta manera, el conjunto puede ser ordenado por pesos.

Al tener ordenados todos los elementos se pueden realizar una serie de consultas que antes no se podía, como obtener rangos de elementos que cumplen una serie de requisitos en sus pesos.

Se podría, por ejemplo, crear un ranking con la clasificación de una carrera, o de un juego con puntuaciones.

Ejemplo de uso y comandos:

```
> zadd hackers 1940 "Alan Kay"
 (integer) 1
> zadd hackers 1957 "Sophie Wilson"
 (integer) 1
> zadd hackers 1953 "Richard Stallman"
 (integer) 1
> zadd hackers 1949 "Anita Borg"
 (integer) 1


> zrange hackers 0 -1
 1) "Alan Kay"
 2) "Anita Borg"
 3) "Richard Stallman"
 4) "Sophie Wilson"


> zrange hackers 0 -1 withscores
```

```
 1) "Alan Kay"
 2) "1940"
 3) "Anita Borg"
 4) "1949"
 5) "Richard Stallman"
 6) "1953"
 7) "Sophie Wilson"
 8) "1957"

> zrangebyscore hackers -inf 1950
 4) "Alan Kay"
 5) "Anita Borg"
```

El comando utilizado por los conjuntos ordenados para añadir elementos es *zadd*. Por otro lado, *zrange* se utiliza para obtener los datos ordenados mediante algún criterio. Por ejemplo, *zrangebyscore*, devuelve los datos ordenados por valor, de menor a mayor. Además, permite especificar un rango de valores de forma que solo se muestren los que cumplen con esos rangos.

6.4.6 Bases de datos

Redis utiliza bases de datos de forma similar a los sistemas relacionales, alojando los datos en su interior.

Pero en Redis el nombrado de estas bases de datos es distinto. Redis utiliza números para identificar las distintas bases de datos que tiene creadas en el sistema. Así, la base de datos por defecto será la número 0, y a cada base de datos que se cree en adelante se le asignará un número natural creciente superior al último que se haya asignado.

6.5 Ejemplos reales de uso del SGBD

A continuación, se darán una serie de ejemplos reales en los que empresas muy importantes en el mundo de internet utilizan Redis en entornos de producción.

6.5.1 Pinterest

Pinterest es una red social que permite a sus usuarios colgar imágenes, vídeos, textos, etc., y compartirlo con el resto de la comunidad, permitiendo también crear tablones temáticos donde poder colgar y clasificar más fácilmente el contenido.

Para llevar a cabo esta tarea, Pinterest mantiene un modelo de datos al que ellos llaman el Follower Graph (el grafo de seguidores).

Pinterest permite que sus usuarios sigan a otros usuarios, lo que significa que, por extensión, seguirán todos los tablones que haya creado ese usuario. Pero también permite que los usuarios sigan tablones concretos de usuarios.

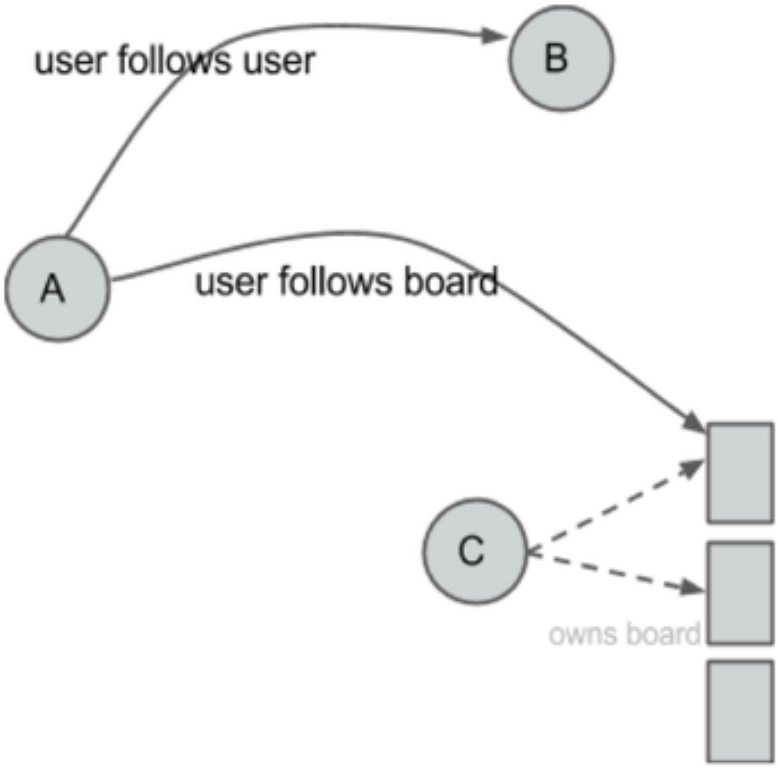

Ilustración 15. Grafo de followers de Pinterest

Según dicen en Pinterest, el tamaño del grafo no es excesivo, por lo que almacenarlo totalmente en memoria es una posibilidad. Para esta tarea, utilizan Redis. El grafo está particionado por UserID, y además persiste a disco cada segundo para reducir la posibilidad de pérdida de información. Cada instancia de Redis mantiene una partición diferente, lo que facilita la división y el reparto de particiones cuando las máquinas alcanzan sobrecargas.

6.5.2 Twitter

Twitter es una de las redes sociales más utilizadas y por tanto con más carga de del mundo.

Los usuarios de Twitter tienen pueden seguir a otros usuarios, y por cada usuario Twitter mantiene un timeline con todos los tweets que ha escrito la gente a la que sigue ese usuario.

Cada vez que el usuario accede a Twitter a través de la web o de una aplicación externa, el timeline suele ser la primera pantalla que se muestra. Este timeline tiene mucho que ver con Redis.

Twitter mantiene en un clúster de Redis en el que se mantienen los timelines de los usuarios del sitio en memoria, de forma que, cada vez que un usuario escribe un nuevo tweet, un proceso lee los seguidores de ese usuario y escribe en sus timelines una referencia al tweet que acaba de ser escrito.

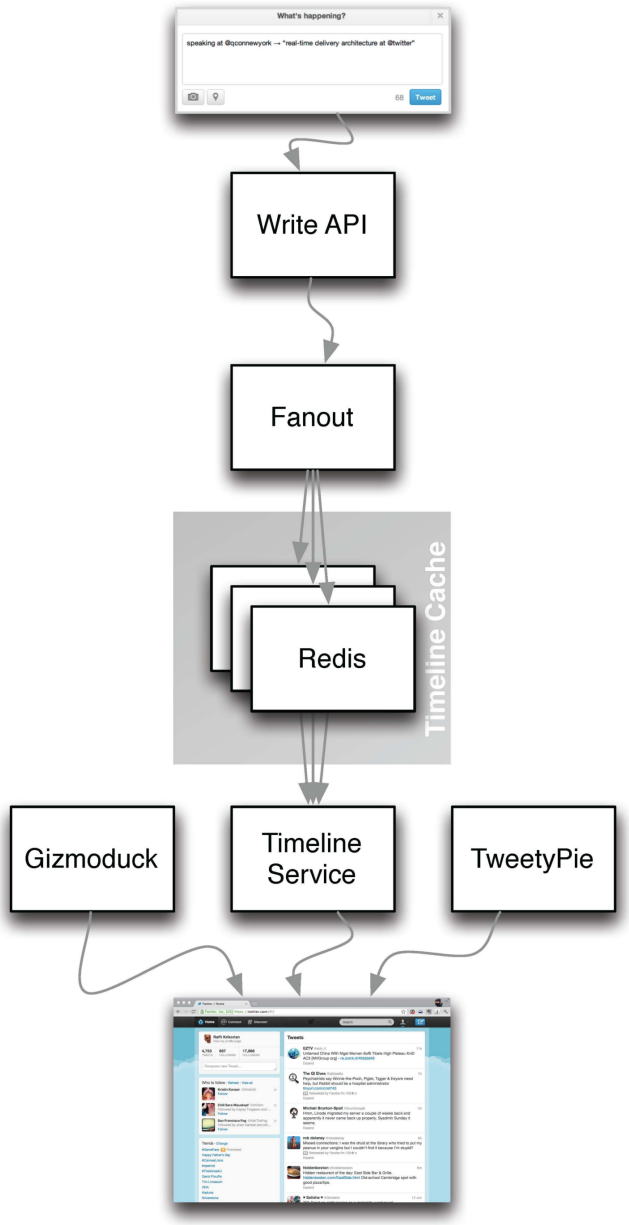

Ilustración 16. Flujo de escritura de Tweets desde una aplicación y sus actualizaciones en Redis

Es decir, cada vez que un usuario escribe un tweet, se realiza una inserción en Redis en los timelines de cada uno de sus followers. Twitter recibe un promedio escrituras de 5.000 tweets/s, con picos de 12.000, y un ratio de lecturas de 300.000 tweets/s.

Además, el rendimiento es aun más impactante si se piensa en personalidades como Katy Perry, Justin Bieber o Barack Obama, que cuentan con decenas de millones de seguidores cada uno. Cada tweet que escribe una de estas personas, provoca unos 50 millones de actualizaciones en Redis, una por cada seguidor.

7 Bases de datos en columna

8 HBase

8.1 Introducción

HBase es un SGBD, la base de datos NoSQL de Hadoop. HBase fue concebida inicialmente a partir de los artículos que liberó Google sobre BigTable allá por 2004. Se podría decir que HBase es la implementación Open Source del proyecto Bigtable de Google.

HBase es un software construido sobre Hadoop, sin el cual no podría funcionar a día de hoy. Hadoop confiere a HBase una característica indispensable en su modelo de datos: la capa de almacenamiento que proporciona disponibilidad y fiabilidad (HDFS), mientras que la otra capacidad de Hadoop, el entorno de computación de alto rendimiento (MapReduce) frecuentemente es utilizado junto con HBase para acceder a la base de datos y tratar datos en ella.

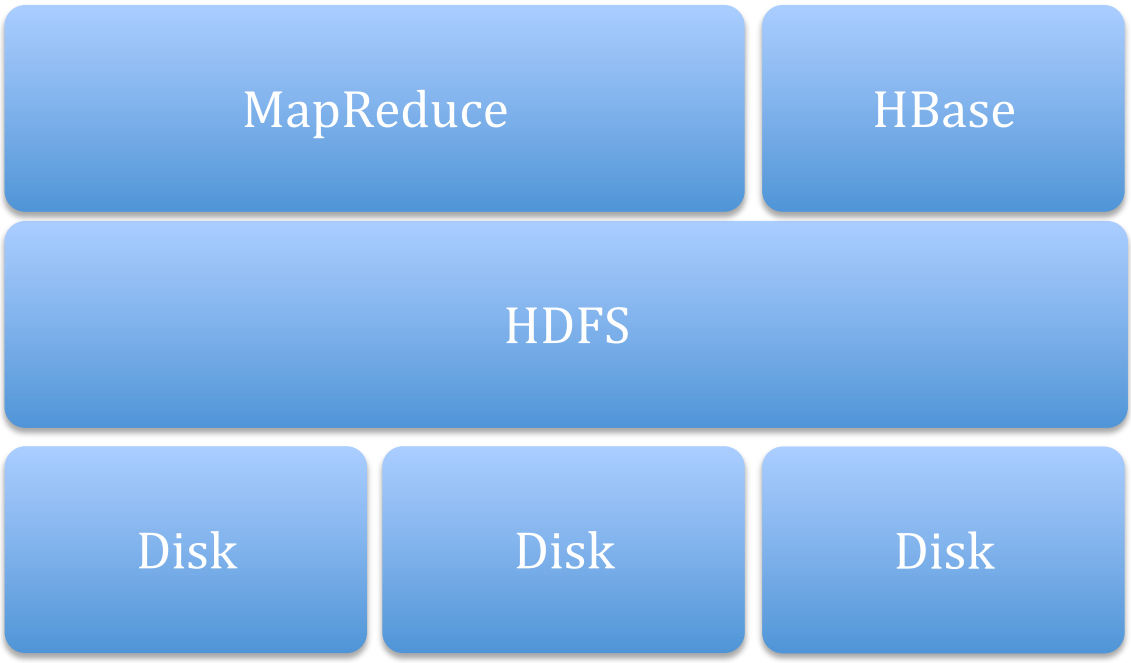

Ilustración 17. Integración de HBase con los componentes de Hadoop

Todo el contenido de este apartado se ha sacado de la documentación oficial de Apache (Apache Software Foundation 2014) y del libro *HBase, the Definitive Guide* (George 2011).

Al final de este documento se puede encontrar un anexo llamado en el que se describe como instalar un sistema básico de Hadoop con HBase. (19.1.3 Instalación de HBase).

8.2 Características principales

HBase es una base de datos **distribuida**, **persistente**, **dispersa** y **ordenada en un mapa multidimensional**. Una celda de esta base de datos viene definida por una clave primaria de fila (**rowkey**), por una clave de columna (**columnkey**) y una marca de tiempo (**timestamp)**.

Esto puede resultar ciertamente abstracto, por lo que primero se analizará punto por punto qué significa que HBase tenga estas características.

8.2.1 Distribuido

Como se ha comentado anteriormente, esta característica no es propia de HBase, sino que viene heredada del tipo de sistema de ficheros en el que se apoya. HBase utiliza para almacenar la información el HDFS de Hadoop, aunque también puede apoyarse en otros, como Amazon S3 (el sistema de almacenamiento propio de Amazon). Estos sistemas de ficheros tienen la característica de ser distribuidos y tolerantes a fallos.

8.2.2 Mapa Ordenado

A grandes rasgos, se puede equiparar HBase a un enorme mapa ordenado (*SortedMap*) de cualquier lenguaje de programación moderno que exista hoy en día. En este tipo de mapas de datos, los pares clave - valor son almacenados en estricto orden alfabético.

Es muy importante escoger una clave primaria relevante, con la que se hará la ordenación del sistema. Por ejemplo, se considerará un ejemplo práctico de una tabla que almacena nombres de dominio. Tal vez tenga sentido almacenar estos DNS's en notación inversa, esto es, en lugar de guardar *hbase.apache.org*, se almacenará *org.apache.hbase*. De esta manera se tendrá acceso en primer lugar al dominio de primer nivel, y se dejarnán los subdominios para el final de la clave. Así, las columnas referidas a subdominios permanecerán "cerca" unas de otras, al estar ordenadas primero por el dominio común a todas ellas.

8.2.3 Multidimensional

Ya se ha comentado que HBase es un tipo de base de datos NoSQL orientado a columnas. ¿Que significa esto?

Para comprender el hecho de que HBase sea una base de datos multidimensional, debe olvidarse el significado clásico de los términos tabla y columna. Estos conceptos han cambiado con respecto a lo que se acostumbra a leer en la literatura tradicional de los sistemas SGBD. Ahora el concepto de tabla se acerca más al que se tiene de un hash / map.

A continuación se expone un ejemplo para ilustrar como almacenaría HBase la información.

```
{
 "1" : {
   "A" : "x",
   "B" : "y"
 },
```

```
 "2" : {
   "A" : "y",
   "B" : "z
 },
 "3" : {
   "A" : "z",
   "B" : "a"
 }
}
```

Tabla 7. Tabla Bidimensional de HBase

	A	B
1	x	y
2	y	z
3	z	a

En el ejemplo superior se representa un *SortedMap* cuyas claves apuntan a otro *SortedMap* con exactamente 2 claves: A y B. En adelante, se identificará al par clave-valor de nivel superior como **fila**. En este ejemplo, las claves de nivel superior vienen representadas por "1", "2" y "3" respectivamente, y sus correspondientes valores podrían ser valores propiamente dichos, aunque en este caso son nuevos mapas clave-valor, cuya clave viene identificada por las letras "A" y "B". Los mapeos que realizan A y B se conocen como Familias de Columnas.

Las familias de columnas de una tabla son especificadas en el momento de la creación de la tabla, y además es muy difícil o directamente imposible modificarlas una vez creadas. Así es preferible pensar bien el diseño de la tabla antes de empezar a poblarla con datos.

La principal característica de las Familias de Columnas es que pueden tener un número arbitrario de columnas. Estas columnas siempre pertenecen a una Familia de Columnas de las que se han preestablecido anteriormente. Estas columnas vienen identificadas por una etiqueta o calificador.

Ampliando el ejemplo propuesto anteriormente:

```
{
```

```
 "1" : {
   "A:foo" : "x"
   "A:bar" : "y"
   "B:" : "a"
 },
 "2" : {
   "A:foo" : "z",
   "A:bar" : "k",
   "B:" : "b"
 },
 "3" : {
   "A:bar" : "S",
   "B:" : "c"
 }
}
```

Tabla 8. Tabla Multidimensional de HBase

	A		B
	foo	bar	“”
1	x	y	a
2	z	k	b
3		S	c

En el ejemplo expuesto, las Column Families A y B se mantienen, pero se han añadido 2 etiquetas para A (foo y bar) y una para B (el conjunto vacío). El conjunto de Column Family y Etiqueta es lo que se denomina Columna. Para referirse a una columna concreta, simplemente se pondrán dos puntos (“:”) entre medias de ambos valores. Esto es, para este ejemplo, se obtendrían las columnas: “A:foo”, “A:bar” y “B:”

Aún existe una dimensión más. En HBase, el tiempo marca esta última dimensión. El versionado se consigue incorporando un timestamp de forma implícita con cada fila. También se puede añadir este campo explícitamente. Esto significa que para una misma celda (combinación de columna – fila) se pueden tener varios valores, valores que serán distintamente accedidos en función del valor del timestamp asociado.

Por último, vale la pena mencionar que todos los miembros de una column families se almacenarán físicamente juntos.

8.2.4 Disperso

En las bases de datos tradicionales, las filas son dispersas pero no las columnas. Esto significa que cada vez que se crea una fila, se reserva siempre espacio para todas y cada una de las columnas, con independencia de si el valor va a ser proporcionado.

La dispersión en esta base de datos principalmente hace referencia al hecho de que, una fila en HBase puede contener cualquier número de columnas de cada column family, desde todas hasta ninguna incluso repetir algunas de ellas. La única condición que debe cumplir es la de que el timestamp ha de mantenerse distinto.

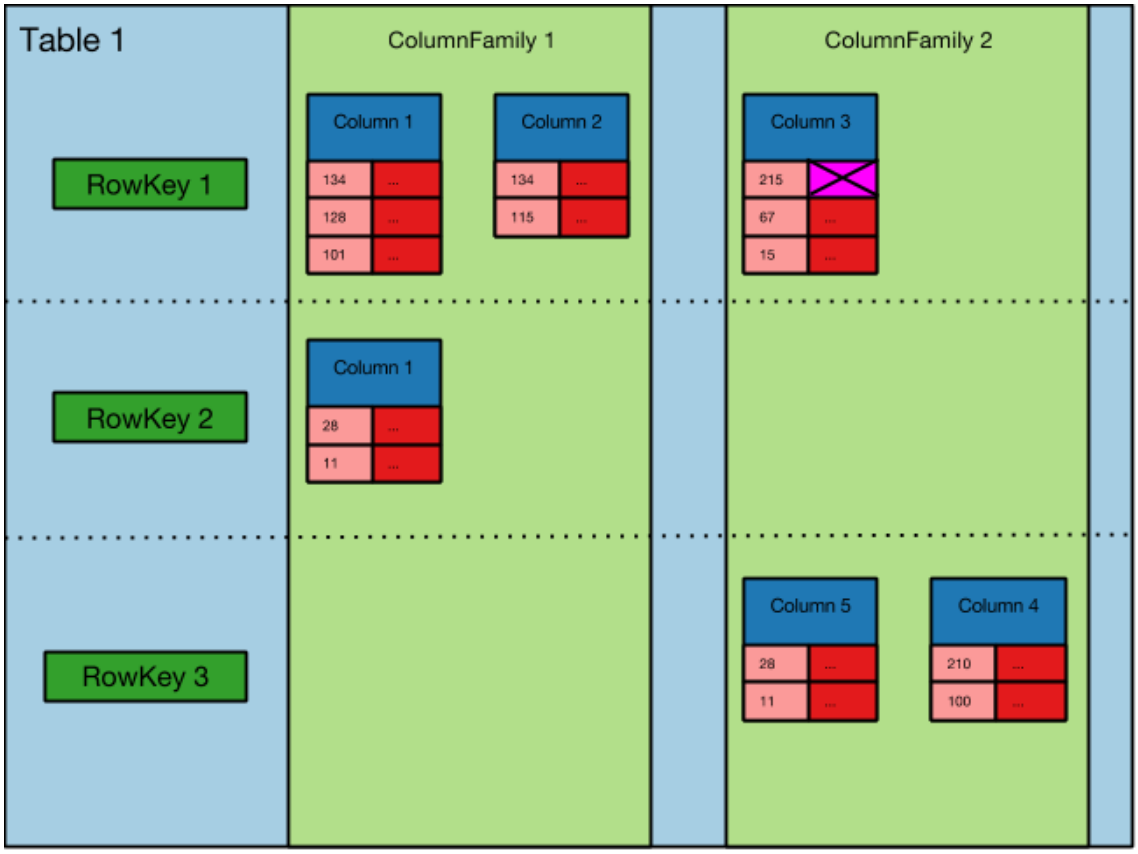

Ilustración 18. Dispersión en una tabla de HBase

En caso de que no se asignara valor en alguna columna para un registro concreto, no se reservaría espacio ninguno.

8.3 Arquitectura

La arquitectura de HBase sigue un modelo tradicional Maestro-Esclavo. Típicamente, el clúster posee un nodo maestro denominado *HMaster*, y uno o más servidores esclavos, conocidos como *HRegionServer*, servidores que a su vez son capaces de gestionar una o varias regiones, *HRegion*. A continuación se describirá más en profundidad todos estos conceptos.

8.3.1 Diseño del Sistema

8.3.1.1 HMaster

El servidor maestro, cuya implementación en HBase se conoce con el nombre de *HMaster*. Cuando se está ejecutando sobre un clúster de Hadoop en modo distribuido, habitualmente se aloja en la misma máquina que el *NameNode* (maestro del *HDFS*).

El *HMaster* es el encargado de monitorizar el estado de todos los nodos esclavos del clúster, y es donde se efectúan los cambios en los metadatos.

8.3.1.2 HRegionServer

Los servidores esclavos, en los clústers de HBase, son llamados *HRegionServers*. Cada uno de estos servidor de región es responsable de alojar y servir un conjunto de regiones, mientras que una región concreta solo puede pertenecer a un *HRegionServer*.

Los procesos *HRegionServer*, en el modelo distribuido, suelen correr en los nodos que también tienen los procesos de los *DataNodes*.

8.3.1.3 Regiones

Las regiones son las unidades mínimas que utiliza HBase para agrupar las filas y asignarlas en conjunto a los esclavos. Así, las regiones se pueden mover de unas máquinas a otras, se pueden realizar backups, etc.

La jerarquía de elementos dentro de cada región es de la siguiente forma (ver Ilustración 19):

- **Tabla**.
- **Región**. Regiones por cada tabla
- **Almacén**. Almacenes por región. Cada almacén representa una column family. Existen dentro de las regiones. Un almacén aloja un memstore y 0 o más *StoreFiles*.
- **MemStore**. Los *memstores* son los encargados de mantener en memoria los datos modificados del sistema. Existe un único memstore por cada almacén.
- **StoreFile o HFile**. Los *storefiles* son las estructuras que maneja HBase para almacenar reside la información. Puede haber varios *Storefiles* por cada región.
- **Bloque**. Los *StoreFiles* están formados por bloques. El tamaño de cada uno de estos bloques puede variar entre column families. La compresión de los datos sucede siempre a nivel de bloque dentro de cada *StoreFile*.

En general, HBase ha sido diseñada ser capaz de mantener en el clúster unas pocas regiones (20-200) de un tamaño de unos 5-20 GB cada una por cada región server.

Idealmente, el número de regiones que deberían existir en HBase nunca debería ser muy elevado. Esto es debido a que los gastos de memoria, de entrada / salida y la los tiempos de comunicación con Zookeeper son directamente proporcionales al número de regiones.

Por ejemplo, el MSLAB (los buffers de memoria de los MemStores), por defecto requiere de 2 MB de memoria por cada column family y por cada región. Teniendo, por

ejemplo, 1000 regiones que tengan 2 familias, hace un total de 4 GB de memoria necesarios, aparte después de los propios datos.

8.3.1.3.1 Asignación de regiones a los distintos RegionServer y caída de los mismos

Cuando el clúster de HBase arranca, las regiones son asignadas de la siguiente manera.

- El maestro invoca al proceso `AssignmentManager` durante el arranque.
- Éste `AssignmentManager` busca las asignaciones de las regiones en la tabla META.
- Si estas asignaciones son correctas (el nodo sigue en disposición de recibir la región), entonces se mantienen.
- Si alguna de las asignaciones no son válidas, entonces se invoca el `LoadBalancerFactory` y se reasigna dicha región. Lo que hará el `LoadBalancerFactory` será básicamente asignar la región a un servidor de forma aleatoria.
- Tras el cambio, se actualiza la tabla META para que contenta la información actualizada del cambio. Además, el esclavo que ha recibido la región arranca el proceso que gestionará esta parte de los datos y registra todas estas actualizaciones.

De manera similar, cuando un RegionServer falla y sale del clúster, el procedimiento a seguir consistiría en que cuando el maestro detecte que el nodo ha caído marcaría las regiones como inválidas, y como resultado de esta invalidación en la tabla META se procedería a reasignarlas de forma análoga a lo explicado antes para el caso del arranque del clúster.

8.3.1.3.2 Balanceo de Carga

De forma continuada, el sistema realiza chequeos periódicos sobre el estado de las regiones y de los región servers, y puede decidir que hay que cambiar una región de esclavo o bien que una región ha crecido demasiado y hay que dividirla en dos.

Siempre que no haya ninguna otra región en tránsito, el balanceo moverá regiones entre RegionServers para tratar que la carga del sistema quede lo más repartida posible entre todas las máquinas que conforman en clúster.

HBase tiene un parámetro mediante el cual se especifica un tamaño máximo que puede alcanzar las regiones. Una vez que se alcanza ese tamaño, se divide automáticamente la región en 2 más pequeñas y, opcionalmente, se redistribuye una o las 2 a nuevos RegionServers.

En general, esta labor de división es bastante costosa, por lo que hay que intentar minimizarla en la medida de lo posible.

8.3.1.4 Catalog Tables

La mayoría de tablas que existen en la vida de un sistema HBase serán las tablas creadas por los usuarios o aplicaciones del sistema.

Pero existen dos tablas especiales, cuyo cometido no es el de almacenar datos de usuario, sino que sirven como metadatos del propio sistema HBase. Estas tablas se

llaman *-ROOT-* y *.META.*, y también son conocidas como **Catalog Tables**. Son tablas que no aparecen cuando, por ejemplo, se intentan listar las tablas que contiene la base de datos, pero son tablas como cualquier otra del sistema. Son creadas por HBase la primera vez que arranca.

La tabla *-ROOT-* almacena información sobre la ubicación de la tabla META. La región en la que está ubicada y el RegionServer encargado de alojar esa región.

La tabla *.META.* contiene el listado de regiones que están creados en un momento determinado en el sistema.

8.3.2 Funcionamiento

Como ya se ha comentado anteriormente, el maestro de HBase coordina el clúster, y es el responsable de las tareas de administración del mismo.

Un RegionServer puede estar al cargo de varias regiones distintas. Cada una de estas regiones es asignadas a un RegionServer durante el arranque del clúster, y el nodo maestro puede decidir que una región va a cambiar de RegionServer como resultado de una operación de balanceo, o bien debido a un fallo en uno de los esclavos que obligue a reubicar todas las regiones que estaban alojadas en el mismo.

El mapeo entre las regiones y los nodos esclavos son almacenadas en una tabla especial de HBase llamada META. Consultando la tabla meta, un proceso externo es capaz de determinar en que RegionServer se encuentra el registro que está buscando. Así, el maestro queda fuera en las operaciones de lectura y escritura, permitiendo a los clientes contactar directamente con los RegionServers para solicitar o proporcionar los datos (ver Ilustración 19).

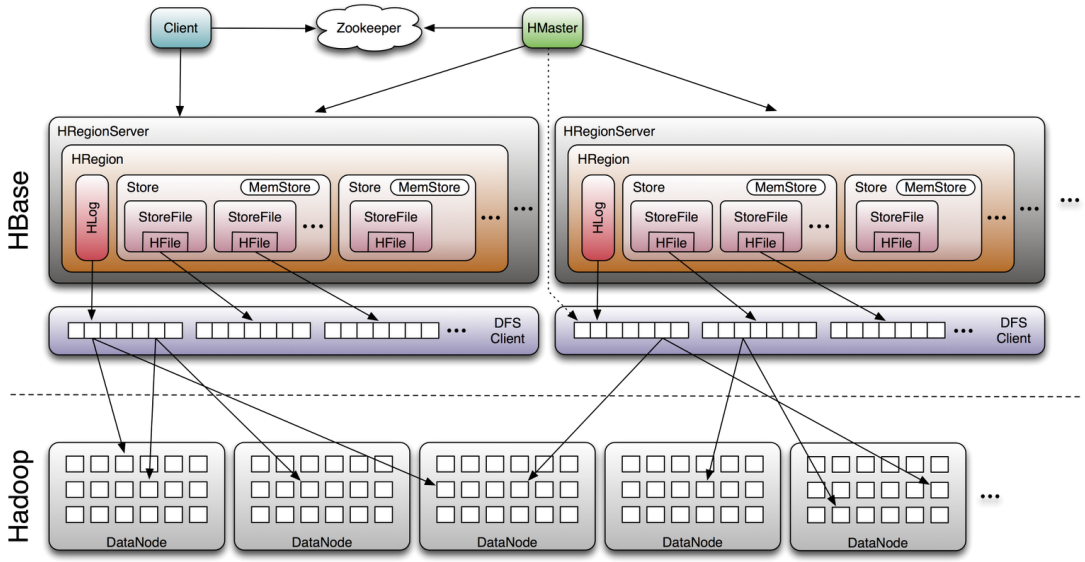

Ilustración 19. Arquitectura HBase[53]

[53] Imagen extraída de http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

El flujo normal de comunicación es: un nuevo cliente contacta con el clúster de zookeeper primero, cuando intenta acceder a una fila en particular. Esto se hace obteniendo el nombre del servidor que está alojando en ese momento la región -ROOT- de zookeeper. Con esta información, puede consultarle a ese servidor que otro servidor es el que está alojando la tabla .META. que contiene la columna en cuestión. Estos dos datos son almacenados en memoria caché y solo se realizan las búsquedas una vez. Por último, se consulta el .META. server y se obtiene el nombre del servidor al cargo de la región que contiene la fila que busca el cliente.

8.4 Modelo de Datos

En este apartado se dará una visión sobre cómo y de qué manera HBase almacena sus datos.

Como se ha comentado anteriormente, HBase utiliza tablas para almacenar la información. Estas tablas están divididas en filas y columnas. Estas columnas son agrupadas a su vez en *column families* previamente definidas (Chang, et al. 2006).

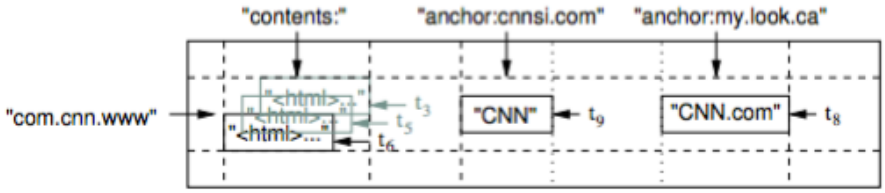

Ilustración 20. Ejemplo de una fila en una tabla de HBase

8.4.1 Filas

Las claves principales que se tienen en las filas son String aleatorios, y pueden tener hasta 64 KB de tamaño.

Cada lectura / escritura de una fila en una tabla es atómica, independientemente del número de columnas a leer o escribir. Esto fue una decisión de diseño que se tomó pensando en tratar de facilitar, en la medida de lo posible, la predicción del comportamiento del sistema bajo una gran carga de trabajo (multitud de accesos simultáneos sobre los mismos datos).

Asimismo, las claves se mantienen ordenadas lexicográficamente. Los rangos de filas (conocidos como *Regiones*) son particionados dinámicamente. Se puede decir que las regiones son las unidades mínimas de distribución y carga a lo largo del sistema. Cuando una región crece demasiado, se divide, o se puede mover de un servidor a otro. De esta manera, cuando se realizan lecturas sobre un conjunto no consecutivo de filas en la tabla, sólo se requiere comunicación entre un pequeño número de máquinas.

Por ejemplo, si se quisiera almacenar todas las páginas web que son servidas desde un mismo dominio en HBase, la clave principal debería comenzar por el dominio, porque de esa forma se conseguirá que permanezcan "cerca" dentro de la base de datos, y se facilitarán posteriores lecturas y escrituras en el sistema.

8.4.2 Column Families

Las columnas, en HBase, se agrupan en lo que se conoce como *Column Families*. Estas familias de columnas deben ser creadas antes de comenzar a almacenar información dentro de la base de datos. Una vez que la familia se ha definido, las columnas se pueden crear en cualquier momento, incluso al vuelo mientras la base de datos está corriendo sin necesidad de reiniciarla. Pero no se podrán agregar nuevas *Column Families* una vez se ha empezado a poblar la tabla.

Los nombres de las columnas siguen la siguiente sintaxis:

familia:nombre

El prefijo *familia* debe estar compuesto por caracteres imprimibles, mientras que el *nombre* puede estar compuesto por cualquier combinación aleatoria de caracteres.

Los controles de acceso y el almacenamiento de los datos en disco se realiza todo a nivel de *column family*. Las columnas de las *column families* se almacenan todas consecutivas en memoria y en disco.

8.4.3 Timestamps

Cada celda de HBase puede contener varias versiones de los mismos datos. En ese caso, la información tiene una dimensión más: el *timestamp*.

En HBase, los *timestamps* son números de 64 bits, y pueden ser asignados automáticamente en función del momento en el que se crea el objeto, o bien pueden ser especificados por la aplicación. El almacenamiento se realiza en orden descendente de tal forma que se puedan leer primero las versiones más recientes.

A continuación se expondrá un ejemplo de un diseño tanto conceptual como físico de una tabla de HBase. La tabla en cuestión se llama *webtable*, y está basada en el ejemplo descrito en el artículo de Google.

8.4.4 Diseño conceptual

En este ejemplo se mostrará el diseño de una supuesta tabla creada en HBase llamada Webtable. Esta tabla contiene dos *Column Families*: *contents* y *anchor*. A su vez, la CF anchor tendrá dos columnas*: anchor:cssnsi.com* y *anchor:my.look.ca*, mientas la CF *contents* tan solo tendrá la columna *contents:html*.

Así es como quedaría el diseño conceptual de la tabla Webtable.

Tabla 9. Diseño Conceptual de la tabla Webtable

Row key	Timestamp	Column Family contents	Column Family anchor
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"

"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"
"com.cnn.www"	t6	contents:html = "<html>..."	
"com.cnn.www"	t5	contents:html = "<html>..."	
"com.cnn.www"	t3	contents:html = "<html>..."	

8.4.5 Diseño físico

Físicamente, se representarán las columnas en dos diseños independientes puesto que, a pesar de pertenecer a la misma tabla, nada tienen que ver la una con la otra. Las columnas dentro de cada *column family* se almacenan juntas, y así quedaría representado.

8.4.5.1 Column Family anchor

Tabla 10. Column Family Anchor

Row key	Timestamp	Column Family anchor
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

8.4.5.2 Column Family contents

Tabla 11. Column Family Contents

Row key	Timestamp	Column Family contents
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

Es importante destacar que las celdas vacías que aparecen en la vista conceptual, no aparecen en el modelo físico de datos. Esto significa que físicamente no se almacenará ninguna referencia a estas columnas.

Una consulta a la columna *contest:html* con un *timestamp* = *t8* no devolvería ningún valor, al igual que si se preguntara por *anchor:my.look.ca* con *timestamp* = *t9*.

En caso de no especificar *timestamp*, se devolverán las filas cuyos *timestamps* sean más recientes. Esto es: *contest:html* con *timestamp t6*, *anchor:cnnsi.com* con *timestamp* t9 y *anchor:my.klook.ca* con *timestamp t8*.

8.5 Zookeeper

Se ha comentado anteriormente por encima, pero para hablar de HBase es necesario dar una breve introducción también sobre lo que es Zookeeper

HBase necesita de **Zookeeper** para funcionar. Habitualmente, cuando se arranca un sistema HBase, con él se levanta también un clúster de Zookeeper. Si el sistema tiene otros proyectos del ecosistema Hadoop funcionando, es habitual que el *quórum* de Zookeeper se mantenga independiente de HBase y del resto de herramientas, pero en la versión actual más estable de HBase (0.94) Zookeeper se ofrece como servicio embebido, por lo que no es necesario descargar o instalar ningún componente adicional aparte del propio HBase y Hadoop.

Zookeeper es otro proyecto de la fundación Apache y forma parte del ecosistema Hadoop. Zookeeper es el servidor de coordinación y sincronización distribuido del que hacen uso todas las herramientas del ecosistema Hadoop. Sin ZK ninguna de estas herramientas podría funcionar ni comunicarse con los servicios del core de Hadoop (HDFS y MapReduce).

Los clústers de Zookeeper son más conocidos como *quórums*, y será lo que emplean los clientes de HBase para contactar con la base de datos. Todos los componentes del clúster de HBase (maestro y esclavos) deben registrarse en el *quórum* de ZK. Cuando un cliente quiere realizar una operación de lectura o escritura en HBase, primero debe contactar con Zookeeper y obtener la información de qué RegionServer es el que contiene en ese momento la tabla ROOT. A partir de ese punto, comenzará a articular las llamadas necesarias para ponerse en contacto con el resto de nodos del clúster de HBase hasta que haya resuelto la operación que deseaba realizar.

8.6 Ejemplos reales de uso del SGBD

8.6.1 Facebook

El sitio que más intensamente utiliza HBase en producción actualmente es probablemente, Facebook con el caso particular de su sistema de mensajería.

Facebook implementó, allá por el año 2010, su sistema de mensajería instantánea, que se llamaba Messages. Actualmente este producto a evolucionado y se ha convertido en Facebook Messenger.

Por aquel entonces, Facebook necesitaba una infraestructura que fuera capaz de soportar los 350 millones de usuarios que tenía la plataforma (actualmente cuenta con más de mil millones), que compartían a través de la plataforma más de 15 mil millones de mensajes al mes.

Cuando evaluaron diferentes tecnologías que pudieran dar soporte a este movimiento de información, analizaron alternativas como clústers de MySQL, HBase o Cassandra. Facebook había liberado dos años antes el artículo técnico que describía cómo habían desarrollado Cassandra (se verá en la siguiente sección) para su Omnibox (la caja de búsqueda que actualmente puede verse en cualquier cuenta de Facebook). Cassandra se perfilaba, en un principio, como candidata, ya que tenían a los equipos técnicos formados y era un sistema desarrollado casi enteramente en la propia empresa.

Cuando visualizaron los patrones de comportamiento de los datos de Messages se dieron cuenta que había dos tipos de datos: un pequeño conjunto de datos muy volátil, y un cada vez más creciente conjunto de datos que eran muy raramente accedidos.

Finalmente, la decisión fue montar el sistema en HBase, puesto que Cassandra tiene unas particularidades en temas de consistencia de la información que no cuadraban con el sistema que deseaban montar (Muthukkaruppan 2010).

"*HBase proporciona muy buena escalabilidad y rendimiento para esta carga de trabajo y un modelo de consistencia más simple que Cassandra. A pesar de que hemos trabajado mucho con HBase durante el año pasado, cuando empezamos nos pareció que era la más completa en cuanto a términos de nuestras necesidades (balanceo de carga y failover automático, soporte de compresión, múltiples particiones por servidor, etc.) HDFS, el sistema de archivos subyacente utilizado por HBase, proporciona varias características interesantes, como la replicación, las sumas (checksum) de comprobación de extremo a extremo, y el rebalanceo automático. Además, nuestros equipos técnicos ya tenían bastante experiencia operativa con HDF de haber utilizado procesamiento de datos con Hadoop. Desde que empezamos a trabajar en HBase, nos hemos interesado en mantener el proyecto mismo de HBase, trabajando de cerca con la comunidad. La versión de código abierto de HBase es lo que estamos utilizando en la actualidad*."

Actualmente, Facebook utiliza un software propio llamado HydraBase, que toma la idea inicial de HBase, pero realiza algunas modificaciones en los conceptos más básicos.

Por ejemplo, en lugar de alojar cada una de las regiones en un servidor de región, HydraBase aloja cada región en un grupo de servidores de región. De este modo, en caso de que alguno de los servidores deje de estar accesible no será necesario inicial una reubicación de la región, sino que esa misma región está ya repartida por otros servidores.

8.7 Diseño de una base de datos real con HBase

A continuación se mostrará un diseño de tablas basado en cada uno de los ficheros de los que forman el corpus. La información sobre este corpus está descrito en el anexo correspondiente (19.2 Anexo II: Descripción de los tweets que conforman el corpus).

8.7.1 Instalación de Hadoop y HBase

Previo a la realización de cualquier diseño sobre HBase, habrá que tener el software correctamente instalado y configurado en el sistema. Para ello, se deberá seguir el anexo que se ha preparado para tal efecto (19 Instalación de HBase).

8.7.2 Diseño de tablas por fichero

- **Fichero info-general-tweets-train.xml**

Tabla 12. HBase: diseño del fichero info-general-tweets-train.xml

	TweetInfo				SentimentInfo		TopicInfo
	user	date	lang	content	polarityType	polarityValue	topic
Tweet ID							
Tweet ID							
Tweet ID							

RowKey: *Tweet ID*

Column Families: *Tweet Info, Sentiment Info, Topic Info*.

Columnas: TweetInfo:user, TweetInfo:date, TweetInfo:lang, TweetInfo:content, SentimientInfo:polarityType, SentimientInfo:polarityValue, TipocInfo:topic.

Vale la pena recordar que las *column families* deberían definirse al inicio y no variar nunca.

Con estas columnas el tweet queda perfectamente reflejado y encuadrado en la base de datos. Incluso se deja abierta la posibilidad de que, si en un futuro se desea agregar más información referida al sentimiento del tweet o de información sobre tópicos, se pueda añadir una nueva columna y la información continúe estructurada correctamente.

En caso de haber más de una polaridad o más de un topic se especificarían todas con diferentes timestamps, y tendría que ser tenido en cuenta en el momento de las lecturas.

Variaciones al diseño: Como se ha visto anteriormente, HBase establece una ordenación lexicográfica por clave de fila, o *RowKey*. En este caso, no nos proporciona demasiada ventaja haber establecido el TweetID como RowKey, puesto que mantener este campo ordenado tan solo nos sirve para buscar por TweetID, lo que no parece un caso de uso muy habitual. Quizá una posible variación sería intercambiar las posiciones de user y TweetID. De esta manera, se obtiene una ganancia en los casos en los que se quiera realizar búsquedas por usuario.

- **Fichero info-general-tweets-test.xml**

Tabla 13. HBase: diseño del fichero info-general-tweets-test.xml

	TweetInfo			
	user	date	lang	content
Tweet ID				
Tweet ID				
Tweet ID				

RowKey: *Tweet ID*

Column Families: *Tweet Info*.

Columnas: *TweetInfo:user, TweetInfo:date, TweetInfo:lang, TweetInfo:content*.

Este fichero es similar al anterior, solo que éste contiene tweets que aun no han sido etiquetados con sentimientos, así que no tiene estos campos. Tan solo se deja la información sobre el tweet.

- **Fichero info-general-users.xml**

Tabla 14. HBase: diseño del fichero info-general-users.xml

	UserInfo							TwitterInfoAccount					
	Name	Group	Description	URL	Timezone	lang	Location	Screen name	Profile image	Followers	Creation date	Type	Tweets
User ID													
User ID													
User ID													

RowKey: *User ID*

Column Families: *User Info*, *Twitter Info Account*

Columnas: *UserInfo: Name, UserInfo: Group, UserInfo: Description, UserInfo: URL, UserInfo: Timezone, UserInfo: Lang, UserInfo: Location. TwitterInfoAccount: ScreenName, TwitterInfoAccount: ProfileImage, TwitterInfoAccount: Followers, TwitterInfoAccount: CreationDate, TwitterInfoAccount: Type, TwitterInfoAccount: Tweets*.

Este fichero, info-general-users.xml, contiene información acerca de los usuarios que aparecen en el resto de ficheros que forman el corpus.

Cada column family trata de agrupar aspectos comunes al usuario. La column family *UserInfo* recopila tags referidos a la información personal del usuario y la CF *TwitterInfoAccount* hace referencia a los tags que tienen que ver directamente con su cuenta d Twitter.

- **Fichero info-politics-tweets-test.xml**

Tabla 15. HBase: diseño del fichero info-politics-tweets-test.xml

	TweetInfo				SentimentInfo	
	User	Date	Lang	Content	Source	Entity
Tweet ID						
Tweet ID						
Tweet ID						

RowKey: *Tweet ID*

Column Families: *Tweet Info y Sentiment Info*.

Columnas: *TweetInfo: User*, *TweetInfo: Date*, *TweetInfo: Lang*, *TweetInfo: Content*, SentimientiInfo: Source y SentimientoInfo: Entity.

Este es un conjunto de tweets que contienen información política. Además, en este conjunto de tweets se han buscado diferentes entidades que guardan relación con algún partido político, y se ha etiquetado la entidad y el partido en un apartado de sentimientos, que compone una de nuestras Column Families.

Ya se han realizado diseños para todos los ficheros que comienzan por "**info-**". Como se ha comentado anteriormente, los ficheros que comienzan por "**info-**" contienen la misma información que los que no comienzan por "**info-**", pero con algún campo extra. Así que los diseños de cada uno de esos ficheros coindicen con sus homónimos, pero restando los campos que no son necesarios en cada caso.

Por ejemplo, para el último diseño presentado, el **info-politics-tweets-test.xml**, se podría hacer un diseño análogo para **politics-tweets-test.xml**. Se eliminaría la columna *TweetInfo:content*, ya que es lo único en lo que se diferencia quedando de la siguiente manera:

Tabla 16. HBase: diseño del fichero politics-tweets.test.xml

	TweetInfo			SentimentInfo	
	User	Date	Lang	Source	Entity
Tweet ID					
Tweet ID					
Tweet ID					

Como se aprecia, es prácticamente igual, y gracias a la dispersión que nos brinda HBase se puede hacer esto sin ningún coste adicional.

8.7.3 Diseño general de tablas

Tras haber indicado los diseños para cada uno de los ficheros del corpus, se puede extraer algunas conclusiones generales.

Lo importante de los diseños sería:

- El ID de cada registro es el ID del Tweet, de manera que se pueda identificarlo siempre unívocamente entre todas las tablas.
- Las column families son *TweetInfo*, *SentimientoInfo* y *TopicInfo* para casi todos los ficheros. Los únicos que tienen CFs distintas son general-users.xml e info-general-users.xml. Estos ficheros harían uso de unas column families nuevas que serían **UserInfo** y **TwitterInfoAccount**. Al permitir HBase la dispersión en la base de datos, se podrían rellenar o no a placer las demás columnas, y si alguna de las columnas no aplica en algún caso, se dejaría vacío.

Es decir, una vez visto las generalidades que presentan los diseños, se podría reducir el diseño de las tablas a únicamente 2 versiones de tabla, que quedarían como sigue:

Tabla 17. HBase: Tabla genérica 1

	TweetInfo	SentimentInfo	TopicInfo
Tweet ID			
Tweet ID			

Tabla 18. HBase: Tabla genérica 2

	UserInfo	TwitterInfoAccount
Tweet ID		
Tweet ID		

Con estas dos tablas, con esta estructura de column families, se podrían alojar cualquier tweet de los que se encuentran dentro del corpus que se maneja. Cada uno de los ficheros elegirá que columnas es necesario incluir y cuales no.

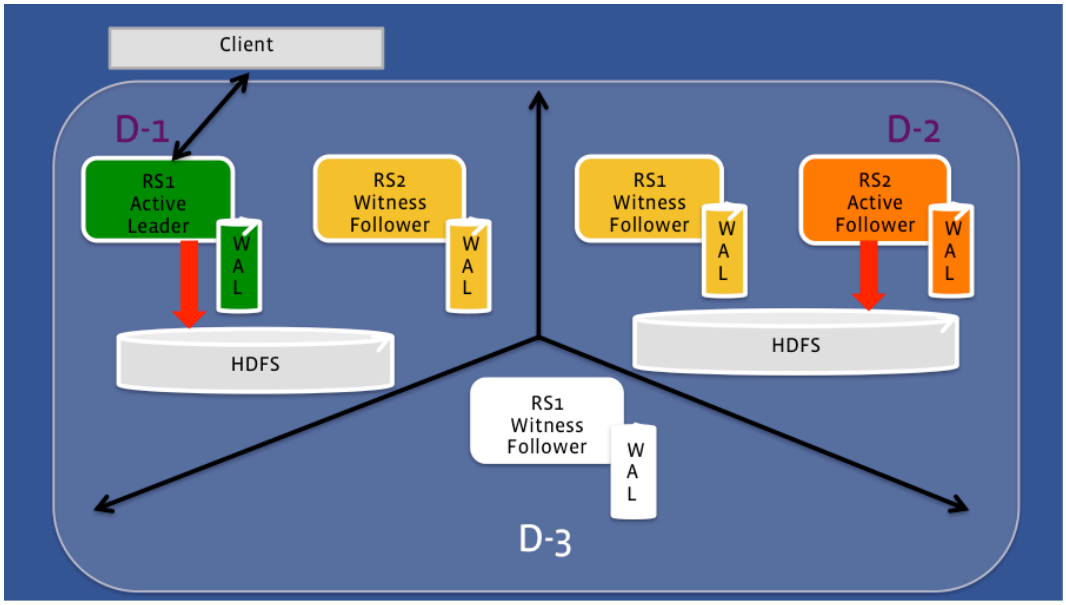

Ilustración 21. Arquitectura desplegada en HydraBase

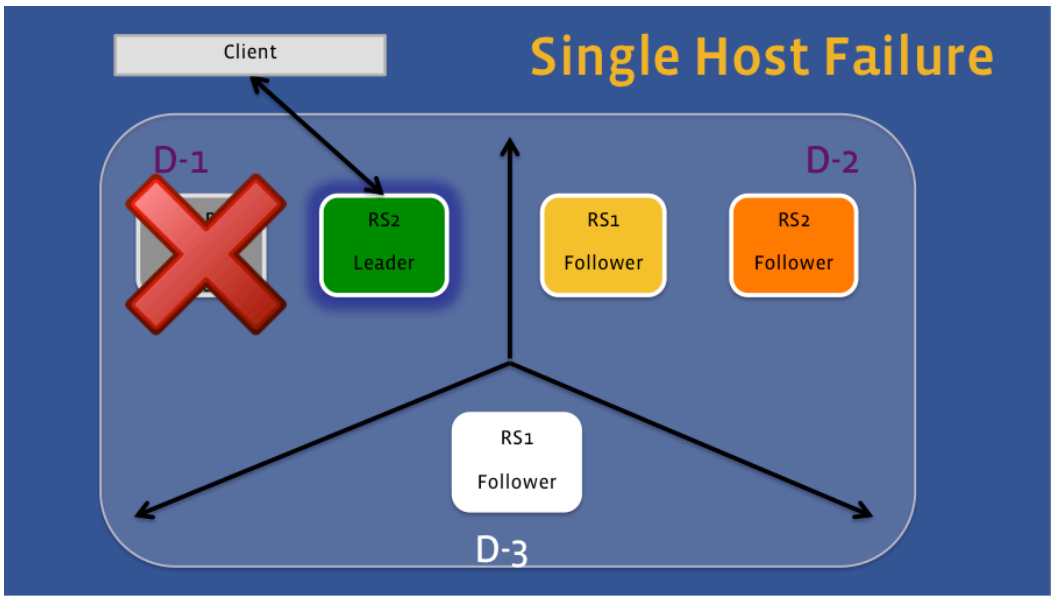

Ilustración 22. Erro de un nodo en HydraBase[54]

54 Imágenes obtenidas de https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/

9 Cassandra

9.1 Introducción

Cassandra es un almacén clave valor, altamente escalable, distribuido y eventualmente consistente.

Cassandra es similar a DynamoDB con respecto a la consistencia eventual que implementa en su sistema, y a la vez, toma prestados de Google y de su proyecto BigTable el modelo de datos estilo columnar.

Cassandra ha sido concebido e inicialmente desarrollado por Facebook hasta que su código fue liberado en el año 2008. Desde 2010 forma parte de los proyectos de primer nivel de Apache.

Cassandra fue inicialmente ideado para mejorar la barra de búsqueda de la popular red social Facebook, y uno de sus creadores originales, Avinash Lakshman, llegó de trabajar directamente con Amazon, formando parte del grupo de ingenieros que dio forma a DynamoDB.

Actualmente Cassandra se utiliza en varios de los sitios más conocidos y con más necesidades en cuanto a tráfico y almacenamiento tienen en el mundo, como por ejemplo, Netflix, Ebay, Spotify, Twitter, Reddit, SoundCloud o Digg entre otros.

9.2 Características Principales

Cassandra, al tener un modelo de datos tan similar al presentado por Google en BigTable, comparte muchas de las características con las que se han expuesto en el apartado anterior de HBase. Aunque como se verá, a nivel arquitectónico son dos sistemas muy diferentes, por lo que resulta interesante analizarlas por separado (Lakshman and Malik 2009).

Se puede afirmar que Cassandra y HBase son las dos bases de datos orientado a columnas que más se utilizan de todas las existentes.

9.2.1 Distribuido

Cassandra es distribuido, lo cual significa que puede mantener cuantos nodos sean necesarios para su correcto funcionamiento, pero a un cliente o una aplicación externa siempre parecerá un sistema con un punto de entrada único.

Cassandra está especialmente diseñado y preparado para funcionar en modo distribuido. Esto no significa que no se puedan sacar buenos rendimiento de un único servidor con Cassandra instalado y funcionando, pero cuando realmente se obtienen beneficios de Cassandra es cuando está distribuido no solo por varias máquinas, sino por varios racks y por varios CPDs repartido por distintas zonas geográficas.

Además, como después se verá, es un sistema especialmente bien preparado para recibir escrituras, por lo que cualquier cosa que se escriba en cualquiera de estas múltiples ubicaciones, será silenciosamente propagada por todo el clúster.

9.2.2 Descentralizado

Que Cassandra sea descentralizado significa que es un sistema que ha sido pensado de forma que no exista ningún punto de único de fallo. Todos los nodos en Cassandra desempeñan la misma funcionalidad: no hay maestros, no existe un nodo que coordine las actividades que desempeñan los demás .

En la mayoría de las bases de datos distribuidas funcionan de forma que uno de los nodos recibe las escrituras, y el resto de nodos del clúster replican esa información y mantienen copias de lectura de la base de datos de cara a las aplicaciones. Incluso bases de datos relacionales poseen modos de funcionamiento que les permiten desarrollar una funcionalidad similar a la que se acaba de describir.

El problema de mantener una solución centralizada de este tipo es el punto de fallo único del sistema. Este tipo de arquitecturas están pensadas para escalar las lecturas de forma que la carga se reparta entre varias máquinas, pero ¿que sucede si se tiene un sistema que realiza muchas más escrituras que lecturas? Además, si en el esquema descrito anteriormente, el maestro falla, se está poniendo en riesgo el correcto funcionamiento del sistema completo.

Por tanto, la descentralización proporciona principalmente dos beneficios: funcionalmente es más sencillo que los sistemas maestro – esclavo, y te ayuda a evitar pérdidas en el servicio.

9.2.3 Alta Escalabilidad

La escalabilidad podría ser definida como la capacidad de un sistema para, ante un incremento considerable de la demanda en un corto espacio de tiempo, continuar ofreciendo el servicio sin degradar la calidad del mismo.

La escalabilidad horizontal es la mejor manera de alcanzar esta meta. Agregar más máquinas cuando el sistema se queda corto, unido a la filosofía de este tipo de sistemas de utilizar hardware de coste moderado, es la opción que mejor se perfila para empezar con un sistema pequeño y gastos menores, e ir ampliando el sistema a medida que las necesidades lo van haciendo necesario.

Por supuesto, también esta la opción de hacer un escalado “hacia abajo”, o lo que es lo mismo, sacar máquinas del clúster cuando debido a la poca carga que está recibiendo el sistema ya no son necesarias.

9.2.4 Tolerancia a Fallos

Cassandra es tolerante a fallos. Si uno de los nodos tiene un error de comunicaciones, el resto pueden cumplir su función sin problema, y en caso de que sea necesario, meter un nuevo nodo en lugar del que está fallando. Si el error es de hardware, el nodo puede ser apartado hasta que se solucione el problema que le afecta. Si es necesario, puede incluso ser totalmente retirado del clúster y sustituido por otro nuevo.

9.2.5 Eventualmente Consistente

Le eventualidad en la consistencia le viene “impuesto” por el teorema de CAP: Cassandra es un sistema AP. Esto significa que antepone la disponibilidad de la

información y la capacidad del sistema a funcionar con el sistema particionado, a la consistencia de los datos.

La consistencia básicamente hace referencia a la capacidad del sistema de devolver, ante una petición de lectura, siempre la información más actualizada que se había escrito con anterioridad.

Decir que Cassandra es eventualmente consistente no es del todo cierto. Cassandra es altamente parametrizable, y como parte de esta parametrización, el sistema permite incrementar la consistencia del sistema todo lo que se desee. Eso sí, a cambio de sacrificar disponibilidad.

La consistencia eventual es tan solo uno de los múltiples modelos que los arquitectos de sistemas pueden escoger para su instalación de Cassandra. Otras posibles configuraciones serían:

9.2.5.1 Consistencia estricta

La consistencia estricta significa que cada lectura que se realice en el sistema devolverá la escritura más recientemente recibida. Parece que es una solución perfecta, ¿verdad? En una máquina única y local en el que se realizan escrituras no es excesivamente complicado saber que es lo último que se ha escrito, pero en un sistema que permite escrituras en todos sus nodos, que probablemente esté distribuido geográficamente por varios CPDs, y que previsiblemente va a recibir un elevado número de escrituras, las labores de sincronización y de asegurar que las últimas escrituras serás las respuestas devueltas cuando se realice una lectura, tendrán un elevado impacto en la disponibilidad del servicio, o en su defecto, en los tiempos de respuesta del mismo.

9.2.5.2 Consistencia casual

La consistencia casual trata de relajar la carga de sincronización de la anterior solución aplicando una visión más semántica de las operaciones. El sistema "descubre" que operaciones de escritura están relacionadas de una u otra manera, y hace consistentes las llamadas de lectura relacionadas con esa información.

Si una escritura se produce tras otra ordenadamente, se puede deducir que están casualmente relacionadas. La consistencia casual dice que las escrituras casuales se deben leer de forma secuencial.

9.2.5.3 Consistencia débil o eventual.

La consistencia eventual dice que la información que se escribe en un nodo del clúster actualizará al resto de nodos en algún momento, pero que podría tomar algún tiempo. Eventualmente las replicas serán consistentes.

9.3 Arquitectura

9.3.1 Espacio de Claves

Cassandra mantiene un espacio en la base de datos llamado *system* que utiliza para gestionar la información sobre los metadatos. Este esquema almacena metadatos relacionados con el nodo local, como por ejemplo: el token del nodo, el nombre del clúster, el espacio de claves, la definición del esquema, etc.

Concretamente, existen dos column families importantes. Una gestiona el espacio de claves del usuarios y las definiciones de los esquemas, y el otro los cambios que se hacen sobre el espacio de claves. Son las *column families*: *Schema* y *Migrations*.

9.3.2 Peer to Peer

El modelo que propone Cassandra es un modelo basado en la comunicación *peer*-to-*peer* entre nodos. De esta manera se consigue que todos los nodos sean funcionalmente idénticos entre ellos. Es decir, no existe un maestro que controle las escrituras, o cierta parte del clúster.

De este modo, el diseño peer-to-peer hace el el sistema escale de una forma mucho más sencilla, además de mejorar la disponibilidad de los datos, ya que la pérdida de un nodo no supone una degradación del servicio demasiado elevada.

Agregar nuevos nodos al clúster se convierte en una tarea casi trivial, puesto que todos los nodos desempeñan el mismo rol, lo único que hay que hacer es añadir un nodo nuevo al clúster y él solo aprenderá la topología existente en ese momento, y cuando esté preparado, se unirá al anillo de servidores para ser un miembro más del anillo.

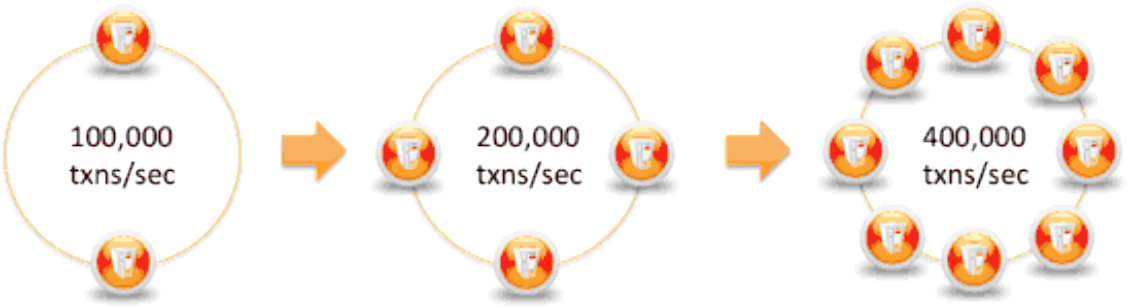

Ilustración 23. Representación de la escalabilidad casi lineal de Cassandra[55]

9.3.3 Particionado

Cassandra, al igual que DynamoDB, particiona la información a través de todo el clúster utilizando Consistent Hashing. Los fundamentos y bases del Consistent Hashing ya han sido expuestos en la sección anterior del Estado del Arte.

Que Cassandra utilice Consistent Hashing para particionar sus datos significa que cada nodo es responsable de una parte de los datos contenidos en la base de datos. Igualmente, la marcha o entrada de un nodo en el sistema afectará únicamente a sus vecinos directos en el anillo, que tendrán que hacerse cargo o repartir parte del anillo con sus nodos vecinos.

9.3.4 Replicación

Cada dato de la base de datos es replicado N veces entre los nodos del clúster, donde N es el factor de replicación, y se configura individualmente por instancia.

---

55 Imagen extraída de http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html

Como se ha explicado en la sección anterior, cada conjunto de claves se asigna a un servidor, que será el encargado de gestionar la replicación de esa parte concreta de la base de datos al resto de miembros que conforman el sistema.

Cassandra proporciona diversas políticas de replicación, tales como políticas sensibles al rack donde están ubicados los datos, o sensibles al CPD, de forma que al final se consiga tener la información distribuido por distintos racks de un mismo CPD, y distintos racks de otros CPDs.

Además, Cassandra proporciona garantías de disponibilidad de la información en presencia de errores a nivel de red y de CPD. Problemas de refrigeración o ventilación en los centros de datos, o problemas de red internas en los CPDs, son salvados por Cassandra permitiendo que los datos se distribuyan homogéneamente entre CPDs evitando así pérdidas de servicio.

9.3.5 Gossiping

Para poder tener un sistema descentralización y tolerante a particiones, Cassandra hace uso de un protocolo de *Gossiping* para hacer que sus nodos se comuniquen entre ellos y obtengan información los unos de los otros.

Primero se explicará este término. Los algoritmos de *Gossiping,* a veces también llamados *"protocolos epidémicos"* tienen muy buen rendimiento, especialmente en grandes clústeres. *Gossip* en castellano se traduce como "rumor", y recibe este peculiar nombre porque la forma que tiene el sistema para divulgar la información se asemeja bastante a la forma en la que las personas transmitimos un rumor o cotilleo.

En Cassandra, cuando un nodo arranca, la clase *Gossiper* será la encargada de gestionar todas las comunicaciones con el resto de nodos de aquí en adelante. Esta clase mantendrá una lista de nodos vivos y muertos, y actúa de la siguiente manera:

Cada cierto tiempo, el *gossiper* elige un nodo del anillo de forma aleatoria e inicia una comunicación con él. Estas comunicaciones se basan en el intercambio de tres mensajes inspirados en el método de apertura de conexión de las conexiones TCP: primero se envía un mensaje *GossipDigestSynMessage*, a continuación el receptor responde con un *GossipDigestAckMessage* y finalmente el emisor nuevamente responde, esta vez con un *GossipDigestAck2Message*.

De esta manera, si el nodo objetivo no responde estos mensajes, se deducirá que es un nodo inaccesible y pasará a formar parte de la lista de nodos "muertos". Al final, el conjunto de nodos

9.3.6 Anti entropía

La anti entropía es un mecanismo por el cual los sistemas distribuidos de bases de datos son capaces de asegurar que los datos que contienen sus nodos están en la última de las versiones.

DynamoDB utiliza los árboles Merkle para lograr este objetivo. Cassandra también lo utilizará, pero su implementación es ligeramente distinta, ya que Cassandra mantiene un árbol Merkle por cada *column family*.

9.3.7 Procedimiento de escritura

Puesto que Cassandra es un sistema que proporciona alto rendimiento para las escrituras, parece interesante analizar este proceso con más detenimiento, describiendo con más detalle los aspectos más relevantes del proceso (Hewitt 2011).

9.3.7.1 Commit Log

El proceso de escritura en la base de datos debe pasar por varias fases antes de que sea efectivo. El primer paso antes de llevar a cabo la escritura es el de dejarla reflejada en el *commit log*. De esta manera se asegura la consistencia de los datos incluso en el caso de que suceda un problema y la operación se quede a medias.

9.3.7.2 Memtable

Después de haber escrito en el *commit log*, el dato es creado el *memtable*, es decir, en una estructura en memoria preparada para el efecto. Cuando el número de objetos contenidos en un *memtable* supera el límite impuesto, se vuelcan los datos a disco y se crea un nuevo *memtable*. Una vez que esta operación se ha completado, se informa al *commit log* de que la operación ya ha sido finalizada correctamente, para que elimine esa entrada de sus registros.

9.3.7.3 SSTable

Cuando las *memtables* se vuelcan a disco, éstas lo hacen sobre las *SSTables*. Estas tablas son inmutables y no pueden ser modificadas por la aplicación.

Todas las escrituras en Cassandra se realizan de forma secuencial. Es por esto por lo que el sistema experimenta un tan alto rendimiento en lo que a escrituras se refiere. No se requieren de lecturas previas o posicionados aleatorios en disco, tan solo escribir de forma secuencial.

9.3.8 Tombstones

Los *tombstones* son elementos que se utilizan en Cassandra cuando se quiere eliminar un objeto del sistema (su significado literal en castellano es "*lápida*").

En lugar de eliminar físicamente la información del clúster, el procedimiento que lleva a cabo es el de "marcar" la información como borrada. Esta marca de borrado, junto con todos los datos a los que esta marca está haciendo referencia, serán borrados la próxima vez que el proceso de compactación tenga lugar.

Como dato curioso, vale la pena apuntar que el valor por defecto que toma el tiempo entre que una recolección de *tombstones* tiene lugar y la siguiente es de 10 días.

9.3.9 Compactación

La compactación es un proceso de agrupación de datos y liberado de información. Se produce cuando el número de *SSTables* en disco comienza a ser elevado. La compactación realiza una fusión de todas las *SSTables* que se han estado escribiendo a disco desde que la última compactación fue procesada: las claves son entrelazadas, las columnas son combinadas, los *tombstones* son eliminados y se recrean de nuevo los índices.

El objetivo de hacer esto es el de liberar espacio ocupado en disco innecesariamente, unificar los datos y recrear los índices para que la información pueda ser fácil y rápidamente accesible. Toda esta información ordenada y unificada es nuevamente escrita en una única *SSTable*.

Además, en Cassandra existe un límite de *SSTables* que un proceso puede consultar para encontrar el dato que busca. Si una clave es modificada frecuentemente, es probable que se alcance este límite. Realizar una labor de compactación cada cierto tiempo ayuda a mantener la velocidad y la estabilidad del clúster en general y de cada nodo en particular.

9.3.10 Filtros Bloom

Los filtros Bloom (llamados así en honor a su creador, Burton Bloom), son un mecanismo que emplea Cassandra para mejorar el rendimiento del clúster.

Los filtros Bloom son un tipo de algoritmos no deterministas que determinan si un objeto dado se encuentra dentro de un conjunto. Que estos algoritmos sean no deterministas significa que nunca darán un falso negativo, pero que sí pueden dar un falso positivo. Es decir, en caso de que, como resultado de la aplicación del filtro, un elemento no resulte ser miembro de un conjunto, no hay opción a la duda, pero en caso de que sí lo sea, el siguiente paso debe ser ir a comprobarlo.

Los filtros Bloom funcionan manteniendo en memoria un array reducido de información que representa toda una colección completa de datos. Este array será un resumen de los objetos que pertenecen a esa colección, y debe tener un tamaño tal que se pueda mantener siempre en memoria. Con este array en memoria y los filtros de Bloom, el objetivo es tratar de minimizar en la medida de lo posible que se realicen accesos innecesarios a disco.

9.3.11 SEDA

SEDA son las siglas de *Staged Event Driven Architecture*, o lo que es lo mismo, Arquitectura Orientada a Eventos por Etapas.

Cassandra implementa SEDA, una arquitectura propia de los servicios de Internet que tienen unas necesidades muy grandes a la hora de escalar (Welsh 2006).

En una aplicación típica, habitualmente para gestionar operaciones entrantes al sistema, la forma de proceder es: se crea un hilo que atiende esa petición entrante, el hilo hace el trabajo que tenga que hacer, devuelve el resultado de la petición al cliente y muere. Con SEDA esto no es exactamente así.

Con SEDA un hilo puede comenzar a atender una petición, después pasarle el trabajo a otro hilo que continua realizando la parte que le corresponda de la operación, y finalmente podría delegar en otro hilo diferente la finalización de la misma.

SEDA propone ciertas bases:

1. El trabajo es dividido en etapas o *stages*.
2. Se crean pools de hilos que serán los encargados de ejecutar las peticiones que lleguen al sistema.

De este modo, las etapas serían la unidad mínima en la que se dividen las peticiones, y éstas se asocian a pools de hilos que ejecutan todas estas tareas.

Puesto que cada tarea puede ser asociada a un pool de hilos distinto, esta técnica hace que Cassandra obtenga un aumento del rendimiento de manera considerable.

Además, los pools de hilos son conscientes del hardware del que disponen, y harán asignaciones de recursos entre los hilos que más lo necesiten. Es decir, si un hilo está haciendo un trabajo intensivo a nivel de entrada / salida, u otro esta ocupando mucho espacio en memoria principal, o hay un thread que está empleando muchos ciclos de CPU, etc., el pool conoce los recursos de los que dispone y sabe asignarlos entre los hilos en función de las necesidades de cada uno.

9.4 Modelo de Datos

Cassandra es una bases de datos NoSQL orientada a columnas. Su modelo de datos se basa en una serie de column families que a su vez se contienen columnas, que pueden darse ninguna, una o varias veces para un mismo elemento.

Cassandra, al igual que HBase, basa su modelo de datos en el paper sobre Bigtable que liberó Google en 2006 (Chang, et al. 2006).

Cassandra y HBase poseen muchos aspectos comunes en lo que al modelo de datos se refiere, por lo que aquí se intentará no repetir lo ya expuesto con anterioridad pero sí dar una visión de lo que nuevo que aporta Cassandra.

En esta sección se hará un repaso por el modelo de Cassandra de arriba hacia abajo, es decir, desde la agrupación más grande que existe en el sistema que es el clúster, hasta las columnas, que es el elemento más básico del que se componen los objetos de la base de datos.

9.4.1 Clúster

Cassandra es una base de datos que está diseñada e ideada para funcionar formando clústers de máquinas. Es decir, si se quiere aprovechar al máximo todo el rendimiento y la funcionalidad que nos ofrece esta base de datos, no se deberá correr en un único nodo, sino de forma distribuida en varios nodos. Y cuanto más distribuido mejor: si es en varios racks mejor, y si además se ubican en varios CPDs, Cassandra dará un mejor rendimiento a la aplicación.

El protocolo peer-to-peer hace que los nodos se comuniquen entre ellos para conocer su estado y repliquen los datos de forma transparente a usuario.

9.4.2 Keyspaces

Un clúster está conformado por uno o más *keyspaces*. En un sistema de bases de datos relacionales, un *keyspace* correspondería con una *database*.

Cada *keyspace* tiene asociado un conjunto de atributos que definen el comportamiento global del *keyspace*.

Los atributos más importantes que se pueden configurar son:

- Factor de replicación

El factor de replicación hace referencia al número de copias que debe guardarse de cada dato de la base de datos. El factor de replicación indicará cuantos nodos tendrán copia de los objetos de los que ese nodo sea responsable.

Este parámetro impacta directamente en el nivel de consistencia y de rendimiento del sistema. A mayor factor de replicación, más rendimiento y menos consistencia.

- Estrategia de localización de replicas

Este parámetro establece qué política se seguirá para alojar las replicas en el anillo, y para decidir qué nodos serán los que alojen qué copias.

Las políticas de localización de replicas tienen en cuenta aspectos como la tomar en consideración el rack donde están alojados los servidores, los centros de datos, la topología de la red, etc.

- Column families

Los *keyspaces* en Cassandra actúan como contenedores de *column families*, de igual manera que en el modelo relacional una *database* sería un contenedor de tablas.

Cada column family contendrá varias columnas, y las filas se conforman de estas estructuras para almacenar la información final.

9.4.3 Column Families

Las column families han sido ampliamente tratas en anteriores secciones, por lo que no se darán demasiados detalles teóricos sobre ellas aquí.

Cuando se escribe en una column family, se especifican valores para una o más columnas. Al conjunto de esos valores, junto con una clave identificativa, es lo que se le llama fila. En Cassandra, la estructura de datos es frecuentemente llamada hash cuatridimensional se estructura de la siguiente forma.

[Keyspace][ColumnFamily][Key][Column]

Viéndolo desde el punto de vista de las filas, una column family estaría estructurado de la siguiente manera

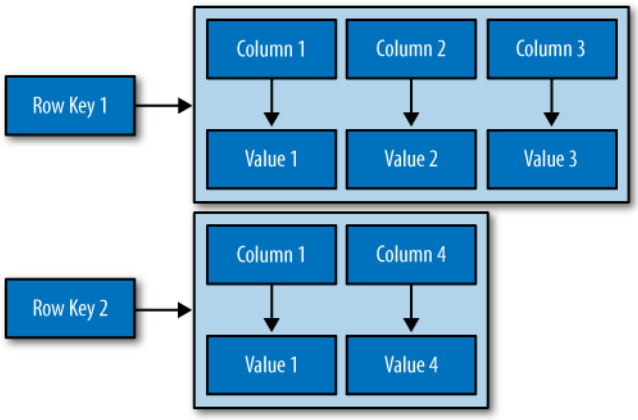

Ilustración 24. Diseño de una Column Family

Este sería un ejemplo de una column family llamada *Hotel* que almacena hoteles así como información relacionada con el mismo:

```
Hotel {
     key: AZC_043 {
          name: Cambria Suites Hayden, phone: 480-444-4444,
          address: 400 N. Hayden Rd.,
          city: Scottsdale,
          state: AZ, zip: 85255
     }key: AZS_011 {
          name: Clarion Scottsdale Peak,
          phone: 480-333-3333,
          address: 3000 N. Scottsdale Rd,
          city: Scottsdale,
          state: AZ, zip: 85255
     }key: CAS_021 {
          name: W Hotel,
          phone: 415-222-2222,
          address: 181 3rd Street,
          city: San Francisco,
          state: CA, zip: 94103
     }
}
```

La clave key en este caso es una clave primaria formada por números, letras y símbolos especiales. La column family se llama "Hotel", y cada una de las columnas de esa column family son las que aparecen con valores: *name*, *address*, *city*, etc.

9.4.4 Columas

Las columnas son las unidades mínimas que alojan información en Cassandra. La información que contiene el valor de cada columna está conformado por un array de bytes.

Además, en Cassandra no es necesario definir todas las columnas en el momento de diseñar la base de datos, pero sí las column families. Una vez éstas estén creadas, el resto de tablas se pueden crear en tiempo de ejecución, incluso aunque previamente no existieran.

9.4.5 Supercolumnas

Las supercolumnas son un tipo especial de columna. Ambos tipos de columnas son pares nombre-valor, pero mientras que una columna normal almacena el valor para esa columna, una super columna contiene un mapa de columnas, que estas ya sí, almacenan el valor en forma de array de bytes para esa columna.

Con las superscolumnas, Cassandra pasa de ser un hash de 4 dimensiones a serlo de 5, con la siguiente ordenación en sus dimensiones.

[Keyspace][ColumnFamily][Key][SuperColumn][Subcolumn]

Para definir una supercolumna, simplemente hay que generar una column family como si fuera de tipo *super*. De esta manera, este tipo de column families se pueden utilizar como si fueran un tipo de column family normal, únicamente habría que indicar explícitamente el nombre de la super columna.

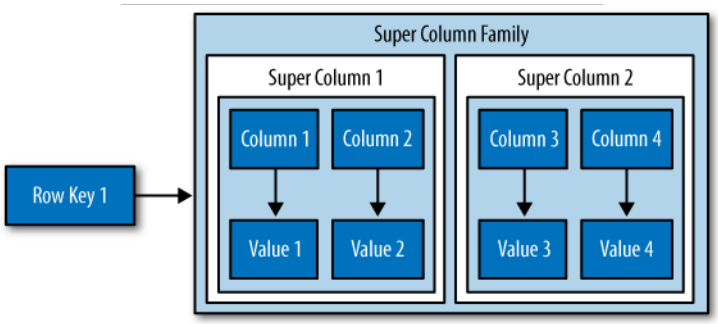

Ilustración 25. diseño de una súper columna

A continuación se expone un ejemplo de la supercolumna *PointOfInterest*, que indica lugares de interés cercanos a los hoteles.

```
PointOfInterest (SCF)
     SCkey: Cambria Suites Hayden{
          key: Phoenix Zoo{
               phone: 480-555-9999,
               desc: They have animals here.
          },
          key: Spring Training{
               phone: 623-333-3333,
               desc: Fun for baseball fans.
          }
     }
}
```

Una cosa a tener en cuenta es que no se pueden generar índices de los datos almacenados en las súper columnas, sino que cada vez que se carga una súper columna en memoria, se cargan todas las columnas asociadas también.

9.5 Ejemplos reales de uso del SGBD

9.5.1 Hulu

Hulu es un conocido servicio americano de streaming de películas y series. Una especie de videoclub virtual mediante el cual tienes derecho a ver cierto contenido con una tarifa plana que se consigue pagando una suscripción.

Andres Rangel, ingeniero de software en Hulu, cuenta que en Hulu utilizan Cassandra para almacenar, entre otras cosas, el historial de visualizaciones en tiempo real de sus usuarios, de cara a utilizar esta información por otras aplicaciones. Cassandra se utiliza como base de datos de persistencia redundada por varios CPDs.

"*Cassandra, ofrece buen rendimiento, y una escalabilidad cercana a la lineal par anuestro modelo de datos y geo replicación, y todo ello con unos requisitos de mantenimiento mínimos. Tambien llegamos ae valuar HBase y Riak pero finalmente nos decidimos por Cassandra porque consideramos que cubre mejor nuestras necesidades.*" Andrés Rangel (Planet Cassandra 2014)

Cassandra actualmente cuenta con 5 millones de suscriptores y unos 30 millones de accesos al mes.

9.5.2 Spotify

Spotify es un servicio de escucha de música en streaming, presente en más de 50 países, que cuenta con 20 millones de canciones, donde diariamente se añaden 20.000 canciones y que cuenta con unos 40 millones de usuarios activos, entre los cuales, 10 millones de ellos son usuarios que poseen suscripción de pago.

Spotify comenzó únicamente con postgreSQL como backend para la plataforma, pero a medida que furon creciendo y las necesidades de escalabilidad se fueron haciendo más fuertes comenzaron a mirar por nuevas alternativas.

Según cuenta Alex Liljencrantz, ingeniero backend en Spotify, "*básicamente, una vez que alcanzas varios CPDs, la replicación en tiempo real de postgreSQL no funciona especialmente bien con para escribir volúmenes de datos muy grandes. De esta forma, cuando comenzamos a buscar una solución BigData, Cassandra fue la que más cumplía nuestras expectativas, por su ausencia de puntos únicos de fallo, y lo que es más importante, su diseño basado en BigTable. Esto nos aseguraba un nivel de confianza de forma que no perderíamos datos, incluso siendo aun un proyecto muy joven. Básicamente, aun produciéndose bugs o errores, estaremos seguros de no perder nuestros datos*."

Existen varios servicios de Spotify que hacen uso de Cassandra, pero el que más datos almacena es un clúster con unos 50 terabytes de datos comprimidos.

10 Bases de datos documentales

A continuación se expondrá en detalle la arquitectura y el modelo de datos de uno de los SGBDs documentales más utilizados: MongoDB.

11 MongoDB

11.1 Introducción

MongoDB (derivado de la palabra inglesa "homongous") es una base de datos NoSQL Open Source **orientada a documentos**, y ha sido diseñada con la idea de que fuera fácil tanto desarrollar para ella como de ser administrada. La empresa 10gen es la que actualmente se encarga de su mantenimiento y desarrollo.

Pero, ¿Qué es una base de datos orientada a documentos? Un documento es una estructura de datos compuesta de pares de campos y valores. Además, estos objetos son muy similares a lo que serían objetos en notación JSON. La principal diferencia con respecto a los almacenes clave – valor es que los valores en MongoDB pueden ser otros documentos, arrays o incluso arrays de documentos.

Ilustración 26. Ejemplo de Documento en MongoDB

Pues bien, todos los registros en MongoDB son documentos. Utilizar documentos tiene ciertas ventajas:

- Los documentos son tipos de datos nativos en muchos lenguajes de programación.
- Los documentos embebidos y los arrays minimizan la necesidad de realizar joins muy pesados.
- Tener un esquema dinámico es la base del polimorfismo.

11.2 Características generales

Todas las características serán ampliamente desarrolladas en posteriores secciones. Aquí simplemente se dará una visión global del enfoque principal que se le ha querido dar a esta base de datos NoSQL (Chodorow 2013).

Indexado. MongoDB soporta índices secundarios, permitiendo una buena aceleración en muchas querys, pudiendo indexar cualquier campo de la base de datos.

Agregación. MongoDB viene con muchas funciones predefinidas que facilitan en gran medida el tratamiento de las colecciones almacenadas.

Replicación y Balanceo de Carga. MongoDB utiliza un sistema de replicación empleando la arquitectura maestro – esclavo con el resto de máquinas. Además, hace uso de un tipo de particionado horizontal llamado sharding. Estas dos herramientas de escalado hacen de mongo una herramienta poderosa cuando se trata de crecer mucho en poco tiempo.

11.3 Arquitectura

11.3.1 Replicación

La replicación en MongoDB adquiere un papel fundamental. Se llama replicación al proceso de sincronización de datos entre múltiples servidores. Básicamente consiste en tener copias idénticas de los datos en varios servidores.

Pueden existir varios motivos por los cuales se podría querer tener un entorno replicado de base de datos: incrementar la capacidad de lectura, proteger la base de datos de diferentes posibles caídas de servidores, backup, incrementar la localización y la disponibilidad de la información, etc.

La forma que tiene MongoDB de mantener replicación entre sus nodos es a través de lo que se conoce como **replica set**. Un replica set es un conjunto de servidores que siguen una arquitectura de maestro – esclavo. Existe un servidor maestro y uno o varios esclavos. El servidor maestro puede realizar operaciones de escritura y lectura, pero los esclavos únicamente permiten la lectura. Además, existen varios tipos de esclavos, cada uno de ellos con una serie de funciones y competencias (10gen 2014).

11.3.1.1 Nodo primario

Únicamente puede existir un nodo primario por cada replica set. Será el nodo principal o maestro y únicamente este nodo el que acepte las operaciones de escritura por parte de los clientes. Debido a esto, los replica set proporcionan consistencia estricta.

La forma que emplea el maestro para comunicar las actualizaciones a los nodos secundarios es mediante un log de operación llamado *oplog* donde se escriben todas las operaciones que modifican la base de datos.

Los nodos secundarios aplican las operaciones del *oplog* de forma asíncrona. Esto significa que entre que el maestro recibe una operación de escritura y el nodo esclavo escribe esos datos en su base de datos, puede pasar un tiempo indeterminado. Por tanto, los nodos secundarios en algunos momentos podrían no devolver a los clientes la información más actualizada.

11.3.1.2 Nodos secundarios

Los nodos secundarios son los nodos que se comunican con el nodo primario, reciben el *oplog* y lo aplican en su base de datos local.

Si el nodo principal queda inaccesible, los nodos secundarios (junto con otros tipos de nodo que se verán más adelante) efectuarán lo que se conoce como elecciones. Las elecciones serán comentadas en una sección posterior con más profundidad, pero

básicamente destacar que estas elecciones tienen lugar con el objetivo de encontrar un nuevo nodo principal para el clúster.

Aparte de los nodos secundarios tal y como se han visto hasta ahora, pueden darse dos configuraciones específicas que derivan en dos nodos secundarios especiales:

- **Los nodos secundarios ocultos**: Son un tipo de nodo invisible para las aplicaciones, y además no pueden ser elegidos nodos primarios en las elecciones (tienen la prioridad a cero). Sin embargo, sí disponen de voto en caso de que sucedan elecciones. Puesto que no reciben ni sirven más trafico que el propio de la replicación, son un tipo de nodos que se emplean sobre todo par labores de reporte y backup.
- **Los nodos secundarios retardados**. Este tipo de nodos son similares a los ocultos, en el sentido en el que deben ser invisibles para las aplicaciones, no pueden ser elegidos como nodo maestro en ningún caso, y además disponen de voto en caso de elecciones. La principal característica de este tipo de nodos es que mantienen una versión histórica de la base de datos. Es decir, llevan un cierto retardo en la replicación, lo cual los hace perfectos para mantenerlos como nodos backup de corto plazo. Actualizaciones de software erróneos, errores en operaciones, bases de datos borradas por error, etc.

11.3.1.3 Elecciones

Los replica set utilizan las **elecciones** para elegir un nuevo nodo principal cuando se ha perdido contacto con éste.

Las elecciones son esenciales para el correcto funcionamiento de la plataforma. A través de ellas, el sistema se auto gestiona y responde a fallos del nodo primario de forma completamente autónoma. Pero las elecciones toman su tiempo en ser ejecutadas. Desde el momento en el que el maestro deja de estar disponible hasta el momento en el que uno nuevo adquiere su nueva función pasa un tiempo, tiempo en el cual el sistema no tiene nodo principal y, por lo tanto, no está aceptando que ningún proceso escriba en la base de datos.

11.3.1.3.1 Funcionamiento de las elecciones

Los miembros del replica ser tienen un *heartbeat* configurado que envía pings cada dos segundos. Si el heartbeat no es devuelto en 10 segundos, se marcará el nodo como no disponible y se iniciará el proceso de elección.

El parámetro `priority` es un parámetro que tienen todos los miembros del replica set, y que contiene un valor numérico. Cuanto más alto sea este valor en un nodo, más posibilidad tiene ese nodo de salir elegido. Por el contrario, una prioridad igual a cero imposibilita por competo a un nodo para que sea elegido nodo primario.

Las elecciones no serán convocadas mientras el nodo primario sea el que tiene mayor número de prioridad.

Aunque esto es así, y teniendo en cuenta que la prioridad puede ser asignada a mano, lo habitual en la construcción del replica set es que todos los nodos cuenten con una prioridad por defecto, por lo que todos tendrán las mismas posibilidades de salir elegidos.

Todos los miembros tipos de nodos que se han comentado con anterioridad tienen derecho a voto cuando se forman elecciones, y además, un nuevo nodo que no se ha comentado hasta ahora: los árbitros.

11.3.1.3.2 Nodos Árbitro

Los nodos árbitro, a diferencia de los nodos secundarios, no realizan replicación de los datos. Es decir, no mantienen una versión de la base de datos copia de la que tiene el maestro.

Los nodos árbitro existen únicamente para tener otro miembro del replica set que pueda emitir un voto. Esto es, para tener nodos que puedan participar en las elecciones y decidir en un posible empate de votos, sin la necesidad de añadir la sobrecarga de un nuevo nodo replicando datos del maestro.

Es decir, los nodos árbitro se crean cuando el número de nodos que deben participar en una votación es un numero par, puesto que, idealmente, las votaciones se deben realizar con un número impar de nodos.

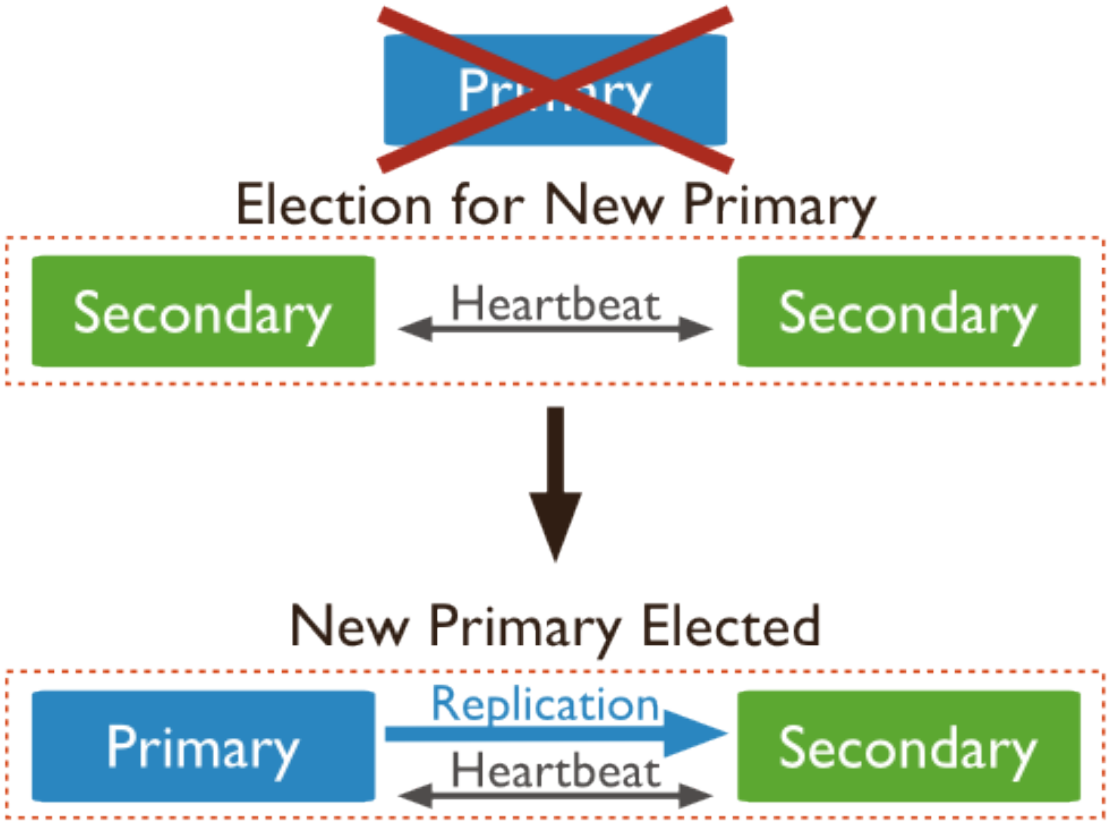

Ilustración 27. Caída del nodo primario y elección para obtener uno nuevo

11.3.2 Sharding

El sharding hace referencia al proceso de repartir los datos de la base de datos entre múltiples nodos. El sharding es lo que tradicionalmente se conoce como particionado horizontal.

Mientras que la replicación busca realizar copias de los datos y repartirlos por nuevos hosts, permitiendo ofrecer alta disponibilidad, balanceo de carga y copias automáticas de seguridad, el sharding busca particionar esas tablas que, previsiblemente, se van haciendo más grandes y cada vez son más costosas de replicar, ocupan demasiado en disco, y están empezando a experimentar demasiadas operaciones simultáneas (10gen 2014).

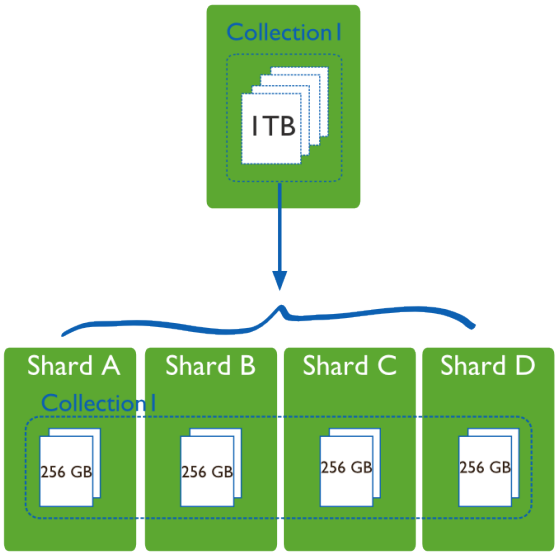

Ilustración 28. Colección dividida en 4 shards

El sharding divide el conjunto de datos y lo distribuye entre múltiples servidores. Estos servidores también son conocidos como **shards**. Cada shard es una base de datos independiente las cuales, en conjunto, conforman una base de datos única.

11.3.2.1 Componentes de un sharded clúster

Cuando se crea un clúster de MongoDB con shards se crean nuevos roles y nuevas máquinas. Entre ellas:

11.3.2.1.1 Shards

Un shard puede ser un host con una instancia de MongoDB corriendo, o puede ser un replica set, la arquitectura de replicado descrita anteriormente.

Lo habitual es que en entornos productivos se proporcione la redundancia y disponibilidad en cada shard que ofrecen los replica set.

De entre todos los shards, existe un shard especial: shard primario. Este shard en principio se hará cargo de todas las colecciones que no han sido particionadas. El primer shard que se configura cuando se comienza a particionar el clúster será el shard principal.

11.3.2.1.2 Config Servers

Los *config servers* son instancias especiales de MongoDB. Estos servidores almacenan metadatos relacionados con el clúster, información que refleja el estado actual del sistema y de los datos. Estos metadatos incluyen los listados de chunks que está alojando cada shard, o los rangos de cada uno de los chunks.

Los *config servers* no pueden formar replica sets, y deben estar todos disponibles para realizar cualquier cambio en los metadatos del clúster.

Mientas que en entornos previos el número de *config servers* no es demasiado relevante, se recomienda utilizar una configuración de exactamente 3 de estos servidores en entornos de producción. Hay que tener en cuenta que si se pierde la información que estos servidores alojan, el acceso al resto del clúster quedará completamente inoperativo. Hay que evitar en la medida de lo posible que esta parte del clúster tenga un punto único de fallo.

Como se ha mencionado con anterioridad, el sistema necesita de todos los config servers que se configuraron en la creación del clúster para poder funcionar con normalidad. Si alguno de los servidores queda inaccesible, el conjunto de metadatos pasará a modo lectura. Incluso si todos los servidores de metadatos caen, el sistema podrá seguir funcionando hasta que el resto de componentes se reinicien. Pero no se podrá realizar ningún cambio en los metadatos hasta que los tres servidores vuelvan a estar operativos (10gen 2014).

11.3.2.1.3 Instancias de Routing

El tercer y último tipo de instancia que conforman los sharded clústers de MongoDB son las instancias de routing.

Las aplicaciones externas no pueden contactar directamente con los servidores que tienen los chunks con la información, sino que tienen que contactar con estos servidores de routing que son los que hacen las redirecciones pertinentes.

Los servidores de ruteo funcionan de la siguiente manera:

1. Una petición llega a la instancia de routing
2. Los servidores de ruteo contactan con los servidores de metadatos (los *config servers*) y cachearán la información para ahorrar peticiones. En caso de que la tuvieran cacheada de antes no sería necesario realizar este paso.
3. A continuación, utilizarán esta metadata para enrutar las operaciones de las aplicaciones a las instancias que contienen los datos.
4. Los servidores de routing no tienen estado persistente, por lo que el consumo de recursos es mínimo. En algunas implementaciones se suelen poner en las máquinas en las que reside la propia aplicación, o incluso en los servidores donde se alojan los shards.

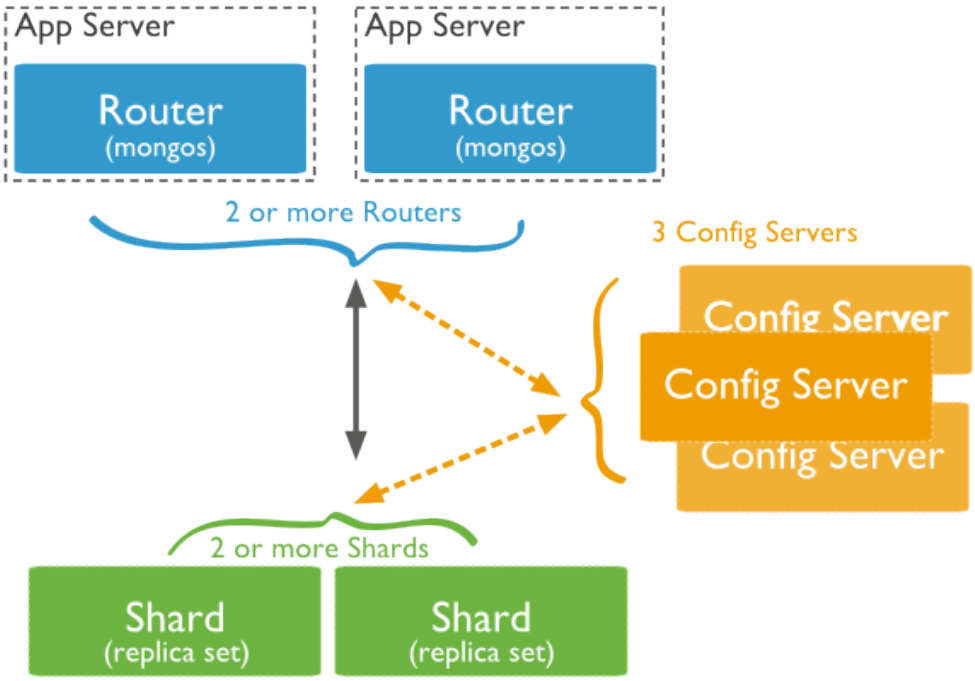

Ilustración 29. Arquitectura de un sharded clúster

11.3.2.2 Particionado de datos

MongoDB distribuye los conjuntos de datos o shards partiendo de las colecciones de la base de datos.

Existen diferentes mecanismos para decidir que criterio se va a seguir para dividir y repartir los datos entre todas las máquinas.

Para realizar un particionado MongoDB se basa en un sistema de claves. Se escoge un campo que sea común a todos los elementos de la colección que se desea particionar, y MongoDB divide los valores de ese campo entre los chunks. A continuación, se distribuyen los chunks entre todos los servidores de shard con los que se cuenta de una forma más o menos homogénea.

Existen dos formas de dividir las colecciones (10gen 2014):

11.3.2.2.1 Sharding basado en rangos

En el sharding basado en rangos, MongoDB divide la colección en tramos o chunks en función de los valores del campo que se haya elegido. Los chunks generados son conjuntos de documentos no solapados, y vienen determinados por un valor máximo y otro mínimo de la clave elegida como *range key*.

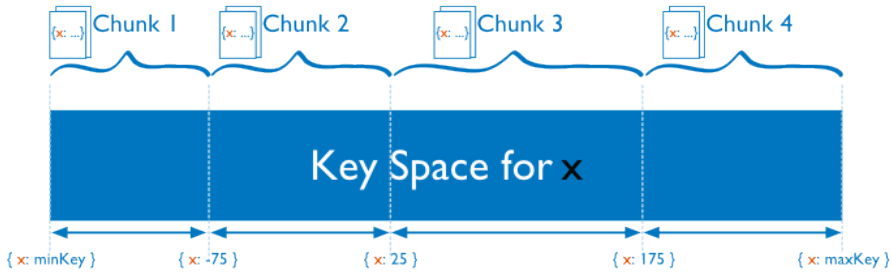

Ilustración 30. Ejemplo de sharding basado en rangos

11.3.2.2.2 Sharding basado en hashes

En el sharding basado en hashes, MongoDB genera las claves hash de los valores de un campo concreto común a todos los documentos, y entonces utiliza estos hashes para generar los chunks.

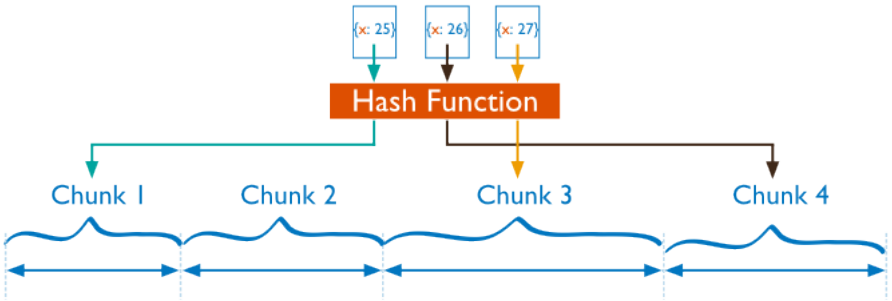

Ilustración 31. Ejemplo de sharding basado en hashes

Las diferencias entre ambos sistemas son:

El sharding basado en rangos…

… tiene una ganancia de rendimiento en búsquedas de documentos secuenciales.

… sin embargo, esto mismo puede resultar contra producente. Supóngase un proceso que siempre trata de acceder al mismos rango de dato. Realmente aquí, el sharding carecería de sentido, puesto que siempre estaría haciendo las querys sobre el mismo chunk.

El sharding basado en hashes…

… asegura un reparto más homogéneo de los datos. Este tipo de asignación distribuye los datos aleatoriamente entre los chunks y por tanto entre los shards. Pero a la vez, asegura un balanceo más igualitario a la hora de procesar operaciones en la base de datos.

11.4 Modelo de datos

MongoDB es un tipo de base de datos de esquema flexible. Esto significa que ha de ser el usuario el que el que determina y declara el esquema de cada una de las tablas justo en el momento de realizar las inserciones. Esta funcionalidad la comparten en cierta forma muchas bases de datos NoSQL.

Esto es bueno y malo a la vez. Por un lado se tiene la flexibilidad de añadir campos cuando son necesarios sin haberlos tenido que haberlos definido de antemano, o obviar algunos en los casos en los que no sean necesarios, pero a la vez se obliga al programador a ser mucho más metódico en su trabajo, puesto que el sistema no avisará de un posible error o despiste en el código.

Las bases de datos en MongoDB residen en un host que puede alojar varias bases de datos de forma simultánea almacenadas de forma independiente. Cada una de estas bases de datos puede contener una o más **colecciones**, a partir de las cuales se almacenarán los objetos o documentos.

11.4.1 BSON

BSON es un formato binario que se utiliza para almacenar la información en MongoDB. BSON es la codificación binaria del formato JSON. Esta codificación ha sido elegida porque presenta ciertas ventajas a la hora de almacenar los datos, como la **eficiencia** o la **compresión**.

Básicamente BSON y JSON son los formatos con los que trabaja MongoDB: JSON es el formato con el que se presenta la información a los usuarios y a las aplicaciones y BSON el formato que utiliza MongoDB de forma interna.

11.4.2 Estructura del documento

El reto principal que tienen los documentos como objetos en MongoDB es el de realizar un buen diseño de los mismos para que sean capaces de representar lo más ampliamente el mundo real. Para ello existen 2 herramientas que permiten a las aplicaciones representar las relaciones entre los datos: las referencias y los datos embebidos.

Una de las decisiones a tomar a la hora de implementar una aplicación que utilice MongoDB como capa de backend de aplicación es la de referenciar o embeber los documentos que se necesite.

11.4.2.1 Referencias

Las referencias almacenan relaciones entre los datos mediante enlaces entre documentos.

Los modelos de datos normalizados utilizan relaciones referenciales entre los documentos. Se usarán modelos de datos normalizados cuando:

- Embeber documentos produzca una duplicación de la información y los costes de mantenimiento sean mayores que las ganancias en lecturas.
- Sea necesario representar un modelo “varios a varios” con cierta complejidad.
- Para modelar conjuntos de datos de mucho tamaño.

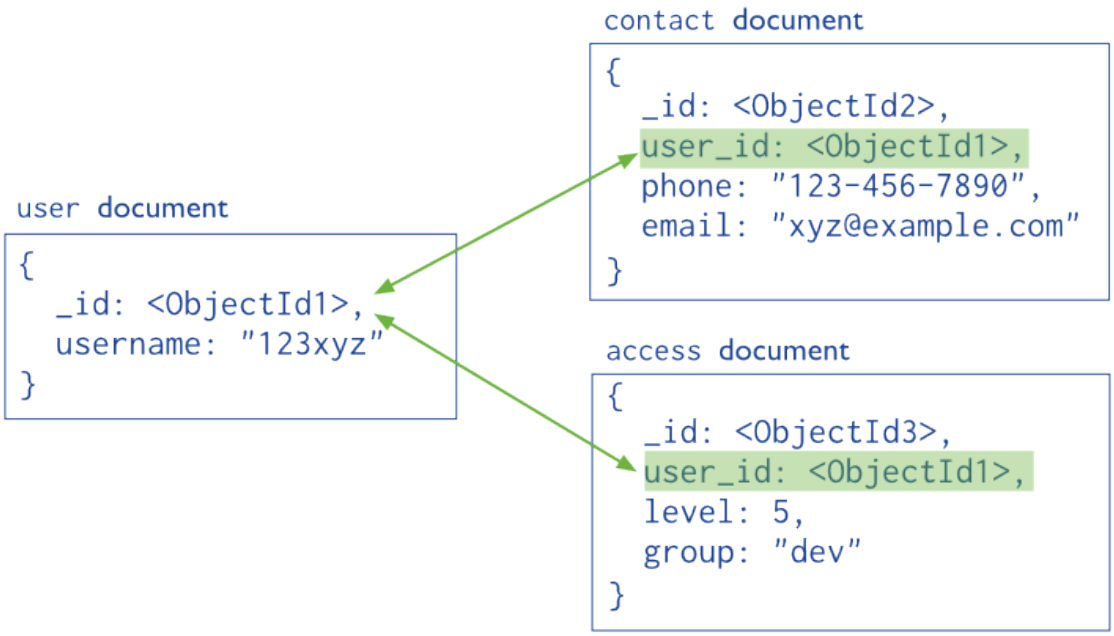

Ilustración 32. Modelado de datos utilizando enlaces entre documentos

11.4.2.2 Datos embebidos

Los documentos embebidos representan relaciones entre los datos almacenando la información relacionada bajo el mismo documento. Los documentos permiten embeber estructuras documentales como subdocumentos dentro de un campo o de un array.

En general, se utilizarán modelos embebidos cuando:

- Existan relaciones contenidas entre dos entidades. Estos es, que dos entidades independientes sean habitualmente accedidas a través de sólo una de ellas. Por ejemplo, *Persona* y *Dirección*. Cuando únicamente se permitan búsquedas por *Persona* para obtener su dirección. En ese caso, es preferible tener el objeto *Dirección* embebido dentro del de *Persona*.
- Cuando exista una relación "uno a varios" entre entidades. En este caso, la parte de "varios" suele ir embebida dentro de la de "uno". Los motivos son similares al anterior caso.
- En general, en ambos se trata de minimizar en la medida de los posible las operaciones de la base de datos. Embebiendo la información se consigue leer o escribir todos los datos simultáneamente y de forma secuencial.

Ilustración 33. Documentos embebidos en otro

Por otro lado, esta será una peor solución cuando los documentos crecen mucho después de su creación, puesto que se producirá fragmentación y las labores de mantenimiento del documento repercutirán directamente en el rendimiento a la hora de operar con él.

11.4.3 Índices

Las buenas decisiones en lo que a creación de índices se refiere será lo que determine, llegados a un volumen medio-alto de información, que una base de datos en MongoDB tenga unos tiempos de consulta razonables o los tenga muy elevados.

Los índices se utilizan para mejorar el rendimiento de las consultas. Es trabajo del administrador de base de datos construir índices en los campos que sean frecuentemente accedidos. Por defecto, MongoDB crea un índice sobre el campo *_id*, el identificador de cada documento.

A la hora de crear índices habrá que tener en cuenta una serie de consideraciones (10gen 2014):

- Cada índice requiere de al menos 8KB en disco. Cada índice activo ocupa espacio tanto en disco como en memoria. Hay que tenerlo en cuenta puesto que en algunos casos este espacio puede ser muy elevado.
- Los índices tienen un impacto negativo para las operaciones de escritura. Habrá que evaluar el uso que se le va a dar al sistema, puesto que los sistemas que reciba muchas escrituras y pocas lecturas, requerirán también tareas de actualización para cada índice que exista.
- Los índices no afectan para nada en las operaciones de lecturas sobre campos del documento que no han sido indexados.

11.5 Usos reales del SGBD

11.5.1 Foursquare

Foursquare es una red social cuya actividad consiste en permitir a sus usuarios realizar *check-in* en lugares concretos (marcar lugares como visitados a medida que el usuario los visita y compartir la localización con amigos y contactos).

Cuando Foursquare decidió migrar a MongoDB, su capa de backend consistía en una única base de datos relacional.

Debido al crecimiento exponencial que experimentó desde su creación en 2009 hasta pocos años después, los ingenieros de Foursquare decidieron evaluar, entre otras soluciones, bases de datos NoSQL. Finalmente encontraron en MongoDB la base de datos que les permitía solucionar tanto sus necesidades más inmediatas, como las , previsiblemente, pudieran surgirles más adelante.

Las características que apreciaron de MongoDB fueron:

- El particionado automático, mediante el cual simplemente tienes que añadir nodos al clúster a medida que se van necesitando y el software se encarga de lo demás.
- La indexación geográfica de la que dispone MongoDB, la cual les permitía realizar búsquedas basadas en una ubicación espacial concreta.
- Los ReplicaSet, mediante los cuales se consigue alta disponibilidad y redundancia de nodos en caso de caídas.
- Su modelo de datos dinámico y adaptable a las necesidades en cada punto del proyecto.

11.5.2 MTV Networks

MTV Networks, aparte del canal de TV y de las webs de cada país (mtv.com, mtv.es, etc) es propietaria de un conjunto de sitios tales como spike.tv, gametrailers.com, entre otros, los cuales tienen unas estadísticas de tráfico y accesos incluso mayores que la web de la propia MTV.

El equipo de la web de MTV evaluó migrar su base de datos relacional a un modelo NoSQL de cara a facilitar el escalado y un posible cambio de modelo de datos rápido de cara a cambiar ciertos aspectos del desarrollo de la web.

Eligieron MongoDB porque:

- El modelo de datos flexible les permitía almacenar información jerárquica de forma sencilla y sin necesidad de consultas pesadas sobre la base de datos.
- Las querys y el indexado de información es potente y completo, y la MTV necesitaba poder ofrecer estas dos características de cara al público.
- Facilidad de operaciones, en el sentido de la escalabilidad y la alta disponibilidad. Es relativamente sencillo levantar un nodo y hacerlo formar parte del un ReplicaSet ya existente.

11.6 Anexo IV: Diseño real de una base de datos en MongoDB

Para la realización de estas pruebas, se han añadido los repositorios oficiales de MongoDB para Ubuntu, y se ha instalado la última versión estable disponible desde estos repositorios, la 2.6.1. El sistema operativo utilizado ha sido Ubuntu 14.04, la última versión, que además es LTS, del conocido sistema operativo.

11.6.1 Instalación de MongoDB

Primero se han añadido las claves GPG públicas de los repositorios de MongoDB, que nos aseguran la consistencia y autenticidad de los paquetes que se instalen desde estos repositorios.

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv 7F0CEB10
```

Se añaden los nuevos orígenes a la lista de repositorios del sistema, creando un nuevo fichero para MongoDB.

```
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-
upstart dist 10gen' | sudo tee
/etc/apt/sources.list.d/mongodb.list
```

Por último, se actualizan los paquetes y se instala la última versión disponible en estos repositorios.

```
sudo apt-get update
sudo apt-get install mongodb-org
```

Con la consecución de estos pasos ya se tendrá la base de datos instalada y lista para empezar a utilizarse.

11.6.2 Primeros pasos

Tal y como se especificó en la sección del documento que describe de forma teórica a MongoDB, los primeros pasos para comenzar a escribir datos en este sistema de almacenamiento será el de elegir o crear una nueva database dentro propia para el proyecto que se está tratando.

Así, el nombre de la base de datos que se ha decidido crear para nuestro corpus es el de "corpustass". Se selecionará la nueva base de datos mediante el comando.

```
> use corpustass
```

De esta manera, en caso de que la base de datos ya estuviera creada, el comando especificado más arriba nos ubicaría dentro de ella, de forma que se pueda comenzar a operar directamente sobre ella. En caso de que no estuviera creada, el comando `use` la crearía, y la mantendrá al salir de ella únicamente en caso de que se haya escrito información. En otras palabras, en caso de que no se escriba ningún dato, MongoDB no

registrará esta base de datos como una base de datos con información, y por tanto la eliminará de su sistema.

Una vez seleccionada la base de datos que se utilizará para almacenar los tweets del corpus vienen las colecciones que se crearán para alojar los distintos tipos de información que nos proporciona el corpus. En este caso, se han creado las colecciones `users` y `tweets`, que alojarán, como su propio nombre indica, usuarios y tweets respectivamente.

11.6.3 Comandos de inserción y búsqueda

MongoDB muchas de sus funcionalidades a través de comandos que pueden ser ejecutados desde terminal, y los que insertan y buscan datos están entre ellos.

El comando `insert()` sirve para añadir un objeto a la colección que se especifique.

```
> db.users.insert({name: <nombre>, tweets: <num_tweets>, …})
```

El comando `find()`, invocado sobre una colección, permitirá obtener un listado de todos los objetos que estén contenidos en una colección. Se obtendrá un listado de todos los usuarios en el sistema de la siguiente manera:

```
> db.users.find()
```

El comando `pretty()`, frecuentemente se suele utilizar tras la llamada a `find()`, puesto que `find()` devuelve la información de forma secuencial, y `pretty()` presenta los datos de forma indexada y organizada, de manera que mejora la lectura y posterior interpretación de la información.

11.6.4 Diseño literal de la base de datos.

A continuación se muestra un subconjunto de datos para las colecciones de usuarios y tweets correspondientes a los ficheros `general-users.xml` y `info-general-tweets.xml` respectivamente.

11.6.4.1 Colección users

```
> db.users.find().pretty()
{
	"_id" : ObjectId("5378f40d4a54470291363c78"),
	"screenname" : "EvaORegan",
	"lang" : "en",
	"type" : "periodista",
	"tweets" : 420
}
{
	"_id" : ObjectId("5378f6544a54470291363c7c"),
	"screenname" : "jesusmarana",
	"lang" : "es",
	"type" : "periodista",
	"tweets" : 838
```

```
}
```

11.6.4.2 Colección tweets

```
> db.tweets.find().pretty()
{
     "_id" : ObjectId("5378f88b4a54470291363c7d"),
     "user" : "jesusmarana",
     "content" : "Portada 'Público', viernes. Fabra al
banquillo por 'orden' del Supremo; Wikileaks 'retrata' a
160 empresas espías. http://t.co/YtpRU0fd",
     "date" : "2011-12-02T00:03:32",
     "lang" : "es"
}
{
     "_id" : ObjectId("5378f9044a54470291363c7e"),
     "user" : "EvaORegan",
     "content" : "Grande! RT @veronicacalderon \"El
periodista es alguien que quiere contar la realidad, pero
no vive en ella\" via @galtares",
     "date" : "2011-12-02T00:06:32",
     "lang" : "es"
}
```

Este diseño de la base de datos ha sido realizado cogiendo la información de los tweets y llevada a MongoDB tal cual ha sido obtenida del corpus de tweets.

Como se puede observar, es un diseño un tanto deficiente, puesto que MongoDB permitiría realizar un mapeo relacionando usuarios (que escriben tweets) y tweets (escritos por usuarios).

A continuación se expone un diseño mucho más acorde con las posibilidades que ofrece MongoDB.

11.6.5 Diseño del corpus adaptado a MongoDB

11.6.5.1 Colección users

```
> db.users.find().pretty()
{
     "_id" : 268470229,
     "screenname" : "jesusmarana",
     "lang" : "es",
     "type" : "periodista",
     "tweets" : 838
}
{
     "_id" : 44903626,
     "screenname" : "EvaORegan",
     "lang" : "en",
     "type" : "periodista",
```

```
        "tweets" : 420
}
```

11.6.5.2 Colección tweets

```
> db.tweets.find().pretty()
{
        "_id" : 142379080808013820,
        "userId" : 44903626,
        "content" : "Grande! RT @veronicacalderon \"El
periodista es alguien que quiere contar la realidad, pero
no vive en ella\" via @galtares",
        "date" : ISODate("2011-12-02T00:06:32Z"),
        "lang" : "es"
}
{
        "_id" : 142378325086715900,
        "userId" : 268470229,
        "content" : "Portada 'Público', viernes. Fabra al
banquillo por 'orden' del Supremo; Wikileaks 'retrata' a
160 empresas espías. http://t.co/YtpRU0fd",
        "date" : ISODate("2011-12-02T00:03:32Z"),
        "lang" : "es"
}
```

Como se puede apreciar, ha habido ciertas modificaciones con respecto al modelado anterior de la información.

Los IDs tanto de los usuarios como de los tweets ahora ya no son ObjectId's (el tipo de objeto por defecto para mantener identificadores de los elementos de la base de datos) y han pasado a ser identificadores numéricos. Esto ha sido una decisión de diseño motivada porque, a pesar de renunciar a las ventajas que aporta el tipo de objetos ObjectId, el resultado es un corpus original mucho más fielmente representado, con los IDs de cada objeto.

De este modo, el campo `tweetid` de cada tweet, y el campo `id` de cada usuario corresponden con los id's de los nuevos objetos en la base de datos. Además, se ha añadido un nuevo campo en cada objeto de tipo tweet con nombre `userId`, que referencia al autor del tweet en cada caso. Este nuevo campo es un valor numérico que corresponde con el ID del usuario de la colección `users`. De esta manera, si se desea conocer más información acerca del autor del tweet tan solo habría que realizar una búsqueda de usuario en la colección `users` y obtener el objeto correspondiente.

11.6.6 Todos los ficheros de tweets bajo la misma colección

Otra de las características que tiene MongoDB nos sirve para poder almacenar todos los tweets, tanto de test como de entrenamiento bajo la colección tweets. Esto es así porque MongoDB carece de esquema, y se pueden crear y rellenar nuevos campos que no tienen el resto de objetos, o no rellenar los campos que sí son comunes a todos los elementos de la colección.

Estas son las principales características así como las diferencias de los ficheros que almacenan tweets de la colección del corpus.

- **General-tweets-test**: Fichero con información sobre los tweets.
- **General-tweets-train**:. Igual que el fichero de test descrito arriba, pero incorpora el sentimiento con el que se ha etiquetado manualmente al tweet.
- **Info-general-tweets-test**: Igual que el fichero de test, añadiendo el contenido del tweet original, campo que no está incluido en el fichero mencionado.
- **Info-general-tweets-train**: Igual que el fichero de train pero añadiendo el contenido del tweet, que se omite en los ficheros que comienzan por "general-".

Viendo la estructura que mantienen los ficheros se puede apreciar que, finalmente, las estructuras son muy parecidas entre sí. Únicamente varían algunos campos en cada caso. Unos contienen información sobre el sentimiento, otros sobre el contenido, otros ninguna de las dos cosas… De echo, algunos de ellos se solaparían y serían el mismo tweet, con el consiguiente ahorro de espacio en disco al no tener que almacenar dos veces la misma información.

MongoDB nos permite modelar todos los ficheros bajo la misma colección incluyendo para cada caso únicamente los campos conocidos. Otra cuestión sería si, por ejemplo, a efectos prácticos, nos interesara más tener almacenados en colecciones distintas los tweets de test y los de entrenamiento, para poder recuperar los conjuntos de forma independiente. En ese caso, habría que crear dos colecciones: una de ellas tendría todos los tweets con las referencias al sentimiento y la otra no. Así se tendrían los conjuntos de test y entrenamiento separados en dos colecciones claramente diferenciadas.

12 Bases de datos orientadas a grafos

13 Neo4j

13.1 Introducción

Neo4j es una base de datos basada en grafos. Neo4j está escrita en Java y ha sido y es desarrollada por la empresa Sueca Neo Technology. Neo4j es probablemente la base de datos basada en grafos más conocida y más utilizada de todas.

Mas adelante se abordará el tema con más detenimiento, pero vale la pena mencionar que la principal característica de las bases de datos orientadas a grafos es que están especialmente ideadas para representar relaciones entre datos y datos conectados.

Las bases de datos orientadas a grafo son, probablemente, las que más difieren del resto de bases de datos NoSQL, primero porque abordan su diseño (el diseño de la propia base de datos) de una forma completamente distinta, y segundo porque de entre todas las bases de datos NoSQL son las que, probablemente, más se parezca en su manera forma de realizas las operaciones a las bases de datos relacionales.

13.2 Grafos

Antes de continuar, es necesario hacer un breve repaso sobre lo que son los gráfos y en lo que consiste la teoría de grafos (Robinson, Webber and Eifrem 2013).

Un **grafo** no es más que un conjunto de **nodos** que mantienen una serie de **relaciones** entre sí. Las relaciones tienen nombre y sentido, y siempre tienen un nodo de inicio y uno de fin. Tanto los nodos como las relaciones poseen **propiedades**. Los **recorridos** son consultas que se realizan sobre una parte o la totalidad del grafo.

Las siguientes expresiones resumen qué es cada uno de estos elementos y como puede utilizarse conjuntamente con el resto de elementos (Neo Technology 2014):

- Un **grafo** almacena datos en **nodos**
- Un **grafo** almacena datos en **relaciones**
- Los **nodos** se organizan en **relaciones**
- Los nodos y relaciones tienen propiedades
- Un **recorrido** navega un **grafo**
- Un **recorrido** identifica **rutas**
- Las **rutas** ordenan **nodos**
- Los **índices** son **recorridos** especiales para encontrar **nodos** o **relaciones**

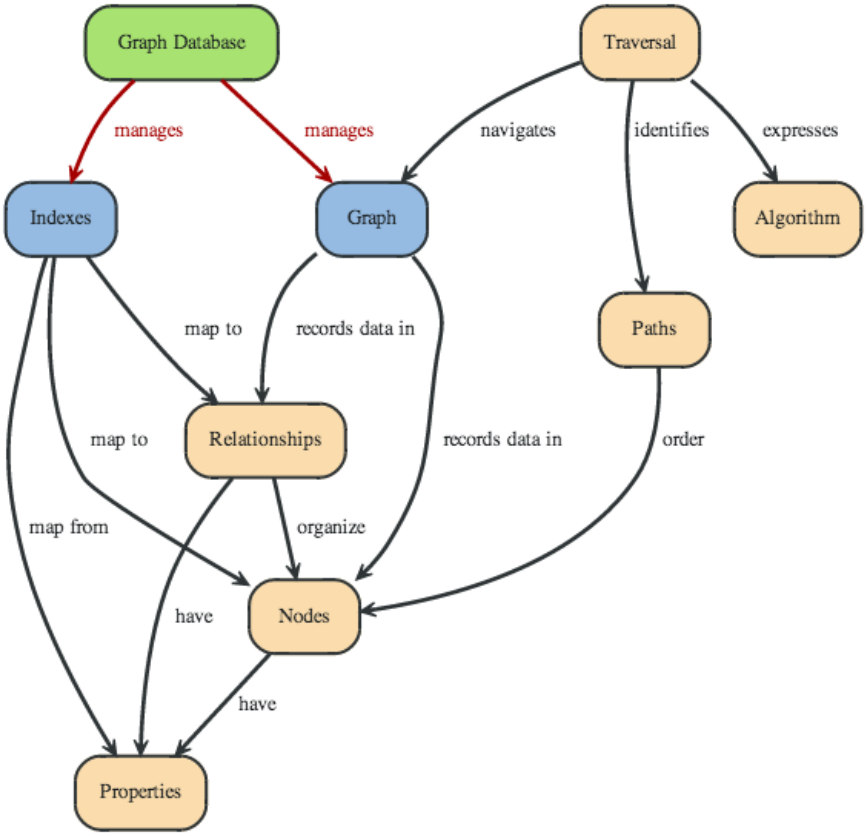

Ilustración 34. Interrelaciones de elementos de un grafo

Realmente todos los elementos comentados aquí forman parte de la fisionomía de los grafos excepto los índices. Los índices son propios de la base de datos orientada a grafos. Es la propia base de datos la que los genera y mantiene, y los utiliza para acelerar según que tipo de consultas.

Los nodos, también conocidos como vértices, representan entidades, habitualmente personas o cosas, mientras que las relaciones, también conocidas como aristas o arcos, son elementos que generan una interconexión entre los nodos.

En teoría de grafos, un grafo se representa como G = (V, E), donde V es el conjunto de vértices y E el conjunto de aristas (edges).

13.2.1 Aplicaciones reales

Muchos estudios sostienen que la representación de la información como grafo es algo inherente a la propia naturaleza de los datos. Representar los problemas del mundo real en forma de grafo es un pequeño paso más allá de lo que hace el resto de bases de datos y, probablemente, el enfoque que muchos problemas necesitan para ser abordados de una manera más sencilla (Buerli 2012, Chang, et al. 2006).

Uno de los campos con más potencial en explotar este tipo de bases de datos es el de la **química** y la **biología**. La información sobre los compuestos químicos en su forma más básica (átomos y enlaces) se acopla perfectamente a la estructura y visión que aportan los grafos.

Para los casos en los que la información tiene muchas repeticiones de los mismos elementos, las operaciones con los grafos pueden consistir en reconocimiento de patrones para futuros análisis. Muchas de estas operaciones son agilizadas por motores basados en grafos que, de otro modo, con otro tipo de base de datos, podría llevar mucho más tiempo debido a la naturaleza recursiva de muchos casos.

Otro caso particularmente interesante y que cuenta con bastante literatura al respecto es el de las **redes sociales**.

En los últimos años no son pocos los trabajos de investigación que se han desarrollado en torno a las redes sociales y su posible tratamiento y extracción de información empleando grafos.

Los grafos aquí no solo tienen porque representar personas y las relaciones que mantienen entre sí, sino también enlaces a sitios web, elementos multimedia o mensajes.

Redes sociales con un volumen muy grande de información, que previsiblemente puedan generar grafos de tamaños enormes, pueden estar interesados en algoritmos de caminos mínimos entre dos puntos, o recorridos maximizando valores o pesos.

La propia **web** como tal es en esencia un grafo de información enlazada entre sí. Los links establecen la forma más básica de unir páginas o comunidades de información entre sí. El motor que Google utiliza para determinar la relevancia de los sitios web tiene su base en recolectar y analizar los enlaces que apuntan desde y hasta cada página web.

Pero la utilización de grafos no está limitado únicamente a compañías online o que generan grafos enormes, cualquier tipo de empresa puede beneficiarse de sus características. Una de las ventajas de emplear grafos es el hecho de que permite representar modelos de datos complejos, por ejemplo, para compañías que necesitan emplear jerarquías de productos, o bien que manejan datos de carácter financiero.

13.3 Características principales

Ya se han visto algunas características propias de este tipo de bases de datos, pero ahora se ahondará un poco más en profundidad en los beneficios que nos aportan los grafos aplicados al campo de las bases de datos.

Concretamente, se encontrará una mejora en un conjunto de casos de uso para los cuales aplicar una solución orientada a grafos nos mejora en varios órdenes de magnitud el rendimiento, o que nos proporciona una latencia mucho más baja comparándola con otras soluciones NoSQL (Neo Technology 2014).

13.3.1 Rendimiento

La principal ventaja a nivel de rendimiento que se puede obtener de las bases de datos orientadas a grafos es cuando se parte de la base de que los datos que se van a almacenar poseen relaciones entre sí o están conectados de alguna manera.

Además, en lo que respecta concretamente a los tiempos de consulta, las bases de datos relacionales habitualmente comienzan a arrojar tiempos muy elevados en el momento en el que se hace un uso intensivo de sentencias join. Esto es debido a que los tiempos

de respuesta se deterioran en gran medida a medida que la base de datos se va haciendo más y más grande.

Sin embargo, las gráficas de rendimiento en lo que a consultas se refiere tiene a permanecer constante en las bases de datos orientadas a grafos. Esto es así porque las consultas se localizan únicamente en la porción del árbol que aplica a la consulta. Por tanto, el tiempo de consulta será proporcional al tamaño del subárbol concreto sobre el que se esté realizando la query en lugar de ser proporcional al la base de datos entera.

13.3.2 Flexibilidad

Las bases de datos orientadas a grafos nos permiten modelar la base de datos de forma gradual. Es decir, es natural comenzar a diseñar una base de datos una vez que se ha comprendido bien el problema, pero también es natural que, a medida que va avanzando la propia experiencia en el dominio que se está tratando, crezca también nuestra comprensión sobre el espacio del problema.

La utilización de grafos permite alterar y modificar partes del modelo de datos de forma rápida y natural, a diferencia de otras bases de datos para las que realizar un cambio de estas características supone un esfuerzo considerable tanto en tiempo como en rendimiento.

13.3.3 Agilidad

De un tiempo a esta parte y cada vez más frecuentemente, las metodologías ágiles están comenzando a dirigir los desarrollos de software, tanto de las grandes como de las pequeñas empresas. Este tipo de metodologías exigen un compromiso, unas fechas y una serie de entregas que vienen siendo más comunes y más frecuentes que las formas tradicionales de trabajar. Y es necesario que la base de datos también forme parte de esa agilidad y de esa flexibilidad que se le demanda a los equipos y al resto de componentes software del proyecto.

La naturaleza sin esquema de las bases de datos orientadas a grafos hace que se perfilen como una opción mucho más viable que las bases de datos mucho más rígidas de esquema y con mecanismos más propios de sistemas relacionales.

13.3.4 Transacciones ACID

Neo4j tiene soporte completo en lo que a transacciones ACID se refiere. Esto convierte a Neo4j en una base de datos transaccional, y por este motivo se acerca mucho más que el resto de bases de datos NoSQL a las bases de datos relacionales. Aunque bien es cierto que no es la única base de datos NoSQL que soporta transacciones. Otras, como Redis, también poseen esta funcionalidad.

No se comentará mucho más lo que son las transacciones ACID puesto que ya ha sido ampliamente tratado con anterioridad. Pero si las características propias de Neo4j.

Todas las operaciones de base de datos, tanto accesos a los grafos como a los índices y a los esquemas se realizan como transacciones. Sin embargo, los datos obtenidos mediante recorridos no están protegidos de modificaciones por parte de otras transacciones.

Los bloqueos son obtenidos a nivel de nodo y de relación, y se realizan de forma automáticamente por el sistema. Neo4j también permite realizar manualmente los bloqueos de modo que se garantice un mayor nivel de aislamiento en las operaciones.

Por último, las detecciones de interbloqueos son aplicadas a nivel de core del sistema, con las ventajas que esto conlleva a la hora de detectar y reaccionar antes tales sucesos.

13.4 Arquitectura

A continuación se expondrá el modo de funcionamiento interno de Neo4j referido a aspectos como son la replicación, las copias de seguridad o el lenguaje propio que tienen las aplicaciones para acceder a consultar y modificar datos directamente en la base de datos (Neo Technology 2014).

13.4.1 Alta disponibilidad

Neo4j ha sido diseñado para correr tanto en modo single-node como en un modo distribuido en el que siempre existe un maestro como nodo principal del clúster, pudiendo existir también cero, uno o más esclavos. La lógica que se sigue en este apartado de Neo4j es muy similar a la que siguen otras bases de datos, como por ejemplo, MongoDB.

La estructura que sigue Neo4j para proporcionar al sistema de alta disponibilidad es similar a la que se suele emplear de maestro-esclavo, con la salvedad de que todos los nodos del sistema pueden recibir operaciones de escritura, y no solo el maestro. Esto se verá en profundidad más adelante.

Cada nodo contiene la lógica necesaria para coordinarse y comunicarse con el resto de miembros del clúster. Cuando un nodo arranca, examinará sus ficheros de configuración. Si pertenece ya a un clúster, pasará a formar parte del mismo como esclavo. En caso de que pertenezca a un clúster que no exista, se creará y se colocará como maestro.

Cuando un esclavo se hace cargo de una escritura en el sistema, de forma síncrona deberá escribirla también en el maestro. Recordar que será este último el encargado de replicar la información por el resto de esclavos del sistema.

Tras haber recibido la orden de escritura, la operación se realizará síncronamente entre el esclavo y el maestro. Para ello, el esclavo debe estar completamente actualizado con el maestro, es decir, mantener exactamente las mismas bases de datos. Una vez que el maestro haya finalizado su escritura, el esclavo comenzará su transacción.

A la vez que se está ejecutando la operación de escritura anteriormente descrita, el maestro puede implementar una replicación optimista. La replicación optimista es un método por el cual, en el mismo momento en el que el maestro intenta realizar la escritura en su base de datos local, también intenta hacer llegar a los esclavos los nuevos datos. Esta replicación evidentemente puede fallar, pero en ese caso se dará por buena la transacción de escritura y se procederá a realizar una escritura retardada en los esclavos.

Cuando un nodo esclavo permanece indisponible por un periodo determinado de tiempo, el resto de nodos del sistema le marcan como nodo fallido. Cuando el nodo

vuelve a desempeñar su funcionamiento normal, deberá actualizarse por completo antes de poder volver a poder formar parte del clúster.

Si es el maestro el nodo que cae, debido a problemas en la red o debido a problemas de hardware, un nuevo maestro es elegido de entre todos los esclavos del sistema, de forma que su rol pasa de esclavo a maestro. Normalmente, entre que el maestro del sistema cae y uno nuevo toma el reemplazo transcurren unos segundos. Durante ese lapso de tiempo, el sistema no acepta escrituras.

13.4.2 Backup

Los backups en Neo4j son copias que se realizan de una base de datos mientras ésta está funcionando, vía red o de forma local.

Existen dos tipos de copias de seguridad, las *full* y las *incremental*.

- Las *full backup* son copias que se realizan de absolutamente toda la base de datos por completo en caliente, es decir, mientras la base de datos continúa funcionando como lo hace normalmente. Además, no solicita ningún bloqueo. Esto significa que, mientras se está realizando la copia de seguridad, otras operaciones y otras transacción pueden estar teniendo lugar a la vez. Para asegurarse de que la copia de seguridad es completamente consistente y contiene toda la información de la base de datos hasta el momento de la copia, todas las transacciones que den comienzo después de haber iniciado la operación de *full backup* serán después también ejecutadas en la copia.
- Los *incremental backup* no son copias de la base de datos directamente. En su lugar, lo que guardan estas copias de seguridad son los logs de transacciones que han tenido lugar desde el último backup, *full* o *incremental*. De esta manera, el tiempo y el espacio que ocupa este tipo de backups será menor, pero también hay que tener en cuenta de que, en caso de tener que ser restaurado, es necesario partir del *incremental* o el *full backup* del que se ha partido.

Para restaurar un backup, lo único que es necesario es reemplazar la carpeta donde Neo4j aloja su base de datos con el contenido del backup. Hay que tener en cuenta que los backups que implementa Neo4j son copias de la base de datos completamente funcionales.

13.5 Modelo de consultas

Neo4j tiene asociado un potente lenguaje que permite realizar querys sobre los datos almacenados en la base de datos. Este lenguaje se denomina Cypher (Neo Technology 2014).

Cypher es un lenguaje de consulta declarativo y orientado a grafos, que realizar consultas y actualizaciones sobre la estructura del grafo. Cypher es un lenguaje sencillo pero potente, de forma que el programador puede centrarse en el dominio más que en tratar de acceder por su cuenta mediante cualquier otro tipo de lenguajes a la base de datos.

Como lenguaje declarativo, Cypher se centra en intentar expresar el *qué* se quiere obtener del grafo en lugar del *cómo* obtenerlo. Este enfoque hace que realizar optimizaciones en las consultas de la base de datos sea simplemente un detalle de

implementación, y le abstrae de todo el componente físico propio del sistema gestor de la base de datos (como por ejemplo, nuevo índices).

Cypher está inspirado en muchos lenguajes de programación, incluso en lenguajes propios de bases de datos. Sin ir más lejos, Cypher toma prestado de SQL la mayoría de su sintaxis, así como muchas de las cláusulas como *WHERE* o *ORDER BY*. También toma prestados ciertos aspectos relacionados con los patrones y las expresiones regulares de SPARQL, y otras tantas funciones de lenguajes como Python o Haskell.

Otro lenguaje que también soporta Neo4j es Gremlin (TinkerPop 2014). Gremlin es un lenguaje de consulta par bases de datos orientadas a grafos, similar a Cypher.

13.6 Ejemplos de uso real del SGBD

13.6.1 eBay

Hace menos de un año eBay compró una empresa llamada Shutl, especializada en realizar envíos. Un servicio especialmente popular que tienes esta pequeña empresa es la modalidad de envío en el mismo día, que te permite enviar una mercancía de un lado a otro de Reino Unido en menos de 24h. El record, que puede visualizarse en su web[56], es de 13' 57''.

EBay Now es un servicio que eBay ofrece, y que se dedica a enviar productos entre tiendas locales y entre tiendas locales y clientes, cumpliendo el servicio en menos de dos horas, o en un intervalo de tiempo que el cliente o la tienda receptora especifique, pagando un coste por el servicio de envío.

El equipo que desarrolla y mantiene el servicio sabía que mediante una base de datos basada en grafos se podría simplificar mucho el modelado del dominio. El sistema que tenían por aquel entonces, basado en MySQL, estaba comenzando a hacer que el código de base de datos fuera cada vez más lento y difícil de mantener. Era necesario un cambio para poder mantener la competitividad y la calidad del servicio.

Neo4j fue elegido por su flexibilidad de esquema, que hacía que los desarrollos fueran más rápidos, su mejor representación del modelo de datos que manejan, y al lenguaje de consulta Cypher, más sencillo y rápido.

A pesar de no tener una escalabilidad tan potente como otros sistemas NoSQL, el equipo implementó Neo4j afirma que sus tiempos de respuesta mejoraron, literalmente, del orden de miles de veces. Además, el código con el que realizabas las consultas a la base de datos se redujo, según el caso concreto, entre 10 y 100 veces (Neo4j 2014).

13.6.2 Infojobs

Infojobs es una web dedicada a mantener los perfiles profesionales de las personas y de las empresas, poniendo en contacto profesionales de todas las áreas y profesiones con empresas que demandan gente.

[56] http://shutl.com/uk/

Además, Infojobs te permite mantener un perfil actualizado de tu currículo, estudios, ponencias, conferencias, etc.

Marc Pou es el responsable del departamento de Nuevas Iniciativas de Infojobs, y fue quien comenzó a interesarse por las bases de datos de tipo grafo cuando comenzaron a experimentar dificultades para manejar la cantidad de información que manejan.

Infojobs tiene 8 millones de currículos de personas que buscan trabajo, cada uno de ellos contiene información profesional específica por cada una de estas personas.

El objetivo del portal "Plan de Carrera" (así lo han llamado en Infojobs) es el de ayudar a una persona que esté buscando trabajo. Esto lo consiguen buscando patrones en las trayectorias profesionales del resto de compañeros de profesión para identificar cual debe ser el próximo paso que debe dar.

Además, el Plan de Carrera proporciona información general sobre, por ejemplo, edades medias de cierto puesto de trabajo, o bien si en el estado profesional actual, un profesional sería capaz de que le contraten en cierto puesto.

Para desarrollar este portal, el equipo de Infojobs ha utilizado Neo4j y Memcached para almacenar los resultados, que se actualizan de forma mensual.

Según Marc Pou: "Sin Neo4j, los algoritmos para llevar a cabo este proyecto habrían requerido una excesiva cantidad de cálculos, complejidad y coste. Con Neo4j la se simplifica la complejidad, la velocidad de cálculo, la escalabilidad y la facilidad de programación".

Los datos de la utilización de Neo4j por parte de Infojobs son los siguientes (Neo4j 2013):

Tabla 19. Datos de la base de datos Neo4j de Infojobs

4 millones de candidatos 12 millones de puestos ocupados por los candidatos 6 millones de estudios 18 millones de habilidades	26.098.726 nodos 171.882.297 propiedades 137.429.999 relaciones 23 tipos de relaciones

14 Discusión sobre los sistemas vistos

A lo largo del presente trabajo se han expuesto algunos de los más importantes SGBDs correspondientes a los almacenes NoSQL más ampliamente usados. Se han analizado en profundidad sus características, de forma que ahora se pueda determinar en que casos es mejor utilizar una base de datos u otra en función de los datos y de las necesidades del sistema.

Para realizar esta comparación final, pongamos en perspectiva todos los sistemas que se han visto a lo largo del trabajo. Primero se encuadrarán según sus propiedades dentro del teorema CAP (3.2.1.1.1 El Teorema CAP)

Tabla 20. Comparativa de SGBDs según el Teorema CAP

Base de Datos	Características	No Cumple
DynamoDB	Disponibilidad y Tolerancia a Particiones	Consistencia
Redis	Consistencia y Tolerancia a Particiones	Disponibilidad
HBase	Consistencia y Tolerancia a Particiones	Disponibilidad
Cassandra	Disponibilidad y Tolerancia a Particiones	Consistencia
MongoDB	Consistencia y Tolerancia a Particiones	Disponibilidad
Neo4j	Consistencia y Disponibilidad	Tolerancia a Particiones

El siguiente es un cuadro comparativo entre los principales tipos de bases de datos NoSQL, atendiendo a las principales características funcionales como puede ser la escalabilidad o el rendimiento.

El objetivo de la comparativa no es otro que el de encontrar el tipo de base de datos adecuada a cada momento en función de las necesidades del proyecto (Strauch 2011).

Cabe destacar que, en los puntos en los que pone que una característica es variable, significa que, dependiendo de la configuración que se le dé, puede adquirir un valor alto o bajo. Por ejemplo, en el caso de la escalabilidad de las bases de datos documentales, el particionado horizontal hay que efectuarlo a mano, de ahí que esté descrito como "variable". Pero una vez que se ha realizado esa configuración inicial, se tiene un sistema que posee una escalabilidad mayor.

Tabla 21. Comparativa de Bases de Datos NoSQL

Modelo de Datos	Rendimiento	Escalabilidad	Flexibilidad	Complejidad	Funcionalidad
Clave-Valor	Alto	Alto	Alto	Bajo	Variable
Columnar	Alto	Alto	Moderado	Bajo	Mínimo
Documental	Alto	Variable	Alto	Bajo	Variable
Grafo	Variable	Variable	Alto	Alto	Teoría de grafos
Relacional	Variable	Variable	Bajo	Moderado	Algebra relacional

14.1 Uso de Clave – Valor

Por un lado, se puede apreciar que los contenedores que poseen **un modelo de datos clave - valor** son bases de datos que permiten atender un mayor número de peticiones por segundo gracias a sus condiciones favorables de escalabilidad y rendimiento. Esto es así a costa de sacrificar buena parte de la complejidad del sistema, permitiendo almacenar estructuras de datos más sencillas que el resto de sistemas aquí expuestos.

Por ello, **DynamoDB** y **Redis** son sistemas que vale la pena tener en cuenta, por ejemplo, cuando se quiere mantener una caché intermedia, o cuando hay datos que necesitan ser rápidamente accesibles. Habitualmente, en estas situaciones, se prescinde del hecho de ir a buscarlos a un sistema de almacenamiento más complejo, como por ejemplo un sistema relacional, manteniendo cacheada la información y sirviéndola desde un almacenamiento intermedio clave – valor.

DynamoDB es un servicio de base de datos basado en el modelo de pago por uso. Esto significa que no hay que encargarse de mantenimiento, rotado de logs, roturas de discos duros, compra de hardware, etc. Además, por ser propiedad de Amazon, DynamoDB se beneficia de la integración que tiene con el resto de servicios, por lo que si se tiene una plataforma montada en Amazon y se está considerando incluir un sistema de almacenamiento NoSQL, DynamoDB probablemente se perfile como buena opción.

Un problema del que adolece DynamoDB, y en general las bases de datos clave-valor, es que las querys están muy limitadas. Este tipo de almacenes no permite generar índices secundarios. Por tanto, las querys se pueden ver beneficiadas por el índice que se establece sobre la clave primaria, pero si se intenta realizar una consulta por cualquier otro campo, habrá que hacerlo sin índices.

Redis es un buen sistema para mantener clasificaciones, sistemas de votaciones, encuestas, popularidad de usuarios y personalidades, etc., puesto que Redis proporciona soporte nativo para gestionar operaciones que tienen que ver con mantener todas estas tablas y clasificaciones.

Otro tipo de aplicación donde Redis puede ser de utilidad es un sistema de alarmas y monitorización de equipos haciendo uso de la funcionalidad generador – consumidor. Así, existirían agentes en cada una de las máquina que se quiera monitorizar, publicando información relevante, y por otro lado, otros agentes se encargarían de extraer esa información y tomar decisiones.

14.2 Uso basado en Columnas

Las bases de datos de tipo columna, por su parte, ofrecen unas características de rendimiento similares a los sistemas clave valor. La flexibilidad, sin embargo, sale algo más perjudicada puesto que poseen un esquema algo más rígido que el SGBD comentado anteriormente. Esto será ampliado en la sección correspondiente, más adelante.

HBase, es un sistema que hace uso del sistema de ficheros de Hadoop, por lo que está preparado para almacenar una enorme cantidad de información en tablas, de manera que después pueda ser recuperada mediante consultas por rangos. Por ejemplo, para el almacenamiento masivo de logs. O para servir de base de un sistema de chat a nivel mundial, como puede ser el Facebook Messenger.

Cassandra, al igual que HBase, es una implementación del artículo de Google, el BigTable. Pero Cassandra fue implementado para dar solución a un problema concreto: el de ser una base de datos en la que se pueda escribir el 100% del tiempo, y se pueda distribuir por diversas localizaciones del mundo sin afectar al servicio.

Así, el uso más conocido que se le da a Cassandra (su concepción original era con este propósito) es el de la conocida barra de búsqueda que se puede ver en todas las cuentas de la red social.

La utilización de Cassandra, en general, viene determinada por unas necesidades muy concretas: un elevado volumen de escrituras, cuando se tienen unas previsiones altas de crecimiento, con una necesidades de proceso casi en tiempo real, probablemente en un servicio distribuido en múltiples CPDs, y cuando no nos importa que la consistencia de los datos sea eventual. Estas son las condiciones en las que Cassandra rinde al 100%. Si las condiciones de la plataforma varían en gran medida, será mejor explorar otro tipo de sistema de almacenamiento.

14.3 Uso basado en documentos

Los sistemas documentales, en general, también cuentan con unas características de rendimiento y escalabilidad bastante altas, a cambio de una complejidad y una funcionalidad menor que otras bases de datos.

La estructura documental de MongoDB la hace especialmente útil para almacenar información relacionada con redes sociales. Por ejemplo, una aplicación sobre series de TV que almacene diferente información sobre ellas. Cada serie habría que subdividirla

en temporadas, cada temporada en capítulos, cada capítulo en actores… Este es un esquema que cuadra muy bien con las posibilidades que ofrece MongoDB.

Además, el framework de agregación con el que cuenta Mongo permite a los programadores hacer consultas más complejas sobre los conjuntos de datos. Además, permite definir tantos índices y de tanta complejidad como se desee. Por tanto, MongoDB es un sistema a tener en cuenta cuando, aparte de almacenar gran cantidad de datos, se prevee que se vayan a realizar consultas complejas de forma más frecuente.

14.4 Uso basado en Grafos

Los SGBDs **orientados a grafo** también tienen esquema flexible, pero sus carencias vienen a la hora de escalar. Esto es así porque este tipo de base de datos cumplen la consistencia y la disponibilidad del Teorema de CAP, por lo que presenta dificultades enfrentándose a particiones de sus datos entre distintos nodos de una red. Por otro lado, permite un modelo de datos más rico y complejo, así como emplear la teoría de grafos para realizar operaciones en la base de datos.

Conceptualmente, la información vertida sobre una red social tiene más que ver con las relaciones que se establecen en los grafos las que se establecen en un conjunto de tablas.

Al estar orientadas a conectar los datos entre sí con una gran cantidad de relaciones, este tipo de bases de datos son especialmente útiles para establecer relaciones entre datos de redes sociales. Twitter utiliza su propia implementación de SGBD orientado a grafos para descubrir quien sigue a cierta persona, de manera que cuando esa persona escribe un tweet, los timelines de todos sus seguidores sea actualizado.

Neo4j también permite realizar consultas de tipo semántico. De forma similar al ejemplo que se acaba de describir, una base de datos como Neo4j nos permitiría realizar querys del tipo “¿a quien sigue cierta persona?” o “¿qué gustos musicales comunes tiene este grupo de personas?”, o “Hay más hombres o mujeres que siguen a cierta personalidad pública”, etc.

Por otro lado, según el teorema CAP esta base de datos no funcionará bien bajo particiones en la red, lo que significa que no es sencillo dividir el conjunto de datos en clústers con muchos nodos. Esto significa que se penaliza la escalabilidad, ante un elevado número de escrituras, frente a caídas parciales de la red, etc.

14.5 NoSQL vs Relacional

Por último, **los SGBDs relacionales** comparten características con las bases de datos orientadas a grafo. Se emplea el álgebra relacional para operar en base de datos y permite diseñar un modelo de datos más complejo que otros SGBDs aquí vistos. Por el contra, presenta dificultades en condiciones de escalado y flexibilidad del modelo de información.

Existen ciertas características comunes a la mayoría de bases de datos NoSQL que hace que se pueda realizar una comparativa relevante con los sistemas relacionales tradicionales.

Por ejemplo, por norma general, las consultas o querys está bastante limitado, especialmente en los sistemas que tienen la tolerancia a particiones como característica

propia del teorema CAP. Es el precio que pagan los almacenes NoSQL a cambio de ofrecer unas condiciones de rendimiento y escalabilidad más favorables.

Además, es cierto que cada base de datos SQL tiene sus particularidades, pero una persona que esté acostumbrada a utilizar sistemas basados en SQL podrá cambiar de una marca de SGBD relacional a otra con relativa facilidad.

Sin embargo, entre las distintas bases de datos NoSQL no hay un paradigma común, unas APIS o una manera conjunta de hacer las cosas, sino que cada tipo de base de datos, incluso cada base de datos tiene su propia implementación y su propio modo de funcionar. Por tanto, esto hace que abandonar un sistema para pasarse a otro haya una elevada barrera de dificultad entre sistemas. Esto es especialmente malo cuando se plantea hacer migraciones o movimientos de datos entre distintos tipos de contenedores.

15 Conclusiones

En los capítulos que componen este Proyecto Fin de Carrera se ha hecho un repaso por las bases de datos más utilizadas en la actualidad para almacenar información.

Así mismo, se ha estudiado el origen y la manera en la que el BigData está afectando a los servicios y a los sistemas software que se construyen hoy en día.

Las bases de datos NoSQL son una tecnología muy potente y variada que, si se sabe emplear de forma correcta, resulta ser una herramienta valiosísima que permitirá almacenar grandes cantidades de datos y extraer conocimiento de ellos de forma eficiente, tan necesario hoy en día debido al fenómeno BigData. No en vano, los puestos profesionales de informáticos que poseen conocimientos sobre alguna de estas tecnologías se están comenzando a demandar ampliamente.

Estas tecnologías son muy flexibles y versátiles, proponiendo soluciones a problemas que de otra manera han resultado ser difíciles de abordar. Pero esta tecnología no ha llegado para reemplazar a las bases de datos relacionales. Los sistemas relacionales poseen una serie de características y cumplen una serie de requisitos indispensables para muchos sectores de la informática, como la banca o el comercio electrónico.

Aún quedan muchos aspectos por explorar en esta tecnología, ya que aún no está madura (es relativamente joven, ya que se comenzó a hablar de ella con fines comerciales hace poco más de 5 años), faltando todavía experiencia en sistemas en producción de cara al público. Si se compara con los sistemas relacionales, éstos llevan solucionando problemas de almacenamiento y securización de datos desde hace más de 50 años.

Las bases de datos NoSQL han llegado para complementar la oferta de bases de datos de la que se dispone a la hora de abordar los problemas, y en ningún caso es recomendable alcanzar una posición extrema con respecto a las decisiones que se toman en este ámbito. Cada problema debe ser analizado en su conjunto, con sus necesidades particulares para cada caso, y una vez se tiene un esquema claro de esto, se debe buscar la solución de almacenamiento de entre todas las disponibles que mejor cubra las necesidades del problema.

El estudio realizado ha finalizado con buenos resultados. Los objetivos han ido cumpliéndose en cada apartado del proyecto, desde el análisis de las necesidades de almacenamiento que se tenían hace unos años, hasta el diseño de base de datos de dos de las bases de datos NoSQL más utilizadas en la actualidad, pasando por el análisis pormenorizado de varias de estas bases de datos NoSQL y una comparativa de ellas de acuerdo a su eficiencia.

El proyecto bien podría servir de guía introductoria a las bases de datos NoSQL más importantes del panorama actual, e incluso como referencia para una posible toma de decisiones sobre qué base de datos elegir para un sistema información que se quiera construir.

16 Trabajos Futuros

Como trabajo a corto plazo, sería interesante ampliar los distintos diseños de base de datos expuestos como ejemplos con diseños para el resto de bases de datos analizadas en el proyecto. De esta manera, se adquiriría una experiencia teórica y práctica para todas las bases de datos aquí expuestas. Así mismo, sería muy interesenta realizar una comparativa práctica derivada de estos ejemplos.

Otro tipo de aplicaciones que están surgiendo alrededor de estas bases de datos son las que proporcionan una capa de inserción y procesado de datos en tiempo real, como es el caso de **Storm**[57]. Esta aplicación hace uso de HBase para dotar a la base de datos de la capacidad de analizar y almacenar información en tiempo real que se puede obtener, por ejemplo, de un servicio como Twitter. Un interesante trabajo relacionado con este aspecto sería el de hacer un repaso por las tecnologías que implementan esta funcionalidad, y centrándose en el más importante o el más utilizado, y comprobar la funcionalidad real que es capaz de ofrecer, junto con un análisis de capacidad o de tiempos, calcular retardos, etc.

Un área de la informática, que va cobrando fuerza en estos últimos años es el del Cloud Computing. Los proveedores de servicios cloud saben que las bases de datos son un área sobre el que volcar sus esfuerzos, y prácticamente la totalidad de las empresas más importantes en este sector ofrecen servicios de base de datos totalmente gestionados por ellos. Así, un usuario no tiene que preocuparse de la infraestructura física, atendiendo únicamente a las necesidades de escalado de la misma.

Pero en realidad, ¿como se construye el sistema que permite ofrecer un servicio de manera que el usuario no tenga que preocuparse por la infraestructura física? Tendría que interactuar con una interfaz o con una capa software intermedia, que le permitiera configurar el sistema de alguna manera. Sería interesante analizar los requisitos y las funcionalidades que necesita un sistema cloud para ofrecer servicios de esta manera. Un estudio con algún sistema de base de datos relacional o NoSQL puede ser una buena opción, aunque esto es aplicable a otro tipo de servicios, como sistemas de ficheros distribuidos, escritorios remotos, envío masivo de correo electrónico, etc.

[57] https://storm.incubator.apache.org/

17 Glosario

BBDD	*Base de Datos.*
CPD	*Centro de Proceso de Datos.*
Clúster	*Agrupación de máquinas o nodos que funcionan de forma transparente para el cliente, dando la impresión de ser un único sistema.*
Framework	*Entorno de desarrollo que busca estandarizar los criterios y enfoques de programación.*
ORM	*Object–Relational Mapping. Mapeo Objeto-Relacional*
Persistencia AOF	*Persistencia Append Only File. Tipo de persistencia que se basa en escribir cada operación en un log binario.*
Persistencia RDB	*Persistencia Redis DataBase. Tipo de persistencia que se basa en hacer volcados periódicos de la base de datos completa.*
Query	*Consulta.*
SPoF	*Single Point of Failure. Punto Único de Fallo.*
Peer-to-Peer	*Red de pares. Red de máquinas en la que dos equipos se conectan directamente y comparten información sin necesidad de clientes ni servidores.*
SGBD	*Sistema Gestor de Bases de Datos.*
SSTable	*String Sorted Table.*

18 Bibliografía

10gen. *The MongoDB 2.6b Manual.* 2014. http://docs.mongodb.org/manual/ (accessed 2014).

Amazon. *Amazon DynamoDB Documentation.* 2012. https://aws.amazon.com/es/documentation/dynamodb/ (accessed 2014).

Amazon Web Services. *Amazon DynamoDB API Reference.* 08 de 2012. http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/Welcome.html (accessed 2014).

—. *Infraestructura Global.* 2013. https://aws.amazon.com/es/about-aws/globalinfrastructure/?nc1=h_l2_cc#reglink-na (accessed Mayo de 2014).

Apache Software Foundation. *HDFS Design.* Edited by Apache Software Foundation. 2014. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (accessed 2014).

—. *The Apache HBase™ Reference Guide.* 2014. https://hbase.apache.org/book.html (accessed Abril de 2014).

Aquilera, Marcos, Wojciech Golab, and Mehul Shah. *A Practical Scalable Distributed B-Tree.* 2008. http://www.vldb.org/pvldb/1/1453922.pdf (accessed Junio de 2014).

Basho Technologies. *Dynamo and Riak comparison.* 2013. http://docs.basho.com/riak/1.3.2/references/dynamo/ (accessed Junio de 2014).

BBVA. *Big Data: ¿En que punto estamos?* 2013. https://www.centrodeinnovacionbbva.com/magazines/innovation-edge/publications/21-big-data/posts/153-big-data-en-que-punto-estamos (accessed Junio de 2014).

Bewer, Eric. *CAP Twelve Years Later: How the "rules" have changed.* 2012. http://www.realtechsupport.org/UB/NP/Numeracy_CAP%2B12Years_2012.pdf (accessed Abril de 2014).

Buerli, Mike. *The Current State of Graph Databases.* 2012. http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/08%20Graph_Databases_Survey.pdf (accessed Mayo de 2014).

Chang, Fay, et al. *Bigtable: A Distributed Storage System for Structured Data.* 2006. http://research.google.com/archive/bigtable-osdi06.pdf (accessed Abril de 2014).

Chodorow, Kristina. *MongoDB: the Definitive Guide.* CA: O'Reilly, 2013.

Codd, Edgard F. *The Relational Model for Database Management: Version 2 .* Addison Wesley Publishing Company, 2000.

Codd, Edgard F, S. B. Codd, and C. T. Salley. *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate.* Codd & Associates, 1993.

Codd, Edgard Frank. *A Relational Model of Data for Large Shared Data Banks.* 1970. http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (accessed Marzo de 2014).

Dans, Enrique. *BigData, una pequeña introducción.* 19 de Octubre de 2011. http://www.enriquedans.com/2011/10/big-data-una-pequena-introduccion.html (accessed 02 de Marzo de 2014).

Dean, Jeffrey, and Sanjay Ghemawat. *Clusters, MapReduce: Simplified Data Processing on Large.* 2004. http://research.google.com/archive/gfs-sosp2003.pdf (accessed 2014).

DeCandia, Giuseppe, et al. *Dynamo: Amazon's Highly Available Key-value Store.* 2007. http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf (accessed 2014).

Dimiduk, Nick, and Amandeep Khurana. *Hbase in Action.* NY: Manning, 2013.

DynamoDB. *DynamoDB Testimonios.* 2014. https://aws.amazon.com/es/dynamodb/testimonials/ (accessed 2014).

Edlich, Stefan. *NoSQL Database.* 2014. http://nosql-database.org/ (accessed 2014).

George, Lars. *HBase, the Definitive Guide.* CA: O'Reilly, 2011.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. *The Google File System.* 2003. http://research.google.com/archive/gfs-sosp2003.pdf (accessed Abril de 2014).

Google. *Google Official Blog.* Junio de 2010. http://googleblog.blogspot.com.es/2010/06/our-new-search-index-caffeine.html (accessed Febrero de 2014).

Hewitt, Eben. *Cassandra: The Definitive Guide.* CA: O'Reilly, 2011.

HowieT. *blogs.msdn.com.* Abril de 2013. http://blogs.msdn.com/b/microsoftenterpriseinsight/archive/2013/04/15/the-big-bang-how-the-big-data-explosion-is-changing-the-world.aspx (accessed Febrero de 2014).

Internetria. *Internetria Blog.* Mayo de 2013. http://www.internetria.com/blog/2013/05/08/nosql/ (accessed Abril de 2014).

Karger, David, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.* 1997. http://thor.cs.ucsb.edu/~ravenben/papers/coreos/kll+97.pdf (accessed Mayo de 2014).

Kreps, Jay. *Proyect Voldemort | Design.* Edited by Proyject Voldemort Team. 2010. http://project-voldemort.com/design.php (accessed Abril de 2014).

Lai, Eric. *Computer World.* Abril de 2009. http://www.computerworld.com/s/article/9131526/Researchers_Databases_still_beat_Google_s_MapReduce (accessed Febrero de 2014).

Lakshman, Avinash, and Prashant Malik. *Cassandra - A Decentralized Structured Storage System.* 2009. http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf (accessed Junio de 2014).

Lamport, Leslie. *Time, Clocks and the Ordering of Events in a Distributed Systems.* 1978. http://www.stanford.edu/class/cs240/readings/lamport.pdf (accessed Mayo de 2014).

Macedo, Tiago, and Fred Oliveira. *Redis Cookbook.* CA: Manning, 2011.

Manyika, James, and Michael Chui. *Big Data: The next frontier for innovation, competition and productivity.* CA: McKinsey Global Institute, 2011.

Merv, Adrian. *www.teradatamgazine.com.* Enero de 2011. http://www.teradatamagazine.com/v11n01/Features/Big-Data/ (accessed 02 de Febrero de 2014).

Muthukkaruppan, Kannan. *The Underlying Technology of Messages.* 2010. https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919 (accessed Junio de 2014).

Neo Technology. *The Neo4j Manual.* 2014. http://docs.neo4j.org/chunked/milestone/ (accessed Junio de 2014).

Neo4j. *eBay Now Tackles eCommerce Delivery Service Routing with Neo4j.* 2014. http://info.neotechnology.com/rs/neotechnology/images/ebaynow_final.pdf.

—. *Europe's leading job site recommends your career path with Neo4j.* 2013. http://info.neotechnology.com/rs/neotechnology/images/Infojobs.pdf (accessed Junio de 2014).

OpenQM. *OpenQM, What is a Multivalue Database?* 2014. http://www.openqm.com/cgi/lbscgi.exe?T0=h&X=603yoxr4c3&t1=mv.overview (accessed Abril de 2014).

O'Really, Media Inc. *Big Data New: 2012 Edition.* California: O'Reilly Media, 2012.

Planet Cassandra. *Cassandra Used to Build Scalable and Highly Available Systems at Hulu, Streaming Content to over 5 Million Subscribers.* Enero de 2014. http://planetcassandra.org/blog/post/cassandra-used-to-build-scalable-and-highly-available-systems-at-hulu-streaming-content-to-over-5-million-subscribers/?page=blog (accessed Junio de 2014).

Redis Community. *Redis Documentation.* 2014. http://redis.io/documentation (accessed Mayo de 2014).

Ricky, Ho. *Ho Ricky Blog.* 2009. http://horicky.blogspot.com/2009/11/nosql-patterns.html (accessed Mayo de 2014).

Robinson, Ian, Jim Webber, and Emil Eifrem. *Graph Databases.* Suecia: O'Reilly, 2013.

SEPLN. *Taller de Análisis de Sentimientos en la SEPLN.* 2013. http://www.daedalus.es/TASS2013/about.php (accessed Junio de 2014).

Soubra, Diya. *Data Science Central.* Julio de 2012. http://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data (accessed Abril de 2014).

Steven L. Scott, Alexander W. Blocker, Fernando V. Bonassi, Hugh A. Chipman Edward I. George, Robert E. McCulloch. «Bayes and Big Data: The Consensus Monte Carlo Algorithm.» 31 de Octubre de 2013.

Stonebraker, Michael, Samuel Madden, Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. *The End of an Architectural Era (It's Time for a Complete Rewrite).* 2007. http://nms.csail.mit.edu/~stavros/pubs/hstore.pdf (accessed Mayo de 2014).

Strauch, Christof. *NoSQL Databases.* 2011. http://www.christof-strauch.de/nosqldbs.pdf (accessed Junio de 2014).

TinkerPop. *Gremlin Traversal Language Wiki.* 2014. https://github.com/tinkerpop/gremlin/wiki (accessed Mayo de 2014).

van Renesse, Robbert, Dan Dumitriu, Valient Gough, and Chris Thomas. *Efficient Reconciliation and Flow Control for Anti-Entropy Protocols.* 2008. http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf (accessed Mayo de 2014).

Velasquez, Washington. *Bases de datos orientadas a grafos y su enfoque en el mundo real.* 2013. https://www.academia.edu/5731075/Bases_de_datos_orientadas_a_grafos_y_su_enfoque_en_el_mundo_real (accessed Junio de 2014).

Welsh, Matt. *SEDA: An Architecture for Highly Concurrent Server Applications.* 2006. http://www.eecs.harvard.edu/~mdw/proj/seda/ (accessed Mayo de 2014).

19 Anexos

19.1 Anexo I: Instalación de Hadoop y HBase

Para la instalación de Hadoop se deberá tener primer configurado el siguiente entorno:

- Sistema Operativo Ubuntu 12.04+. El software funciona en otros sistemas operativos, pero la presente guía se ha realizado y probado en Ubuntu 12.04.
- JDK 6+ descargado en el equipo bajo la ruta `/opt/java` (recomendado JDK 7) (enlace a descarga)
- Ultima versión estable de Hadoop1 descargada, que en el momento de escribir esto es la 1.2.1 (enlace a descarga).

Una vez que se tiene esto, se descomprimen los archivos correspondientes a la versión de Hadoop en `/opt/hadoop`. Se tomará esta ruta como base de la instalación de Hadoop.

19.1.1 Configuración del Hadoop

Una vez en esa ruta, se editará el fichero `conf/hadoop-env.sh`, se buscará la línea donde se refleje la variable JAVA_HOME y se editará y añadirán las siguientes líneas:

```
JAVA_HOME=/opt/java
HADOOP_HOME=/opt/hadoop
PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin
```

A continuación, se editará el fichero `conf/core-site.xml`, y entre las líneas `<configuration>` y `</configuration>` se escribirán las siguientes líneas:

```
<configuration>
 <property>
   <name>fs.default.name</name>
   <value>hdfs://localhost:9000/</value>
 </property>
</configuration>
```

Después, se editará el fichero `conf/mapred-site.xml` y entre las líneas de configuración se añadirán las siguientes líneas:

```
<configuration>
     <property>
         <name>mapred.job.tracker</name>
         <value>localhost:9001</value>
     </property>
</configuration>
```

Y también el fichero conf/hdfs-site.xml con las líneas:

```
<configuration>
 <property>
   <name>dfs.replication</name>
   <value>1</value>
 </property>
</configuration>
```

Además, se asegurará que el contenido de los ficheros `conf/slaves` y `conf/master` sea únicamente una línea que ponga `localhost`.

Existen muchas otras opciones que se pueden especificar, y que permiten tener un control más pormenorizado del funcionamiento de la plataforma, pero esta configuración sirve para obtener un funcionamiento básico del sistema en un solo nodo.

19.1.2 Configuración de SSH

Para que Hadoop pueda comunicarse con todos los nodos que forman parte del clúster (en nuestro caso, será el nodo `localhost`) es necesario tener una comunicación por ssh sin contraseña. Para lograr esto en modo nodo único, se realiza la siguiente configuración desde terminal:

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Para comprobar que todo va bien, se lanza la línea:

```
ssh localhost
```

Se acepta el fingerprint o huella del nodo como huella digital confiable. La siguiente vez que se lance este último comando (que es, de hecho, el que lanza Hadoop) no debe pedirnos contraseña para establecer la conexión.

Inicio del servicio

Antes de lanzar los procesos de sistema propios de Hadoop se debe dar formato al sistema de ficheros. Esto se realiza lanzando por terminal el siguiente comando:

```
bin/hadoop namenode –format
```

De esta manera, se borrará toda la información que se haya almacenado anteriormente (suponiendo que no es la primera vez que se hace esto). Ahora sí, se lanzarán los procesos propios de Hadoop.

```
bin/start-all.sh
```

Para comprobar que todo se ha ejecutado correctamente se puede utilizar el comando jps, que mostrará los procesos de Hadoop corriendo (*NameNode*, *DataNode*, *JobTracker* y *TaskTracker*). También se puede acceder mediante un navegador a las interfaces web propias del *NameNode* y del *JobTraker* en http://localhost:50070/ y https://localhost:50030/ respectivamente.

19.1.3 Instalación de HBase

Antes de realizar una instalación de HBase, ha debido instalarse y configurarse primero Hadoop, tal y como se detalla en el anexo 19.1.1 Configuración del Hadoop.

Sobre este sistema base, se deberá descargar el código propio de HBase en su última versión estable, que en el momento de escribir estas líneas, es la 0.98.3. Esto puede encontrarse en el siguiente enlace. De forma similar a como se ha hecho con Java y con Hadoop, el software se descomprimirá en la ruta `/opt/hbase`.

Partiendo de la ruta base de HBase, `/opt/hbase`. El fichero propio de las configuraciones de hbase será `conf/hbase-site.xml`. De inicio, no será necesario que se especifique ninguna propiedad, pero en caso de necesitarlo, ese será el fichero en el que se deberá hacer.

Para arrancar HBase, desde la ruta base indicada anteriormente se escribirá por terminal:

```
bin/start-hbase.sh
```

Y ya estaría arrancado. Si se desea comprobar que está funcionando correctamente, se puede abrir una Shell de HBase. Esto se consigue mediante el comando:

```
bin/hbase shell
```

Con esto se accede a HBase para comenzar a ejecutar comandos.

19.2 Anexo II: Descripción de los tweets que conforman el corpus

19.2.1 Diseño de bases de datos para el almacenamiento de datos reales.

El siguiente apartado abordará el diseño de una base de datos con datos reales. Para realizar este ejercicio, se parte de un corpus de tweets perteneciente al TASS, el Taller de Análisis de Sentimientos del SEPLN (SEPLN 2013).

Estos talleres giran en torno al análisis de sentimientos centrado en el castellano. Son unas conferencias anuales en las que tratan temas referidos a la clasificación automática de textos, análisis de sentimientos, reputación online, entre otras cosas.

Como parte del programa, se propone un foro en el que los investigadores ofrecen sus puntos de vista acerca de los últimos avances relacionados con estos campos. Paralelamente se ha creado y liberado un corpus que consta de más de 68.000 tweets. Muchos de estos tweets han sido etiquetados con marcas de sentimiento (positivo, negativo, neutro…) o con tópicos con los que tiene relación el contenido del tweet (deporte, política, economía…).

El objetivo de este ejercicio será evaluar distintos tipos de SGBDs NoSQL de manera que se pueda determinar cual sería el más indicado para almacenar este tipo de datos y porqué.

Para ello, se creará un esquema de base de datos válido en dos bases de datos NoSQL: HBase y MongoDB. El diseño intentará hacerse lo más genérico posible, para poder almacenar la información de forma estructurada, de tal forma que sea fácilmente accesible y recuperable de cara a hacer uso de estos tweets para labores de investigación.

19.2.2 Descripción del Corpus

Dentro del conjunto de ficheros que se proporciona como parte del ejercicio, existen varios tipos, los cuales podrían ser clasificados en:

- **General-tweets**: 2 ficheros, uno de entrenamiento y otro de test. Ambos ficheros están en formato XML.
    - El fichero de **entrenamiento** contiene aproximadamente unos 7.000 tweets de personas famosas, tweets que han sido ya previamente evaluados y etiquetados con el sentimiento con el que fue escrito el tweet.
    - El fichero de **test** tiene aproximadamente 60.000 tweets, y son tweets exactamente iguales en estructura a los tweets de entrenamiento, solo que no tienen el sentimiento añadido, puesto que el objetivo de este conjunto es poder extraer este sentimiento partiendo del aprendizaje del conjunto anterior.
- **Info-general-tweets**: 2 ficheros, uno de entrenamiento y otro de test que contienen exactamente la misma información que los ficheros que comienzan por genera-tweet, pero a los que además se le añade el contenido de los tweets, cosa que no sucedía en el caso anterior.

- **General-users e info-general-users**. Estos ficheros contienen un resumen de los usuarios que aparecen en los ficheros vistos hasta ahora. Estos usuarios son identificados con datos como el idioma en el que escriben, su id o el tipo de usuario que es atendiendo a la actividad que desarrolla fuera de Twitter.

El fichero **info-general-users** contiene los mismos tweets que **general-users**, pero con información ampliada. Este fichero contiene la descripción que ha proporcionado el usuario, links a su foto de perfil, etc.

- **Politics-tweets-test e info-politics-tweets-test**. Este fichero recoge opiniones de muchos usuarios de Twitter sobre temas relacionados con la actualidad política.

El fichero que comienza por **info-** contiene la misma información que el otro fichero, pero además proporciona el contenido del tweet. Es la única diferencia entre ambos conjuntos de tweets.

Los sentimientos en los tweets se han clasificado manualmente uno por uno. Como se ha visto en el párrafo anterior, únicamente los conjuntos de entrenamiento han sido etiquetados con sentimiento, y estas etiquetas responden a un conjunto determinado de valores. Estos valores son:

- Para la etiqueta **tipo**: AGREEMENT, DISAGREEMENT.
- Para la etiqueta **valor**: P+, P, NONE, NEU, N, N+.

De todos estos ficheros, se ha decidido que los más relevantes para realizar diseños de las bases de datos en este Proyecto Fin de Carrera son los que recogen más información. Por este motivo, únicamente se recogen diseños para los ficheros cuyo nombre empiezan por "info-", ya que los que comienzan por "general-" son iguales a los otros pero sin el tag "content", que especifica el contenido del tweet. Puesto que ésta es la única diferencia se ha optado por descartar el diseño de estos ficheros.

Tabla 22. Ejemplo de formato de los Tweets del Corpus

Info-general-tweets-test	<tweet> <tweetid>142378325086715906</tweetid> <user>jesusmarana</user> <content><![CDATA[Portada 'Público', viernes. Fabra al banquillo por 'orden' del Supremo; Wikileaks 'retrata' a 160 empresas espías. http://t.co/YtpRU0fd]]></content> <date>2011-12-02T00:03:32</date> <lang>es</lang> </tweet>
Info-general-tweets-train	<tweet> <tweetid>142389495503925248</tweetid> <user>ccifuentes</user>

	<content><![CDATA[Salgo de #VeoTV , que día más largoooooo...]]></content> <date>2011-12-02T00:47:55</date> <lang>es</lang> <sentiments> <polarity><value>NONE</value><type>AGREEMENT</type></polarity> </sentiments> <topics> <topic>otros</topic> </topics> </tweet>
Info-general-users	<user> <screenname>jesusmarana</screenname> <id>268470229</id> <name>Jesús Maraña</name> <description><![CDATA[Periodista. He sido director de 'Público' (y algunas cosas más). Ahora, buscando el futuro...]]></description> <url>http://www.jesusmarana.com</url> <profile_image_url>http://a0.twimg.com/profile_images/1284036321/JESUS_normal.jpg</profile_image_url> <lang>es</lang> <location>Madrid</location> <timezone>Madrid</timezone> <followers_count>24863</followers_count> <friends_count>360</friends_count> <created_at>Fri Mar 18 21:02:44 +0000 2011</created_at> <type>periodista</type> <twits>838</twits> </user>
Info-politics-tweets-test	<tweet> <tweetid>137228516625367040</tweetid>

```
  <user>TonyKrdniosa</user>
  <content>"@marianorajoy: En España las cosas se pueden, se deben y se van a hacer infinitamente mejor que estos últimos 4 años" Eso son soluciones!!</content>
  <date>2011-10-17T19:00:02</date>
  <lang>es</lang>
  <sentiments>
   <polarity>
    <entity source="PP">@marianorajoy</entity>
   </polarity>
  </sentiments>
 </tweet>
```