Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención: Tecnologías de la Información

Estudio del SGBD Cassandra

Autor:
D. Daniel Escudero Romero

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención: Tecnologías de la Información

Estudio del SGBD Cassandra

Autor:
D. Daniel Escudero Romero

Tutora:
Dña. Carmen Hernández Díez

Dedicado a mis padres por el gran esfuerzo y apoyo que han hecho durante estos años por ayudarme a alcanzar los objetivos y metas que me he propuesto a lo largo de mi vida.

Resumen

Cassandra ha aparecido como una gran apuesta en el mundo de las bases de datos, tanto en el movimiento de Big data como en importantes empresas para ofrecer grandes cantidades de información al usuario o cliente, obteniendo excelentes resultados de rendimiento en cuestión de pocos segundos. Y es una gran apuesta por tres factores: ser de tipo NoSQL, por su característica de ser distribuida, y por estar basado en un modelo de almacenamiento *clave-valor*, tres factores que hacen que cada vez se implemente en más aplicaciones por los resultados que ello conlleva frente a otras bases de datos del mismo tipo. Sí que es cierto que es una base de datos relativamente joven frente a otras, aunque todas las de esta corriente de almacenamiento tienen un factor en común, la rapidez en servir al usuario datos, sean del tipo que sean (documentos, imágenes, etc.).

El objetivo de este proyecto es profundizar y desgranar Cassandra, sus orígenes, porqué surgió, comparativas, su arquitectura, modelo de almacenamiento, rendimiento, quien lo mantiene y qué soluciones o beneficios nos aporta cuando implementemos esta base de datos en nuestro trabajo. En definitiva, tener una visión global de este tipo de base de datos para comprender un poco mejor como afrontar la crecida de usuarios e información que hay en Internet, y comprender que una correcta gestión nos puede ayudar a manejar algo complejo creando algo más fácil.

También nos centraremos en escenarios, configuración y operaciones comunes que podemos encontrarnos cuando utilicemos este tipo de base de datos, todo ello enfocado de manera sencilla y concisa, contemplando diversas opciones de resolución para que de cara al lector sea posible su comprensión e implementación paralela. El nivel de detalle técnico será lo más elevado posible sin incrementar la dificultad de comprensión, acorde a la investigación que se está llevando a cabo y que sirva como futuro manual o guía en la docencia, consiguiendo así uno de los objetivos de este proyecto, que una persona con nociones en bases de datos relacionales, comprenda estas nuevas y sea capaz de transmitir esos conocimientos y de implementarlos. Otro objetivo es que sirva de continuación a otras investigaciones que partan de la base que aquí se recoge, llegando a nuevos enfoques o centrándose en partes específicas de Cassandra como el soporte, nuevos desarrollos del lenguaje de consultas o nuevas maneras de optimización de consultas que hagan, como hasta ahora, de manera sencilla operaciones complejas con un rendimiento alto.

Abstract

Cassandra has shown up as a great option in the Database world, both to Big Data industry and big companies to offer huge amounts of information to the final user or client, getting excellent few-seconds performance results. The fact that is a great option is based on three factors: being a NoSQL type database, being distributed, and being based on the key-value storage model. Those three factors have gotten Cassandra to a point where it is implemented in more and more applications because of the results it provides compared to other databases of the same type. It's true that it is relatively young compared to others, but all of this same family have a common factor, the speed at what they can serve data to the user (documents, images, etc.).

The objective of this project is to deepen and thresh Cassandra, it's origin, why it arose, comparatives, it's architecture, storage model, performance, who maintains it and what benefits and solutions it brings to the table when it comes to implementing this database on our work. To have a global overview of this kind of database to comprehend a little better how to face the growth of users and information in the Internet and understand that a correct management can help us handle something complex by creating something easier.

We will also focus on scenarios, configuration and common operations we can find when we use this kind of database, everything shown in an easy and brief way, looking into different resolution options for the user to help his understanding and further investigation. The level of technical detail will be the highest possible without increasing the difficulty of comprehension, at the same level as the investigation that is being carried on and that can serve as a future manual or teaching guide, achieving one the goals of this project, that a person with knowledge on relational databases understands this new kind, being able to pass on this knowledges and implementing them. Another goal is that it can serve as a continuation for different investigations that start from what has been written here, reaching new approaches or focusing in specific parts of Cassandra as support, new query language developments or new ways of query optimisation that can make complex operations with high performance.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y objetivos

Resumen: *A lo largo de este primer capítulo se busca presentar una imagen general del proyecto que se va a desarrollar, de manera que el lector llegue a comprender los principales objetivos que se persiguen con su desarrollo. Junto con ello, se presentará el método que se ha utilizado para llevar a cabo el proyecto, así como una pequeña planificación temporal y las herramientas necesarias para finalizarlo con éxito. Además, se presenta también una pequeña evaluación de posibles riesgos que pueden aparecer y a los que nos podemos enfrentar a lo largo del ciclo de vida de dicho proyecto, y el plan de acción frente a cada uno de ellos.*

1.1. Introducción y objetivos del T.F.G.

Con el paso de los años, la creciente demanda de Internet asociada con el aumento de ventas de dispositivos móviles y el auge de aplicaciones en la nube, surge la necesidad de almacenar gran cantidad de información y que además, esta información sea accesible rápidamente, sin "largas" esperas para consultar un dato en una red social o un buscador, o para encontrar un fichero guardado. Y es que en Internet está todo lo que queramos encontrar.

La cuestión aquí es a qué velocidad encontramos ese todo. Los usuarios exigen a los grandes proveedores de servicios un tiempo de respuesta razonable cuando ofrecen sus servicios, y más si son servicios por los que pagan. El papel que juegan las bases de datos en todo esto es importante, porque en algún lado hay que almacenar toda esa información, que crece cada día. Por ejemplo, en 1997, el tamaño de Internet era de unos 90 TB mientras que en la actualidad ha llegado a sobrepasar la barrera de los 5 millones de TB. Esto quiere decir que, en diecisiete años Internet es casi 60.000 veces más grande. A pesar de este aumento, hay que guardar todo porque alguien lo querrá en algún momento. Esta acumulación masiva de información se llama **Big data** [1], campo que se está explorando y en el que se pueden buscar múltiples soluciones a la cantidad de "problemas" que todo esto produce.

¿Cómo puedo buscar un fichero entre millones y que el tiempo de respuesta sea menor a 3 segundos para que el usuario no se moleste? ¿Cómo tengo que montar mi servidor y estructurar

mi base de datos para que esto sea viable? Y como estas, muchas preguntas para encontrar solución a este fenómeno.

Como ya se ha comentado, las bases de datos juegan un papel clave en este campo de investigación. Una buena manera de almacenar la información puede ser clave para que nuestro negocio o idea sea un auténtico éxito o un completo fracaso. Las bases de datos que llevamos utilizando en los últimos tiempos (bases de datos relacionales) tienen una gran limitación para solventar estos problemas. Por ello surge de esta evolución y de estas nuevas necesidades el objetivo del estudio que se presenta, bases de datos NoSQL, las cuales llegan como una **alternativa** a las bases de datos relacionales; y digo alternativa ya que, para algunas tareas, siguen siendo mucho más eficientes las relaciones que las NoSQL.

Esta necesidad de saber o conocer un poco mejor qué maneras hay de guardar información y que ésta sea accesible rápidamente y sin restricciones, es lo que me ha llevado a realizar este trabajo de investigación sobre un tipo de base de datos NoSQL, como es **Cassandra**. La curiosidad de como Google o Twitter muestran resultados en cuestión de segundos, o como Netflix es capaz de dar un soporte en streaming tan eficaz, es lo que me llevó a decantarme por esta potencial rama de investigación.

Objetivos

Todos los objetivos marcados buscan conseguirse desde un punto de vista introductorio a la tecnología, formativo y docente, por lo que las pruebas que se realizarán estarán enfocadas a conocer el comportamiento de las operaciones más comunes que se puedan realizar en las aulas o en pequeños proyectos.

Las dos líneas que se seguirán en este proyecto son:

- Pequeña introducción sobre NoSQL y **estudio teórico de Cassandra**.
- Construcción de una base de datos Cassandra e implementación de un caso práctico.

Además se deja la puerta abierta a posibles continuaciones de investigación sobre los distintos aspectos que se abran durante el desarrollo.

1.2. Planificación

En el siguiente punto, se detalla la planificación temporal que se ha seguido para la elaboración de este proyecto, así como los problemas que han surgido y han podido alterar dicha planificación, y los recursos que han sido necesarios para llevar a cabo cada fase.

En la tabla 1.1 que hay a continuación, la duración en días viene marcada tras la configuración de las horas de trabajo al día en el fichero Project que se adjunta con esta memoria, tal como se indica en el apéndice A.

Tarea	Descripción	Duración	Fecha Inicio	Fecha Fin
Plan de desarrollo del proyecto		**27 días**	18/11/14	20/12/14
Determinar el alcance del proyecto	Descripción de los propósitos y objetivos que se pretenden alcanzar con el desarrollo del proyecto, así como sus límites.	4 días	18/11/14	21/11/14
Planificación inicial		11,6 días	22/11/14	05/12/14
Identificación de las tareas	Identificación y descripción de cada una de las tareas necesarias para llevar a cabo el proyecto.	3 días	22/11/14	26/11/14
Determinar duración de las tareas	Determinación de las relaciones entre las distintas tareas.	4,2 días	26/11/14	29/11/14
Identificación de los recursos	Identificación de los recursos necesarios para llevar a cabo las tareas planteadas en el proyecto.	3 días	01/12/14	03/12/14
Afianzar recursos		1 día	05/12/14	05/12/14
Identificación y análisis de riesgos	Identificación y descripción de cada uno de los riesgos a los que nos podemos enfrentar y solución que damos ante cada uno.	2,8 días	06/12/14	10/12/14
Documentación de los objetivos	Elaboración de la memoria relativo al plan de desarrollo.	10,8 días	08/12/14	19/12/14
Revisión del plan de desarrollo	Revisión y aprobación del plan de desarrollo del proyecto marcado.	1 día	20/12/14	20/12/14
Búsqueda de información relacionada con NoSQL	Búsqueda de información relativa a las bases de datos NoSQL, su aparición, y relación con nuestro objeto de estudio.	**19,4 días**	24/11/14	17/12/14
Síntesis y redacción de la información	Redacción de la memoria utilizando la información obtenida hasta el momento. Se redacta según se va encontrando información fiable.	**9,8 días**	09/12/14	19/12/14
Instalación de herramientas necesarias para el plan de desarrollo		**1 día**	22/12/14	22/12/14
Cassandra		**28 días**	23/12/14	30/01/15

Sigue en la página siguiente.

Instalación del entorno	Instalación y preparación el entorno donde se va a trabajar.	2 días	23/12/14	26/12/14
Búsqueda de información y estudio de la tecnología	Búsqueda y estudio de Cassandra para conocer su arquitectura, posibilidades y limitaciones y hacer un planteamiento futuro del proyecto.	27 días	26/12/14	30/01/15
Pruebas básicas en el entorno	Realización de casos de prueba para comprobar que lo buscado es correcto y familiarizarse con el entorno	4,8 días	27/12/14	05/01/15
Síntesis y redacción de la información	Redacción de la memoria utilizando la información obtenida hasta el momento.	**28 días**	23/12/14	30/01/15
Caso práctico		**28,2 días**	02/02/15	20/03/15
Estudio de cómo se va a realizar el caso práctico	Planteamiento del objeto de estudio práctico y líneas a seguir.	2 días	02/02/15	03/02/15
Configuración del entorno acorde con el trabajo planificado	Últimos ajustes en la configuración del entorno acorde con el caso práctico.	1 día	04/02/15	04/02/15
Búsqueda de datos para trabajar en la base de datos	Información para introducir en la base de datos y poder hacer las pruebas pertinentes.	3,8 días	05/02/15	09/02/15
Inserción y realización de pruebas	Guardar los datos en la base de datos y realizar las pruebas necesarias para obtener las conclusiones necesarias.	22,4 días	23/02/15	20/03/15
Síntesis y redacción de la información	Redacción de la memoria utilizando la información obtenida hasta el momento.	**24,2 días**	23/02/15	23/03/15
Revisión del proyecto		**3 días**	25/03/15	27/03/15
Preparación de la presentación	Elaboración de una presentación para el día de exposición ante el tribunal.	**3 días**	30/03/15	01/04/15
Entrega del proyecto		**0 días**	28/04/15	28/04/15

Tabla 1.1: Planificación del Trabajo Fin de Grado

El salto temporal que hay en el mes de febrero se debe a un parón en el desarrollo del trabajo fin de grado por la realización de un curso externo con duración de jornada completa en un periodo de dos semanas. Este hecho ha provocado que se haya tenido que hacer un reajuste en la planificación para adaptarla a la situación. Una vez finalizado el curso, se ha continuado el desarrollo del proyecto con la planificación marcada.

1.3. Gestión de riesgos

Lo que se pretende en este apartado es reflejar aquellos casos de riesgo que se pueden sufrir durante la realización de este proyecto. La planificación que se ha presentado en la sección 1.2 es la que se intentará llevar a cabo para la correcta realización de este proyecto, pero es necesario un estudio de riesgos que, en caso de que ocurran, no arruinen dicha planificación.

- **Riesgo 1**:

FORMULARIO ADMINISTRACIÓN DEL RIESGO		
Identificador: **1**	Fecha: 26/11/2014	Categoría: Riesgos de Proyecto.
Título: **Bajas personales**.	Fase: En todos los hitos.	Consecuencia: Mayor carga de trabajo en un periodo de tiempo menor, ya que solo una persona es la encargada del desarrollo.
Impacto: *Medio*.	Probabilidad: *Baja*.	Exposición: *2* (*)
Valoración del riesgo		
Descripción del Riesgo: A lo largo del desarrollo del proyecto, por algún motivo justificable, no es posible realizar las tareas asignadas para ese periodo de tiempo.		
Contexto del Riesgo: Este riesgo se puede producir en cualquier fase del proyecto al sucederse una situación inesperada en la persona que lo esta desarrollando.		
Análisis del Riesgo: El impacto de este riesgo repercute aumentando la carga de trabajo en un menor tiempo, influyendo en la calidad del proyecto que se esta realizando.		
Planificación del Riesgo		
Estrategia: Reducción del riesgo, se tratará.		
Plan de acción del riesgo: Al no poder suplantar a la persona que no puede realizar las funciones, se aumentarían las horas/persona y día para llegar a los objetivos marcados.		

Tabla 1.2: *Riesgo 1* - Bajas Personales

- **Riesgo 2**:

FORMULARIO ADMINISTRACIÓN DEL RIESGO		
Identificador: **2**	Fecha: 26/11/2014	Categoría: Riesgos de Proceso.
Título: **Planificación del tiempo de cada tarea**.	Fase: En todos los hitos.	Consecuencia: Aumento de las horas para el desarrollo o incluso el fracaso del proyecto si no se reacciona a tiempo.
Impacto: *Alto*.	Probabilidad: *Media*.	Exposición: *5* (*)
Valoración del riesgo		
Descripción del Riesgo: Posibilidad de una mala planificación de cada una de las tareas del proyecto.		
Contexto del Riesgo: Este riesgo se puede producir en cualquier fase del proyecto debido a la planificación temporal.		
Análisis del Riesgo: El impacto de este riesgo puede repercutir en tareas posteriores a la ya afectada, exigiendo un cambio de planificación.		
Planificación del Riesgo		
Estrategia: Reducción del riesgo, se tratará.		
Plan de acción del riesgo: El plan de acción sería llevar un seguimiento de la planificación para poder llevar un seguimiento de la planificación para poder reaccionar a tiempo, redistribuyendo recursos y/o añadiendo o eliminando ciertas tareas.		

Tabla 1.3: *Riesgo 2* - Planificación de cada tarea

- **Riesgo 3**:

FORMULARIO ADMINISTRACIÓN DEL RIESGO		
Identificador: **3**	Fecha: 26/11/2014	Categoría: Riesgos de Producto.
Título: **Fallo de hardware**.	Fase: En todos los hitos.	Consecuencia: Retraso de la realización de tareas afectadas por dicho hardware.
Impacto: *Medio*.	Probabilidad: *Baja*.	Exposición: *3* (*)
Valoración del riesgo		
Descripción del Riesgo: Posibilidad de no disponibilidad del hardware en el momento necesitado.		
Contexto del Riesgo: Este riesgo se puede producir en cualquier fase debido a problemas no controlables en el funcionamiento del hardware.		
Análisis del Riesgo: El impacto de este riesgo puede producir retraso en la realización de tareas afectadas por dicho hardware.		
Planificación del Riesgo		
Estrategia: Protección del riesgo. Aceptación del riesgo.		

Plan de acción del riesgo: No se realizará ninguna acción ya que el coste asociado a cubrir el riesgo es excesivo a la probabilidad de que ocurra.

Tabla 1.4: *Riesgo 3* - Fallo de hardware

- **Riesgo 4**:

FORMULARIO ADMINISTRACIÓN DEL RIESGO		
Identificador: **4**	Fecha: 26/11/2014	Categoría: Riesgos de Producto.
Título: **Fallo de software**.	Fase: En todos los hitos.	Consecuencia: Retraso de la realización de tareas afectadas por dicho software.
Impacto: *Medio*.	Probabilidad: *Baja*.	Exposición: *3* (*)
Valoración del riesgo		
Descripción del Riesgo: Posibilidad de un funcionamiento incorrecto del software debido a un problema externo (malware, error de mantenimiento, etc.).		
Contexto del Riesgo: Este riesgo se puede producir en cualquier fase debido a fallos no controlables en del software.		
Análisis del Riesgo: El impacto de este riesgo puede producir retraso en la realización de tareas afectadas por dicho software.		
Planificación del Riesgo		
Estrategia: Reducción del riesgo.		
Plan de acción del riesgo: Se dispondrá de distintas aplicaciones para realizar la tarea implicada.		

Tabla 1.5: *Riesgo 4* - Fallo de software

- **Riesgo 5**:

FORMULARIO ADMINISTRACIÓN DEL RIESGO		
Identificador: **5**	Fecha: 26/11/2014	Categoría: Riesgos de Producto.
Título: **Poca destreza con las herramientas**.	Fase: En todos los hitos.	Consecuencia: Es necesario invertir mayor cantidad de dinero en el proyecto, o mayor trabajo por día.
Impacto: *Alto*.	Probabilidad: *Baja*.	Exposición: *4* (*)
Valoración del riesgo		
Descripción del Riesgo: Aumento del tiempo necesario para la finalización del proyecto por no estar familiarizado con la gestión de Cassandra y bases de datos NoSQL.		
Contexto del Riesgo: Este riesgo se puede producir en cualquier fase debido al desconocimiento inicial en cada fase.		

Análisis del Riesgo: El impacto de este riesgo puede producir retraso en la realización del proyecto.
Planificación del Riesgo
Estrategia: Evitación del riesgo.
Plan de acción del riesgo: Realización de cursos y lectura de tutoriales fuera de la planificación temporal del proyecto, o contar el aprendizaje en la planificación inicial.

Tabla 1.6: *Riesgo 5* - Poca destreza con las herramientas

La exposición de todos los riesgos, mencionados anteriormente, ha sido calculada utilizando el estándar ISO/IEC 7005:2008, mostrado en la figura 1.1.

Figura 1.1: Risk Management Standard ISO/IEC 27005:2008

		Likelihood of Incident Scenario				
		Very Low	Low	Medium	High	Very High
Business Impact	Very Low	0	1	2	3	4
	Low	1	2	3	4	5
	Medium	2	3	4	5	6
	High	3	4	5	6	7
	Very High	4	5	6	7	8

Fuente: management-of-risk.blogspot.com.es

1.4. Herramientas utilizadas

A continuación se listan aquellas herramientas que han sido necesarias para la realización del proyecto que se está presentando.

- Herramientas para la implementación y administración de Cassandra:
  - Equipo: ordenador sobre el que trabajar. Se ha trabajado sobre un ordenador personal con un procesador Intel Core i5 de 64 bits, con un sistema operativo Ubuntu 14.04, con una RAM de 8GB y disco duro de 500 GB.
  - Sistema operativo Ubuntu 14.04: sistema operativo instalado en el equipo de trabajo personal. Sobre este se encuentran instaladas las distintas herramientas para el desarrollo del proyecto.
  - VirtualBox: herramienta para la creación de las distintas máquinas virtuales necesarias para la realización del proyecto. En la sección 3.2 se da con detalle las características de las máquinas creadas.

- Herramientas para la realización de la memoria:
  - Geany 1.23: se ha utilizado este editor, en Linux, para la elaboración de la memoria en LaTeX. A través de los comandos `pdflatex` y `bibtex` se ha conseguido la construcción del documento y la bibliografía.
  - Microsoft Project 2013: herramienta utilizada para la planificación del proyecto, con la que podemos realizar un seguimiento detallado de la evolución del mismo, la carga de trabajo, y así analizar si existen problemas o riesgos con la planificación realizada.
- Otras herramientas:
  - Dropbox: herramienta para almacenar, en la nube, una copia de la documentación.

1.5. Resumen

Una vez definidas las líneas a seguir y establecida una planificación, lo que se pretende con el desarrollo de este proyecto es realizar una investigación exhaustiva sobre una nueva base de datos NoSQL, como es Cassandra, para así comprender y obtener otra visión sobre el almacenamiento, ajustándose a las necesidades que se demandan.

En este estudio se da una descripción detallada en el capítulo 3 de la arquitectura de Cassandra, el lenguaje que utiliza, la configuración del entorno y una serie de pruebas para ver cómo se puede aprovechar para implantarlo en futuros proyectos.

Uno de los objetivos es aplicar los conocimientos aquí plasmados en la docencia, para así poder dar una nueva visión a futuros alumnos de alternativas de almacenamiento eficientes para determinados aspectos, ya que la tecnología y la información avanzan a pasos agigantados.

El proyecto se encuentra a partir de aquí estructurado de la siguiente manera:

- **Capítulo 2 - Bases de datos NoSQL**: se da una pequeña introducción sobre qué son, cómo se estructuran, puntos fuertes y débiles y tipos, antes de entrar en la materia que nos ocupa.
- **Capítulo 3 - Cassandra**: capítulo maestro del proyecto donde se da una visión global y detallada de las funcionalidades de Cassandra, como arquitectura, lenguaje, configuración del entorno y operaciones.
- **Capítulo 4 - Caso práctico**: en el último bloque se hará un caso práctico tratando diversos aspectos explicados teóricamente, para comprobar las ventajas y/o inconvenientes de usar este tipo de base de datos.

Capítulo 2

Bases de datos NoSQL

Resumen: *A lo largo de este capítulo vamos a familiarizarnos con las bases de datos NoSQL. El objetivo será conocer y comprender el entorno con el que se va a trabajar en el siguiente capítulo, saber qué motivaciones han llevado a la creación de este tipo de bases de datos, conocer otros tipos de bases de datos NoSQL con el fin de proporcionar alternativas cuando se desarrolle una aplicación, e identificar diferencias entre ellas. Tras este primer análisis, nos centraremos también en nuestro objeto de estudio, Cassandra, desde una perspectiva general, detallando sus características, sus ventajas e inconvenientes o las principales funciones que este tipo de base de datos nos proporciona.*

2.1. Motivación de las bases de datos NoSQL

Las bases de datos NoSQL (*Not Only SQL*) se presentan como una amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico de gestión de bases de datos relacionales (RDBMS). Es importante conocer las causas que llevaron a la aparición de este tipo de bases de datos y qué nos pueden proporcionar con respecto a las SQL (*Structured Query Language*).

NoSQL se podría decir que llegó con la aparición de la Web 2.0, ya que hasta entonces, a la red sólo subían contenidos empresas que tenían un portal Web. Con la llegada de aplicaciones Web como YouTube, Facebook o Twitter, entre otras, la aparición del almacenamiento en la nube, o la contribución que han hecho Google o Amazon en el manejo de cantidades ingentes de información, cualquier usuario puede subir contenido, provocando así un crecimiento exponencial de los datos en Internet. Es en este momento cuando empezaron a aparecer los primeros problemas de la gestión de toda la información almacenada en bases de datos relacionales. El resultado de poner más máquinas para almacenar la información pronto se vió que no iba a ser suficiente ya que el coste era bastante elevado y el rendimiento se veía empobrecido.

En 1997, el tamaño de Internet era de unos 90 TB[1], mientras que en la actualidad ha llegado a sobrepasar la barrera de los 5 millones de TB. Esto quiere decir que, en diecisiete años Internet es casi 60.000 veces más grande. Esto ha provocado que las bases de datos relacionales muestren

[1] 1 TB (terabyte) = 1.000 GB (gigabytes) = 10^{12} bytes (en el Sistema Internacional, SI).

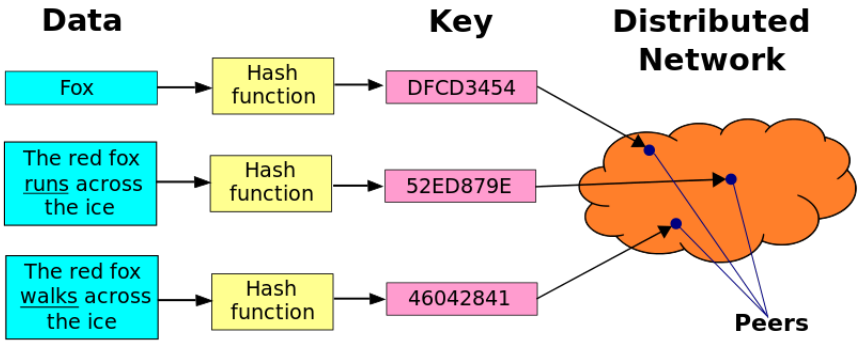
serias deficiencias, y de ahí que surja la necesidad de otro tipo de almacenamiento que sepa cubrir las necesidades actuales, dando prioridad a ciertos aspectos.

Hablar de NoSQL es hablar de una estructura que nos permite almacenar información. Teniendo en cuenta el rendimiento y el acceso a esos datos, se debe proporcionar escalabilidad debido a que las previsiones indican que va a seguir creciendo con el paso de los años y cada vez más rápido; y se debe poder almacenar de forma distribuida para que la información no se concentre en una misma máquina.

Esta forma de almacenar la información ofrece una serie de ventajas sobre los modelos relacionales. Entre las más significativas destacan:

- **Se ejecutan en máquinas con pocos recursos**: estos sistemas, al contrario que los basados en SQL, no requieren excesiva computación, por lo que el coste de montar estas máquinas se ve reducido.
- **Escalabilidad horizontal**: para mejorar el rendimiento, se implementa en varios nodos, con la única operación de indicar al sistema cuáles son los que están disponibles.
- **Manejo de gran cantidad de datos**: esto es gracias a una estructura distribuida que se consigue con mecanismos de tablas hash distribuidas (*Distributed Hash Tables*, *DHT* [2]), mecanismos que proporcionan un servicio de búsqueda similar al de las tablas hash, donde pares (clave, valor) son almacenados en el DHT y cualquier nodo participante puede recuperar de forma eficiente el valor asociado con una clave dada.

Figura 2.1: Distributed Hash Tables



Fuente: es.wikipedia.org/wiki/Tabla_de_hash_distribuida

- **No genera cuellos de botella**: evita el problema de las bases SQL, las cuales deben transcribir la sentencia para que ésta pueda ser ejecutada. Esto, ante muchas peticiones, puede ralentizar el sistema.

Hay que dejar clara una cosa, NoSQL no es un reemplazo de SQL, sino una alternativa. Por ejemplo, hay características necesarias que SQL dispone y las primeras no, como es el caso de las propiedades **ACID**, cualidades por las que las bases de datos relacionales destacan.

1. **A***tomicity* (Atomicidad): asegura que la operación se ha realizado con éxito. Todas las instrucciones deben ejecutarse correctamente; si no es así, ninguna deberá hacerlo.
2. **C***onsistency* (Consistencia): propiedad que se encarga de asegurar que sólo se va a comenzar aquello que se pueda finalizar.
3. **I***solation* (Aislamiento): la ejecución de una transacción no puede afectar a otras.
4. **D***urability* (Durabilidad): cuando se realiza una operación, nos asegura que persistirá, aún teniendo fallos en el sistema.

2.1.1. ACID vs. BASE

El enfoque **BASE**, según Brewer, abandona las propiedades **ACID** de consistencia y aislamiento en favor de "la disponibilidad, la degradación elegante y el rendimiento". El acrónimo BASE está compuesto por:

1. **B***asic* **A***vailability* (Disponibilidad básica).
2. **S***oft-state*.
3. **E***ventual consistency* (Consistencia eventual).

En la siguiente tabla (2.1), Brewer contrasta ambas propiedades, considerando los dos conceptos como un espectro en lugar de alternativas excluyentes entre si.

ACID	**BASE**
Consistencia fuerte	Consistencia débil
Aislamiento	Disponibilidad primero
Prioridad en "commit"	Mejor esfuerzo
Transacciones anidadas	Respuestas aproximadas
Disponibilidad	Agresivo (optimista)
Conservador (pesimista)	Más sencillo
Difícil evolución (p.e.: esquema)	Rápido
	Fácil evolución

Tabla 2.1: ACID vs. BASE [3, pág 32]

2.2. Características de los sistemas NoSQL

A pesar de existir numerosos tipos de bases de datos NoSQL, donde cada cual resuelve sus problemas en diferentes situaciones, todos ellos cuentan con unas caracterísiticias comunes, en las que vamos a ir profundizando poco a poco.

Como ya se ha dicho, presentan una **estructura distribuida**. Esto quiere decir que hablamos de un conjunto de bases que se encuentran lógicamente relacionadas, pero que se encuentran físicamente en distintos lugares, ya sea en un lugar pequeño, o separadas a gran distancia interconectadas por una red de comunicación. Nuestro objeto de estudio es un claro ejemplo de esta característica.

Otra de las características es la **escalabilidad**, propiedad que posee un sistema para reaccionar y afrontar el crecimiento sin perder la calidad en los servicios que ofrece. Nos encontramos con dos tipos de escalabilidad:

- *Escalabilidad horizontal*: cuando un sistema agrega más nodos para mejorar el rendimiento (NoSQL se encuentra dentro de este tipo).
- *Escalabilidad vertical*: cuando se añaden más recursos a un nodo del sistema y este mejora en su conjunto.

Como tercera característica, **no cuentan con una estructura de datos definida**. Permiten hacer uso de otros tipos de modelos de almacenamiento de información, como sistemas clave-valor, grafos, etc., de los cuales se hablará a continuación.

Cuando se habla de las características de NoSQL, es inevitable no nombrar el *teorema CAP* (también llamado *teorema de Brewer*), que dice que en sistemas distribuidos es imposible garantizar simultáneamente las siguientes propiedades:

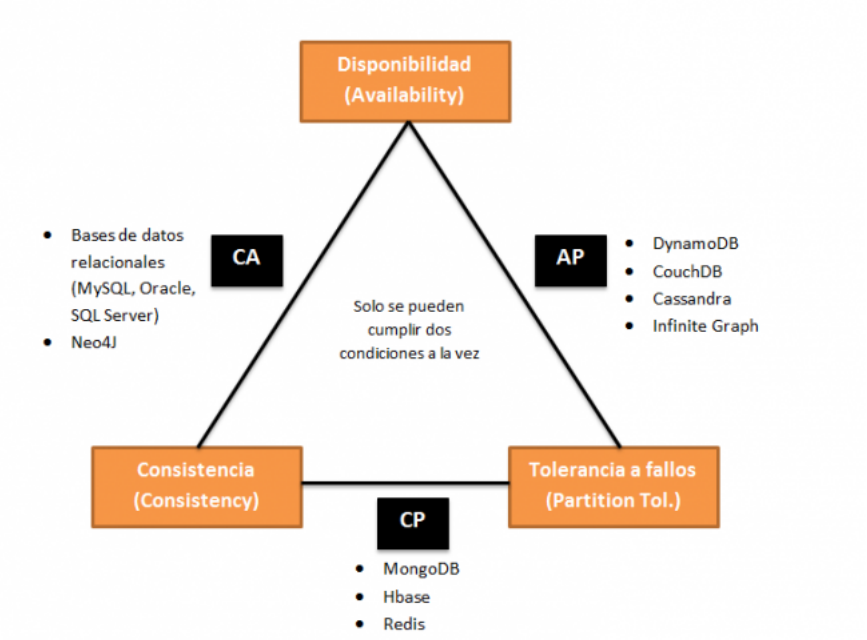
1. *Consistency* (Consistencia): hacer que todos los nodos del sistema vean la misma información al mismo tiempo. Esto se conoce como integridad de la información.
2. *Availability* (Disponibilidad): garantiza que el sistema se podrá utilizar a pesar de que uno de los nodos esté caído.
3. *Partition Tolerance* (Tolerancia al particionamiento): capacidad que tiene el sistema de seguir funcionando a pesar de que existan fallos parciales que dividan dicho sistema.

Para conocer mejor este teorema, se incluye la figura 2.2 en la que aparecen los siguientes términos:

1. *AP*: garantiza tanto la diponibilidad como la tolerancia al particionamiento, pero no la consistencia. En este grupo se encuentra nuestro objeto de estudio, Cassandra, del cual se profundizará con más detalle sobre este tema en el próximo capítulo.

2. *CP*: garantiza la consistencia y la tolerancia.
3. *CA*: indica que garantiza la consistencia y la disponibilidad, pero no la tolerancia al particionamiento.

Figura 2.2: Teorema CAP



Fuente: goo.gl/Fp0F2X

Dado que son sistemas distribuidos, cuentan con una alta velocidad y una alta capacidad de almacenamiento. Debido al bajo coste computacional que esto supone, hace que sean más baratas y por tanto, más asequibles actualmente.

Cuando se habla de inconvenientes, a veces los sistemas NoSQL pueden sufrir pérdidas de datos e inconsistencia. Debido a que son sistemas relativamente jóvenes, este hecho afecta en la compatibilidad con otros sistemas más antiguos.

2.3. Clasificación de las DDBB NoSQL

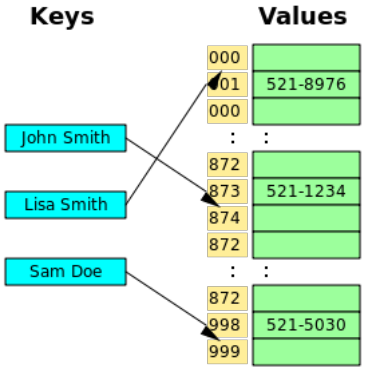
Dependiendo de cómo se almacene la información, podemos encontrar varios tipos de bases de datos NoSQL según su implementación, cada uno con diferentes características, lo que permite adaptarse a diferentes situaciones. Los más utilizados son:

- **Bases de datos clave-valor**.
- **Bases de datos documentales**.
- **Bases de datos orientadas a grafos**.
- **Bases de datos tabular**.

2.3.1. Bases de datos clave-valor

Son el modelo de datos NoSQL más popular y más sencillo de utilizar y comprender. En este tipo de sistema cada elemento está identificado por una llave única (clave), lo que permite recuperar la información de manera rápida. Se caracterizan por ser muy eficientes para lecturas y escrituras en bases de datos. *Cassandra* es una de las bases de datos clave-valor más importantes. *BigTable* (de Google), *Dynamo* (de Amazon) o *HBase* son otros ejemplos de bases de este tipo.

Figura 2.3: Representación de base de datos clave-valor

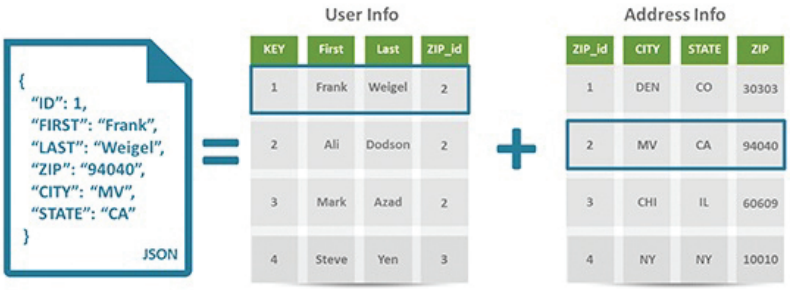


Fuente: www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf

2.3.2. Bases de datos documentales

Este tipo almacena la información como un documento, generalmente utilizando para ello una estructura sencilla como JSON (*JavaScript Object Notation*, formato ligero para el intercambio de datos) o XML (*Extensible Markup Language*, lenguaje de marcas utilizado para almacenar datos en forma legible), y donde se utiliza una clave única para cada registro. Permite además realizar búsquedas por clave-valor y realizar consultas más avanzadas sobre el contenido del documento. *MongoDB* o *CouchDB* son ejemplos de bases de datos de este tipo.

Figura 2.4: Representación de base de datos documental

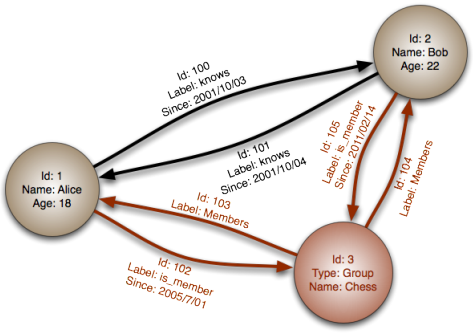


Fuente: sg.com.mx/revista/43/nosql-una-nueva-generacion-base-datos

2.3.3. Bases de datos orientadas a grafos

En este tipo de bases la información se representa como nodos de un grafo (entidades) y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos [6] para recorrerla. Para sacar el máximo rendimiento a este tipo de bases de datos, su estructura debe estar totalmente normalizada, de forma que cada tabla tenga una sola columna y cada relación dos. Gracias a esto es fácil adaptar el modelo de datos a las necesidades por su **flexibilidad**. *Neo4j*, *InfiniteGraph* o *AllegroGraph*, entre otras, son ejemplos de este tipo.

Figura 2.5: Representación de base de datos orientadas a grafos

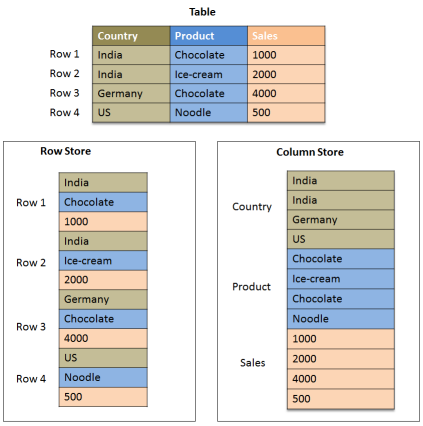


Fuente: www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf

2.3.4. Bases de datos tabular

También conocida con el nombre de *base de datos orientada a columnas*, es muy similar a las bases relacionales, pero con la diferencia de guardar los registros, guarda columnas de datos que son tratadas de manera individual. La misión de este tipo de bases es tratar con grandes cantidades de datos. La realización de consultas, o la agregación de datos de manera rápida, destaca sobre el resto. Los valores de cada columna se almacenan de manera contigua, por lo que los accesos se realizan mucho más rápido. *Cassandra*, *Hypertable* o *BigTable*, destacan en este tipo.

Figura 2.6: Representación de base de datos tabular



Fuente: sg.com.mx/revista/43/nosql-una-nueva-generacion-base-datos

En la siguiente tabla (2.2), se pueden ver los principales tipos de bases de datos NoSQL con su modelo de datos y su API de consulta.

Tipo	**Modelo de Datos**	**Query API**
Cassandra	Columnas	CQL3 / Thrift
CouchDB	Documental	Map / Reduce Views
HBase	Columnas	Thrift, REST
MongoDB	Documental	Cursor
Neo4j	Grafos	Graph
Voldemort	Clave-valor	Get/Put

Tabla 2.2: Comparación de Modelo de Datos y API [7]

Capítulo 3

Cassandra

Resumen: *Tras la pequeña introducción sobre los sistemas NoSQL, el motivo de su aparición y los distintos tipos que hay, pasamos a centrarnos en nuestro objeto de estudio, Cassandra. En este capítulo se va a dar una visión global de esta base de datos, destacando las funcionalidades y aquellas características potentes que se han mencionado en el capítulo anterior, para así comprender mejor, con ejemplos, su funcionamiento y cómo implementarla en la vida real.*

3.1. ¿Qué es Cassandra?

> Cassandra es una base de datos de tipo NoSQL, distribuida y basada en un modelo de almacenamiento de *clave-valor*, escrita en Java. [8]

Esta es la definición que encontramos nada más buscar en Internet. Uno de los objetivos de este proyecto es enfatizar más en esta definición, dejando clara cada característica de esta base de datos. El potencial que está adquiriendo el Big data [1] en estos tiempos, por la gran cantidad de información que se está manejando actualmente en Internet, hace que esta base de datos cobre significado frente a otras, gracias a su **escalabilidad horizontal** y **disponibilidad**.

El que sea una base de datos distribuida hace que permita almacenar grandes cantidades de datos. La arquitectura distribuida de Cassandra está basada en una serie de nodos, iguales, que se comunican con un protocolo P2P (*Peer-to-peer*) [9] con lo que la redundancia es máxima. Un ejemplo claro de uso de esta tecnología es la red social **Twitter** [10], donde diariamente se publican millones de *tuits* que hay que almacenar y tener disponibles para su consulta en cualquier momento. Esta red social tiene más de 300 billones de *tuits*[1] y de ahí que utilice Cassandra, ya que puede tener todos estos datos repartidos en varios nodos, fácilmente escalable y siempre disponibles para los usuarios.

Cassandra es un almacén de datos "orientado a la columna", lo que significa que en lugar de almacenar tuplas idénticas de datos, ordenadas de acuerdo a una estructura fija (el esquema de

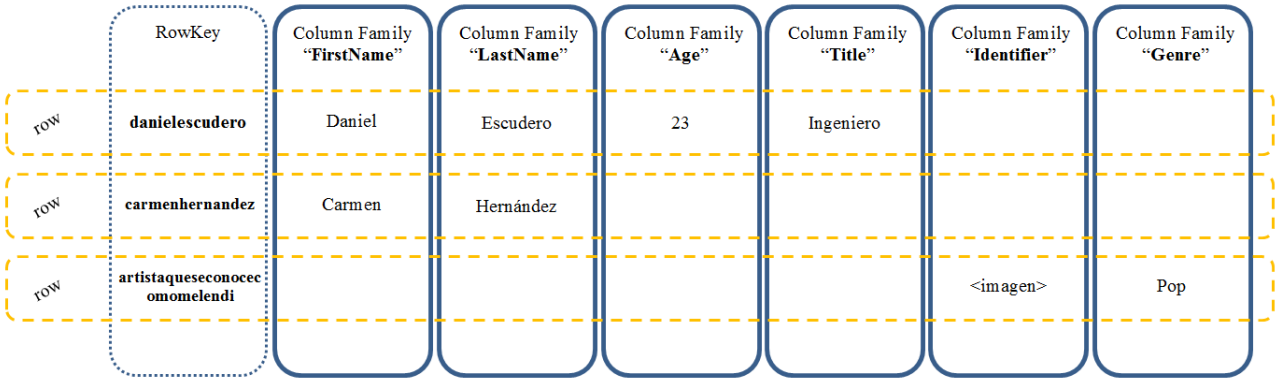
[1] Última cifra publicada en marzo de 2014.

una tabla), Cassandra almacena "familias de columnas" en *keyspaces*. Cada familia de columna se compone de filas identificadas por una clave. Es decir, Cassandra asocia un valor de clave con un número variable de pares nombre/valor (columnas) que puede ser totalmente diferente de las otras filas dentro de la familia de la columna.

En términos prácticos, se va a suponer que se está usando Cassandra para almacenar una colección de personas. Dentro del almacén de claves "Planeta" hay una *familia de columna* llamada "Gente", que a su vez tiene filas con este aspecto:

```
RowKey: danielescudero
  ColumnName:"FirstName", ColumnValue:"Daniel"
  ColumnName:"LastName", ColumnValue:"Escudero"
  ColumnName:"Age", ColumnValue:23
  ColumnName:"Title", ColumnValue:"Ingeniero"
RowKey: carmenhernandez
  ColumnName:"FirstName", ColumnValue:"Carmen"
  ColumnName:"LastName", ColumnValue:"Hernández"
RowKey: artistaqueseconocecomomelendi
  ColumnName:"Identifier", ColumnValue: <imagen>
  ColumnName:"Genre", ColumnValue:"Pop"
```

Figura 3.1: Ejemplo de una estructura de columnas



Como se puede apreciar, cada fila contiene datos conceptualmente similares, pero no todas las filas tienen porqué tener los mismos datos.

El desarrollo de Cassandra fue iniciado por la red social **Facebook**, para intentar solventar la problemática relacionada con el rendimiento del motor de búsquedas, más en concreto, con las que tenían que ver con la comunicación entre usuarios. En 2008 fue liberada por Facebook pasando a ser de código abierto (*open source*), y en 2009 donada a Apache donde se mantiene actualmente. En el año 2010 es escogida como uno de los mejores proyectos de la fundación.

3.2. Instalación de Cassandra

En el siguiente punto se va a describir el proceso de instalación de Cassandra sobre el sistema en el que se va a trabajar. Como se comentó en el apartado 1.4, aquí también se van a describir las características de los sistemas sobre los que se va a alojar Cassandra, es decir, la configuración y características de las máquinas virtuales que se van a utilizar para la parte práctica de este proyecto.

Configuración de las máquinas virtuales. Estas máquinas han sido creadas sobre VirtualBox con las siguientes características:

Características	Máquina A	Máquina B	Máquina C
Nombre máquina	**MaquinaA**	**MaquinaB**	**MaquinaC**
Memoria RAM	**512 MB**		
Disco duro	**8 GB (reserva dinámica)**		
Dirección MAC red NAT	**08:00:27:59:5D:85**	**08:00:27:6A:9E:55**	**08:00:27:00:AE:AC**
Nombre red interna	**tfgcassandra**		
Dirección MAC red interna	**08:00:27:5E:3C:0C**	**08:00:27:E4:25:46**	**08:00:27:ED:61:3D**
IP estática	**192.168.1.3**	**192.168.1.4**	**192.168.1.5**
Sistema operativo	**Linux Mint "Debian" Cinnamon (32 bits)**		
Usuario	**amaquina**	**bmaquina**	**cmaquina**
Contraseña	**maquinaA**	**maquinaB**	**maquinaC**
Nombre en red	**amaquina**	**bmaquina**	**cmaquina**

Tabla 3.1: Características de las máquinas virtuales

Por el momento, se han creado tres máquinas virtuales para probar la característica de **base de datos distribuida** y así situar mínimo un nodo en cada una de ellas. Por temas de rendimiento del equipo host sobre el que están corriendo estas máquinas, se ha tomado esta decisión de sólo crear estas tres máquinas, aunque se pueden crear tantas como el usuario quiera o necesite.

Para comunicar estas máquinas entre si (una red interna) y también tener conexión a Internet, se ha modificado el fichero `/etc/network/interfaces`, quedando así:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp

auto eth1
```

```
iface eth1 inet static
      address 192.168.1.3
      netmask 255.255.255.0
```

Como se puede ver, este es el fichero correspondiente a la *máquina A*. Para las otra dos, sería exactamente igual, sólo que cambiando la dirección IP por la que se indica en la tabla 3.1 (IP estática). De esta manera ya se puede hacer una comunicación entre las máquinas por medio del protocolo SSH (**S**ecure **SH**ell) [11].

Para una mayor comodidad en la práctica, en el fichero `/etc/hosts`, se han añadido estas direcciones IP junto con un nombre (por ejemplo el nombre de usuario), para así no tener que recordar esta dirección (en cada máquina se han añadido las otras, es decir, en el fichero de la máquina A se ha añadido la dirección de la máquina B y C, y lo mismo con las otras dos). Bastaría con hacer, por ejemplo, `ssh amaquina@amaquina -p 22`, desde la máquina B o C, para conectarnos a esta primera.

Una vez configuradas las máquinas y en funcionamiento, se procede a explicar el proceso de instalación de Cassandra sobre ellas. Como el proceso es igual para todas, se cogerá como ejemplo la instalación sobre la *máquina A*.

1. Entrar en la Web oficial del proyecto Cassandra [12] para descargar el servidor disponible. En este caso se va a utilizar la versión 2.1.2 de Cassandra.
2. Se busca el directorio donde se ha guardado el fichero comprimido y sobre el fichero, hacer doble click para entrar en el. Pulsar en la opción "Extraer" y se descomprime sobre el directorio `/tmp` del sistema.
3. Abrir un terminal de comandos y reubicar el directorio de Cassandra con el siguiente comando: `mv /tmp/apache-cassandra* $HOME/cassandra/`
4. A continuación, se crean las siguientes carpetas para la configuración de Cassandra. Para ello, basta con utilizar: `mkdir $HOME/cassandra/{commitlog,log,data,saved_caches}`.
5. Ahora se va a editar la configuración de Cassandra. Hay que situarse en el directorio en el que se ha descomprimido antes (`cd $HOME/cassandra`) y hay que abrir el fichero `cassandra.yaml` (`nano conf/cassandra.yaml`). Con esto se abrirá en el terminal el fichero para modificarlo.
6. Una vez abierto, modificar las siguientes líneas. Aquellas partes en color rojo, son las que se deben personalizar según los datos de la máquina en la que se esté:

   `data_file_directories:` (línea 98 aprox.)
   `- /home/amaquina/cassandra/data`
   `commitlog_directory:` (línea 104 aprox.)
   `- /home/amaquina/cassandra/commitlog`
   `saved_caches_directory: /home/amaquina/cassandra/saved_caches` (línea 219 aprox.)

7. Guardar el fichero. Establecer el directorio de *log*, para tenerlo más accesible. Con el comando `nano conf/log4j-server.properties`, escribir dentro del fichero que se está creando la siguiente línea:

   `log4j.appender.R.file=/home/amaquina/cassandra/log/system.log`

8. Guardar el fichero. Si no se tiene instalado el entorno Java en el sistema (versión 7 o superior), proceder a hacerlo antes de continuar. Para ello, se accede a la Web de Oracle [13] para descargar el entorno de Java. Cuando esté descargado, se procede a instalarlo:

   a) Cuando se tenga el fichero `.tar.gz`, moverlo a la carpeta de librerías de Java con `mv jdk-7*.tar.gz /usr/lib/jvm/` (necesario ser superusuario -`sudo su`- para realizar la operación).

   b) Descomprimir: `tar -zxvf jdk-7*.tar.gz`

   c) Crear un directorio de Java donde se va a extraer el contenido en el directorio de aplicaciones `/opt`: `mkdir -p /opt/java`

   d) Mover a la carpeta creada el contenido extraído anteriormente:

   `mv /usr/lib/jvm/jdk1.7.*/ /opt/java/`

   e) Hay que hacer que esta versión sea la predeterminada en el sistema:

   ```
   $ sudo update-alternatives --install "/usr/bin/java" "java" \
   > "/opt/java/jdk1.7.*/bin/java" 1
   $ sudo update-alternatives --set java /opt/java/jdk1.7.*/bin/java
   ```

   f) Con esto ya se tendría el entorno Java, compatible con Cassandra, instalado. Para comprobarlo, en consola basta con escribir: `java -version`.

9. Si se tiene la versión del entorno Java correcta, o ya está instalado, se procede a inicializar la instancia de Cassandra de la siguiente manera (es posible que se necesite ser `root`): `$ ./bin/cassandra`

10. Si todo ha ido bien, se procede a escribir el siguiente comando: `$ ./bin/nodetool --host 127.0.0.1 ring`, cuya salida confirmará que la base de datos se ha instalado correctamente en el sistema.

11. En caso de que se quiera parar la instancia de Cassandra, hay que utilizar `pgrep -u amaquina -f cassandra | xargs kill -9`. Para volverla a arrancar, basta con hacer lo que se indica en el punto 9.

Al aplicar el paso 10 de la lista anterior de configuración e instalación de Cassandra, se ve que devuelve información dividida en las siguientes columnas:

```
Address     Rack    Status State   Load       Owns        Token
                                                          9190997303966723138
127.0.0.1   rack1   Up     Normal  41,14 KB   100,00%     -9190516199280056013
```

Estos campos están dando información sobre cómo está repartida la información, dónde se encuentra y si está o no disponible, entre otras cosas.

- `Address`: indica la dirección IP del nodo.
- `Rack`: zona de disponibilidad del nodo.
- `Status`: estado de ese nodo. Dos posibles: `UP` o `DN`.
- `State`: estado del nodo en relación con el grupo (normal, salida, union, en movimiento).
- `Load`: cantidad de datos del sistema de archivos bajo el directorio de datos Cassandra.
- `Owns`: porcentaje de los datos manejados por ese nodo. Por ejemplo, si hay 3 nodos en el que la información se reparte equitativamente, este valor será del 33,33 %/nodo. En el caso mostrado, como sólo hay un nodo en el cluster, este tiene el 100 % de la información.
- `Token`: valor simbólico donde se encuentra la información. El primer valor que aparece coincide con el último de la lista, indicando así una estructura de anillo (continuidad).

Con estos pasos ya se tiene listo el entorno de Cassandra para utilizarlo. Más adelante se llevará a cabo la parte práctica, pero antes hay que conocer más a fondo sus características.

3.3. Características

A lo largo de esta memoria se han ido mencionando una serie de características que hacen de Cassandra una base de datos funcionalmente potente dentro de su clasificación, pero ahora es el momento de profundizar más en cada una de ellas.

Recordamos, Cassandra es un sistema de almacenamiento distribuido, escrito en Java, de código abierto, descentralizado, escalable, altamente disponible, tolerante a fallos, eventualmente consistente, y orientado a columnas basado en la estructura de **Dynamo** de Amazon y en el modelo de datos de **BigTable** de Google. Nació en Facebook y ahora es usado en muchos de los sitios más populares de Internet.

3.3.1. Distribuido y descentralizado

Cassandra es un sistema distribuido, lo que significa que está en la capacidad de ejecutarse sobre múltiples máquinas mientras se presenta a los usuarios como un todo. De hecho, no es provechoso ejecutar Cassandra en una única máquina, aunque sí se puede hacer. Por supuesto, se aprovecharán todos sus recursos y beneficios si se ejecuta sobre múltiple servidores.

En almacenes de datos creados con bases de datos relacionales como MySQL, cuando es necesario escalar, algunas máquinas (también llamadas nodos) deben ser validadas como administradores para coordinar las demás máquinas (llamadas esclavos). Sin embargo, Cassandra es **descentralizado**, esto significa que todos los nodos que forman parte del sistema son idénticos, ningún nodo desempeña funciones distintas a las de los otros nodos; en vez de eso, Cassandra implementa un protocolo P2P (figura 3.2) y mantiene una lista de todas las máquinas disponibles e inactivas.

La información transferida a través de este protocolo es también almacenada en disco para poder usarla en caso de reinicio del nodo. La información es intercambiada por los nodos cada segundo, manteniendo una alta tasa de sincronización.

Figura 3.2: Esquema del protocolo P2P

Fuente: blogdojoker.wordpress.com/2013/01/12/protocolo-bittorrent/

El hecho de que Cassandra sea descentralizado quiere decir que no tiene un punto de fallo, todos los nodos en un cluster (grupo de máquinas) funcionan del mismo modo; esto se denomina servidor simétrico, porque todas las máquinas hacen lo mismo. Por definición, no existe un administrador que coordine tareas entre los nodos.

En muchas soluciones de almacenamiento distribuido, se deben hacer múltiples copias de datos en un proceso llamado *replicación*. De esta forma todas las máquinas pueden trabajar con peticiones simultáneas y mejorar el desempeño del sistema. Aun así, este proceso **no** es descentralizado como en Cassandra. Para llevarlo a cabo se requiere de la definición de una relación **administrador/esclavos**, lo que quiere decir que todos los nodos no funcionan de la misma forma. El administrador es el que autoriza la distribución de los datos, y opera bajo una relación unidireccional con los nodos esclavos; si el administrador deja de funcionar, la base de datos completa está en peligro. El diseño descentralizado, por lo tanto, es una de las claves de la alta disponibilidad de Cassandra. Algunas bases de datos NoSQL como MongoDB usan la relación administrador/esclavos.

La descentralización tiene dos ventajas clave:

1. Es más fácil de usar que un administrador/esclavo. Puede ser más fácil de operar y mantener un almacén de datos descentralizado que un administrador/esclavo ya que todos los nodos son iguales, y no se requieren conocimientos adicionales para escalar; configurar 100 máquinas no es diferente a configurar una sola.
2. Un administrador puede ser un simple punto de fallo. Como en Cassandra todos los nodos son idénticos, la inutilización de uno no interrumpe su funcionamiento. Es decir, como Cas-

sandra es distribuido y descentralizado, no hay un punto de fallo, por lo que genera una alta disponibilidad.

3.3.2. Escalabilidad flexible

La escalabilidad de un sistema es poder continuar soportando un creciente número de peticiones sin afectar mucho su desempeño y rendimiento. La escalabilidad vertical (añadir más capacidad hardware y memoria a las máquinas existentes) es la forma más fácil de lograr esto. La escalabilidad horizontal (característica de Cassandra) consiste en agregar más máquinas que ayuden con el almacenamiento de información y recibir peticiones. Para esto el sistema debe tener la capacidad de sincronizar sus datos con los otros nodos en el cluster.

La **escalabilidad flexible** se refiere a una propiedad especial de la escalabilidad horizontal. Esto significa que el cluster puede escalar y *desescalar* uniformemente. Para hacer esto, el cluster debe estar capacitado para aceptar nuevos nodos que empiecen recibiendo parte o toda la información de los demás y peticiones de usuarios, sin configuraciones previas sobre el cluster. No es necesario parar el sistema ni realizar cambios en la aplicación, sólo se agrega el nodo y Cassandra lo encontrará y lo configurará para que forme parte del sistema. *Desescalar*, desde luego significa eliminar máquinas del cluster, lo cual es necesario si las peticiones al sistema disminuyen y no se requiere hardware adicional para soportarlas.

3.3.3. Alta disponibilidad y tolerancia a fallos

En términos generales, la disponibilidad de un sistema está ligada a su habilidad de cumplir peticiones al sistema, pero el sistema puede experimentar muchas formas de fallo, desde componentes hardware defectuosos, hasta problemas de conexión y comunicación. Existen computadores sofisticados (y costosos) que pueden tratar con este tipo de circunstancias. Para que un sistema sea altamente disponible, debe incluir múltiples computadores conectados, y el software que se ejecuta sobre ellos debe tener la capacidad de correr sobre un cluster e identificar posibles nodos defectuosos.

Cassandra es altamente disponible; se pueden reemplazar nodos defectuosos en un cluster sin tener que detener el sistema y también se puede replicar la información a otros almacenes de datos para mejorar el rendimiento local. Cada escritura es automáticamente particionada y replicada en el cluster para asegurar la alta disponibilidad de los datos. El **Partitioner** es el encargado de decidir cómo se distribuirán los datos y es configurable por el usuario. Por otro lado, la estrategia de réplicas determina dónde y cuántas réplicas se almacenarán en el cluster y debe ser configurada al crear el espacio de claves correspondiente.

3.3.4. Consistencia personalizable

Esto significa que se puede decidir el nivel de consistencia que se necesita en la lectura y escritura frente al nivel de disponibilidad. Esta configuración se hace a través del lenguaje de consultas en

cada caso, aunque se pueden poner unos parámentros predefinidos en la configuración del entorno.

Consistencia en escritura (Write)	
Nivel	**Descripción**
ANY	Una escritura debe ser escrita en al menos un nodo disponible.
ONE / TWO / THREE	Una escritura debe escribirse en el registro de `commit` y `memtable` de al menos un/dos/tres nodos de réplica.
QUORUM	Una escritura debe escribirse en el registro de `commit` y `memtable` en un quórum de nodos de réplica. Proporciona una fuerte consistencia si se puede tolerar cierto nivel de fracaso. Se determina por (`replication_factor`/2)+1.
LOCAL_QUORUM	Una escritura debe escribirse en el registro de `commit` y `memtable` en un quórum de nodos de réplica en el mismo *datacenter* que el nodo coordinador. Evita la latencia de comunicación entre el *datacenter*.
EACH_QUORUM	Consistencia fuerte. Una escritura debe escribirse en el registro de `commit` y `memtable` en un quórum de nodos de réplica en **todos** los *datacenter*.
ALL	Una escritura debe escribirse en el registro de `commit` y `memtable` de todos los nodos de réplica en el cluster. Proporciona la más alta consistencia y la menor disponibilidad de cualquier otro nivel.

Tabla 3.2: Niveles de consistencia para escritura

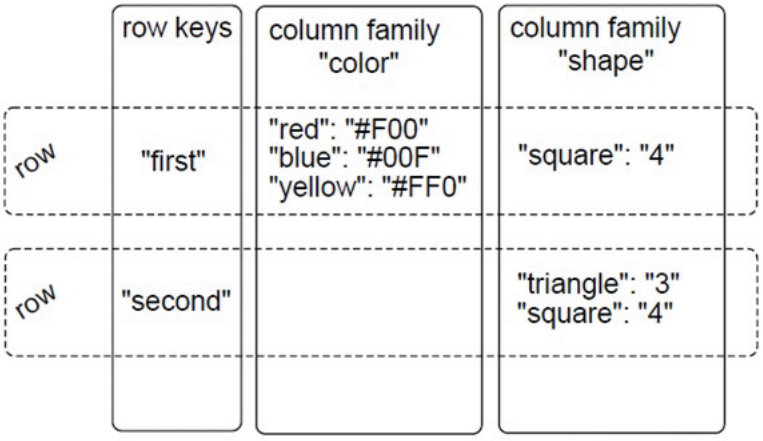
Consistencia de lectura (Read)	
Nivel	**Descripción**
ONE / TWO / THREE	Devuelve una respuesta de una/dos/tres réplicas más cercanas, según lo determinado por el coordinador. Por defecto, se ejecuta una "reparación de lectura" en segundo plano para hacer que las otras réplicas sean consistentes.
QUORUM	Devuelve el registro después de un quórum cuando n/2+1 réplicas hayan respondido.
LOCAL_QUORUM	Devuelve el registro después de un quórum cuando n/2+1 réplicas hayan respondido en el mismo *datacenter* que el nodo coordinador. Evita la latencia de comunicación entre el *datacenter*.
EACH_QUORUM	Devuelve el registro una vez que el quórum de réplicas en cada *datacenter* de la agrupación haya respondido.
ALL	Devuelve el registro después de que todas las réplicas hayan respondido.

Tabla 3.3: Niveles de consistencia para lectura

3.3.5. Orientado a columnas

Como ya se ha dicho, Cassandra no es relacional, representa las estructuras de datos como tablas hash multidimensionales, en donde cada registro puede tener una o más columnas, aunque no todos los registros de un mismo tipo deben tener el mismo número de columnas. Cada registro contiene una llave única la cual permite el acceso a los datos. Cassandra almacena los datos en tablas hash multidimensionales (concepto de *super column* que se verá más adelante), lo que significa que no es necesario tener por adelantado la representación de los datos que se va a usar y cuantos campos se necesitarán para los registros, y esto es útil cuando la estructura de datos está sujeta a cambios frecuentes. Además, Cassandra permite agregar campos para los registros aun cuando esté en servicio.

Figura 3.3: Esquema de base de datos orientada a columnas



Fuente: sg.com.mx/revista/43/nosql-una-nueva-generacion-base-datos

3.3.6. Esquema libre

Cassandra requiere definir un contenedor llamado espacio de claves (*keyspaces*) que contiene "familias de columnas". Es esencialmente un nombre para mantener estas familias de columnas y sus propiedades de configuración. Las familias de columnas o columnas comunes son nombres para información asociada. Las tablas de datos son dinámicas, así que se puede agregar información usando las columnas que quieras.

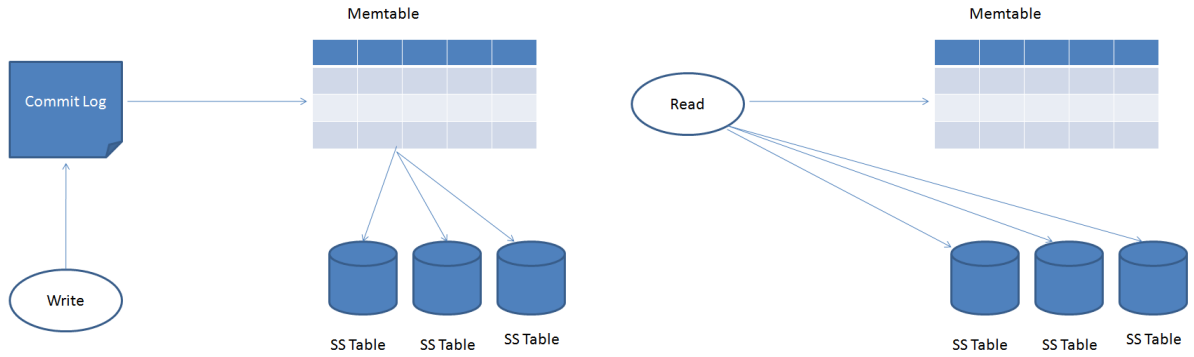
3.3.7. Alto rendimiento

Cassandra fue diseñado específicamente para aprovechar al máximo las máquinas con procesadores *multinúcleo*, y para ejecutarse sobre muchas docenas de estas máquinas alojadas en múltiples almacenes de datos. Por eso Cassandra ha demostrado capacidad suficiente en la carga masiva de

datos. Posee un alto rendimiento en escrituras por segundo. Cuantos más servidores sean agregados, se sacará mayor partido de las propiedades de Cassandra sin sacrificar rendimiento.

Cada vez que hay actividad de escritura en uno de los nodos, este lo transcribe en su log de actividad para asegurar que haya coherencia. Los datos además son almacenados en una estructura en memoria (*memtable*) y, una vez se sobrepasa el tamaño de esta, son escritos en un fichero llamado **SSTable** (Sorted String Table) [14].

Figura 3.4: Esquema de almacenamiento y lectura en una estructura en memoria



(a) Almacenamiento (b) Proceso de lectura

Fuente: bigdata-cassandra.blogspot.com.es

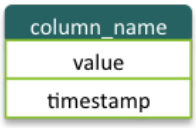
3.4. Modelo de datos

En este apartado se van a enumerar los conceptos de modelado de datos utilizados en Cassandra para comprender mejor su estructura interna.

- **Column**: es la unidad más básica de la representación en el modelo de datos Cassandra. Una *column* es un triplete de un nombre (**clave**), un **valor** y un **timestamp** (última vez que la columna fue accedida). Los valores son todos suministrados por el cliente. El tipo de dato de la *clave* y el *valor* son matrices de bytes Java, y el del *timestamp* es un `long primitive`. Las *column* son inmutables para evitar problemas de *multithreading* (multihilos), no viéndose afectadas así por la ejecución de varios hilos simultáneamente. Se organizan de manera predeterminada dentro de las *columns families* por nombre de columna si el usuario no especifica nada, pudiendo tener la columna como tipo de datos uno de los siguientes:
  - AsciiType (`ascii`).
  - BytesType (`blob`).
  - IntegerType (`varint`).
  - LongType (`int` o `bigint`).
  - BooleanType (`boolean`).

- UTF8Type (`text` o `varchar`).

Esquema visual del modelo de datos:



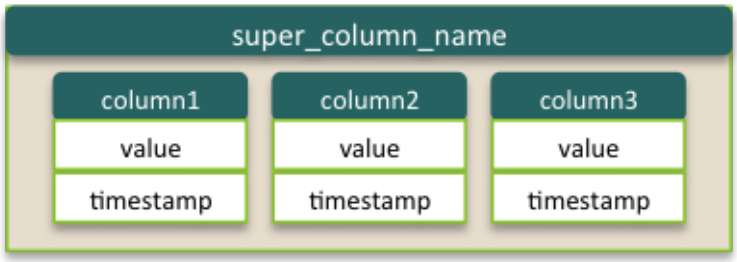

```
Columna (
  Nombre -> "Nombre del campo"
  Valor -> "Valor del campo"
  Timestap -> "Marca de tiempo"
)
```

- **Super Column**: es una *column* cuyos *values* son una o más *columns*, que en este contexto se llamarán *subcolumns*. Las *subcolumns* están ordenadas y el número de columnas que se puede definir es ilimitada. Las *super columns*, a diferencia de las *columns*, no tienen un timestamp definido. Además no son recursivas, es decir, solamente tienen un nivel de profundidad.

  Esquema visual del modelo de datos:

```
Supercolumna (
  Nombre de la columna -> "XXX" (
      columna1 -> "XXX" (
          Nombre -> "Nombre del campo"
          Valor -> "Valor del campo"
          Timestap -> "Marca de tiempo"
      )
      columna2 -> "XXX" (
          Nombre -> "Nombre del campo"
          Valor -> "Valor del campo"
          Timestap -> "Marca de tiempo"
      )
  )
)
```

Figura 3.5: Representación del modelo de datos *Super Column*

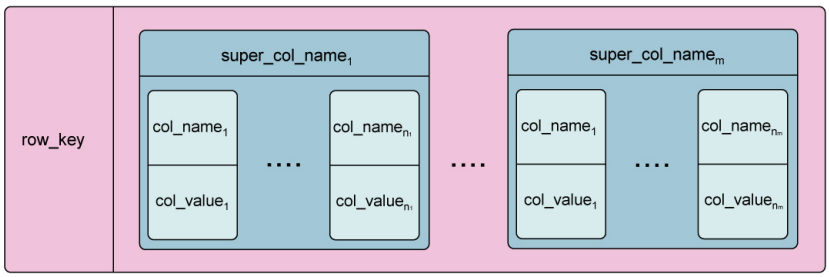


Fuente: docs.datastax.com/en/archived/cassandra/1.0/docs/ddl/column_family.html

- **Column Family**: es relativamente análogo a una tabla en un modelo relacional. Se trata de un contenedor para una colección ordenada de *columns*. Cada *column family* se almacena en un archivo separado, ordenado por clave de fila. No se pueden relacionar dos *column family* entre sí, ya que la arquitectura no lo permite. Las *column family* pueden ser de dos tipos:

  - *SuperColumnFamily*: es una tupla que consiste en un par clave-valor donde la clave está asignada a un valor que son las *columns families*. También se puede ver como un mapa de tablas (figura 3.6).

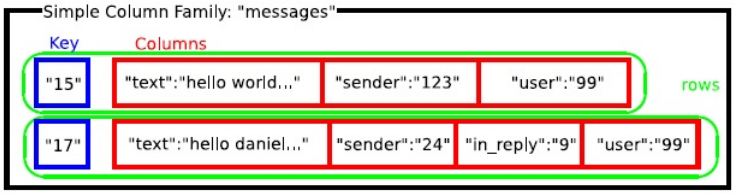
Figura 3.6: Representación del modelo de datos *SuperColumnFamily*



Fuente: www.sinbadsoft.com/blog/cassandra-data-model-cheat-sheet

  - *SimpleColumnFamily*: es un mapa de nombres de columnas ordenadas por valores de columna (figura 3.7).

Figura 3.7: Representación del modelo de datos *SimpleColumnFamily*

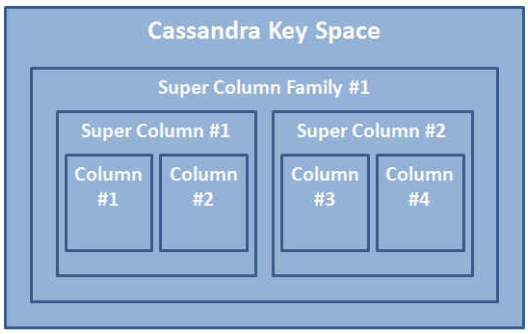


Fuente: blog.protogenist.com/?tag=column-family

- **Row**: es una agregación de *columns* o *super columns* que se referencian con un nombre. No hay más, sólo un nombre contenido dentro de un String. Ese nombre es la **clave** (key) que identifica de forma unívoca a un registro.

- **Keyspace**: es el contenedor para una o varias *columns families*. De la misma manera que una base de datos relacional es una colección de tablas, un *keyspace* es una colección ordenada de *columns families*. Se tienen que definir los *keyspaces* de la aplicación en el archivo de configuración o usando métodos definidos en la API (figura 3.8).

Figura 3.8: Representación del modelo de datos *keyspaces*

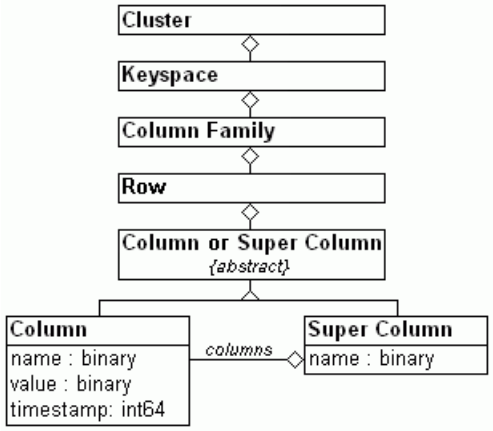


Fuente: goo.gl/mCO3MC

- **Cluster**: es el elemento de más alto nivel que puede referenciarse por un nombre. Es de naturaleza más física que los anteriores, más relacionado con el hardware, ya que agrupa los nodos (máquinas) sobre los que se ejecuta Cassandra. Puede contener uno o más *keyspaces*.

Para resumir todo lo que se acaba de ver, en la figura 3.9 se puede ver el diagrama del modelo de datos de Cassandra completo.

Figura 3.9: Modelo de datos de Cassandra



Fuente: www.inmensia.com/blog/20100327/desmitificando_a_cassandra.html

3.5. Thrift vs. CQL

Como ya se ha comentado anteriormente, las dos interfaces más conocidas en Cassandra son **Thrift** [15] y **CQL** [16]. Desde la implementación de CQL2 (y posteriores), Thrift ha pasado a un segundo plano por su dificultad al operar frente a la primera, y porque la diferencia de nivel entre ambas es bastante grande.

Thrift es un lenguaje de definición de interfaces y protocolo de comunicación [17] binario que se utiliza para definir y crear servicios para numerosos lenguajes de programación, desarrollado en un principio por Facebook (actualmente cedido y mantenido por Apache). Se utiliza como framework para realizar *Remote Procedure Call* (RPC) [18], es decir, como un proceso que se comunica con otro proceso (en este caso el de Cassandra), normalmente en otra máquina, a través de un sistema de bajo nivel de mensajería para que ejecute código en la otra máquina.

La API de Thrift ha confundido históricamente a personas que vienen del mundo relacional por el hecho de que utiliza los términos "filas" y "columnas", pero con un significado diferente que en SQL. En cambio, CQL3 fija que el modelo que expone, fila y columna tienen el mismo significado que en SQL, pese a que la traducción en la capa de almacenamiento subyacente es totalmente diferente. Este cambio de conceptos entre ambas interfaces causa mucha confusión ya que si se quiere pasar de Thirft a CQL3, una fila "Thrift" no siempre se asigna o corresponde con una fila "CQL3", y una columna "CQL3" no siempre se asigna o corresponde con una columna "Thrift".

Para evitar este tipo de confusiones, se establecen unas convenciones entre ambas interfaces:

- Siempre se va a utilizar "fila interna" cuando se quiera hablar de una fila en Thrift. Se utilizará el término "interno" porque esto corresponde a la definición de una fila en la implementación interna que Thrift expone directamente. De ahí, el término "fila" solo describirá una fila CQL3 (aunque se utilice en algún momento el término "fila CQL3" para recordar este hecho).
- El término "celda" en lugar de "columna" para definir las columnas internas de Thrift. Y así "columna" (o "columna CQL3") para designar las columnas CQL3.
- Por último, el término "*column family*" para Thrift y el término "tabla" para CQL3, aunque ambos términos pueden ser considerados como sinónimos.

En resumen, para entender la diferencia entre Thrift y CQL3 en este documento, y obtener conocimiento del lenguaje que se utiliza para cada interfaz, una fila interna contiene celdas mientras que una fila CQL3 contiene columnas, y estas dos nociones no siempre se correlacionan directamente; de ahí que a continuación, se explique cuando estos conceptos coinciden y cuando no.

Familias de columna estándar

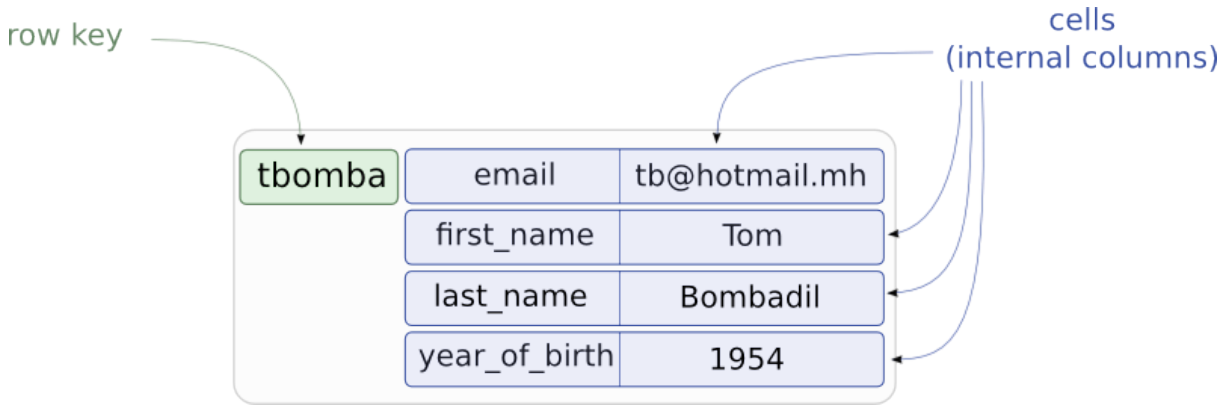
En Thrift, una *column family* estática es donde cada fila interna tendrá más o menos el mismo conjunto de nombres de celdas, y ese conjunto es finito. Un ejemplo típico es perfiles de usuario. Hay un número finito de propiedades en los perfiles que utiliza la aplicación, y cada perfil concreto tendrá algún subconjunto de esas propiedades.

Dicha *column family* estática normalmente se definiría en Thrift (con el cliente CLI [19]) con la siguiente definición:

```
create column family user_profiles
with key_validation_class = UTF8Type
 and comparator = UTF8Type
 and column_metadata = [
   column_name: first_name, validation_class: UTF8Type,
   column_name: last_name, validation_class: UTF8Type,
   column_name: email, validation_class: UTF8Type,
   column_name: year_of_birth, validation_class: IntegerType
 ]
```

Y el perfil se almacena internamente como:

Figura 3.10: *Column Family* estándar en Thrift



Fuente: www.datastax.com/dev/blog/thrift-to-cql3

La definición funcionalmente equivalente de dicha familia de columnas en CQL3 es:

```
CREATE TABLE user_profiles (
  user_id text PRIMARY KEY,
  first_name text,
  last_name text,
  email text,
  year_of_birth int
) WITH COMPACT STORAGE
```

Esta definición CQL almacenará los datos exactamente de la misma forma que la definición de Thrift mostrada. Así que para las *column family* estáticas, una fila interna se corresponde exactamente a una fila CQL3. Pero mientras que cada celda de Thrift tiene una columna CQL3 correspondiente, CQL3 define una columna (`user_id`) que no se asigna a una celda como la `PRIMARY KEY` de Thrift.

El caso es que Thrift, desde la aparición de CQL, ha comenzado a morir ya que el rendimiento en general, comparado con esta nueva interfaz, es mucho más bajo. En los gráficos que presenta DataStax [20] se puede ver cómo la velocidad en una operación de lectura en operaciones por segundo es mucho mayor y en menos tiempo en CQL que en Thrift. En una operación de escritura sucede lo mismo. Y la diferencia, en el caso de CQL es mas notoria con nuevas implementaciones de la interfaz. Lo que está claro es que se está apostando más por esta interfaz más amigable y, por supuesto, más potente.

3.6. Operaciones de Cassandra

Las operaciones que se pueden realizar en Cassandra son un tema clave para la compresión y futura implementación del entorno. A continuación se van a detallar las distintas operaciones que se pueden realizar a través de línea de comandos. La interfaz **primaria y por defecto** para comunicarse con Cassandra es **CQL**[2] (*Cassandra Query Language*) [16]. Como se puede observar, la sintaxis es muy intuitiva y parecida a SQL, con la diferencia de que Cassandra no soporta *joins* o *subqueries*.

Hay que mencionar que en las primeras versiones de Cassandra, la comunicación se realizaba por medio de APIs algo complejas que han quedado obsoletas frente a las ventajas que ofrece CQL, para el que existe una amplia documentación soportada por **DataStax** [21].

Antes de continuar, un pequeño inciso sobre la utilización de CQL en el entorno que se ha configurado. Hay que situarse en el directorio donde se encuentra Cassandra (`$HOME/cassandra/bin`) y se ejecuta el editor con el comando `cqlsh`. Opcionalmente, se puede especificar una dirección IP y un puerto para iniciar `cqlsh` en un nodo distinto (p.e.: `cqlsh 10.0.0.2 9042`).

Figura 3.11: Ejecución de `cqlsh` en nuestra máquina

```
amaquina@amaquina ~/cassandra/bin $ ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 | Native protocol v3]
Use HELP for help.
cqlsh>
```

Tras este inciso, las operaciones que se muestran a continuación son la de creación de un *keyspace* y una tabla de datos, inserción, lectura/búsqueda, modificación y borrado de estos; operaciones básicas en bases de datos.

[2]Interfaz de línea de comandos para interactuar con el sistema.

3.6.1. Operaciones básicas

Crear

Se empieza creando el *keyspace*, que es donde se almacenan todos los datos de la aplicación. Es similar a un schema en bases de datos relacionales.

```
CREATE KEYSPACE prueba WITH REPLICATION =
   { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
```

En la creación hay que indicar cómo se quiere que se distribuyan los datos en los distintos servidores y cuántas réplicas se quieren del dato:

- Estrategia de replicación:

  - `SimpleStrategy` es la estrategia más sencilla y la utilizada por defecto. Adecuada cuando solo se dispone de un *data center* simple. Esta estrategia coloca la primera réplica en un nodo determinado por el particionador. Réplicas adicionales se colocan en el siguiente nodo en el anillo, en sentido horario, sin considerar bastidor o la ubicación de los datos en el *data center*. La figura 3.12 (a) muestra tres réplicas de tres filas colocadas en cuatro nodos.

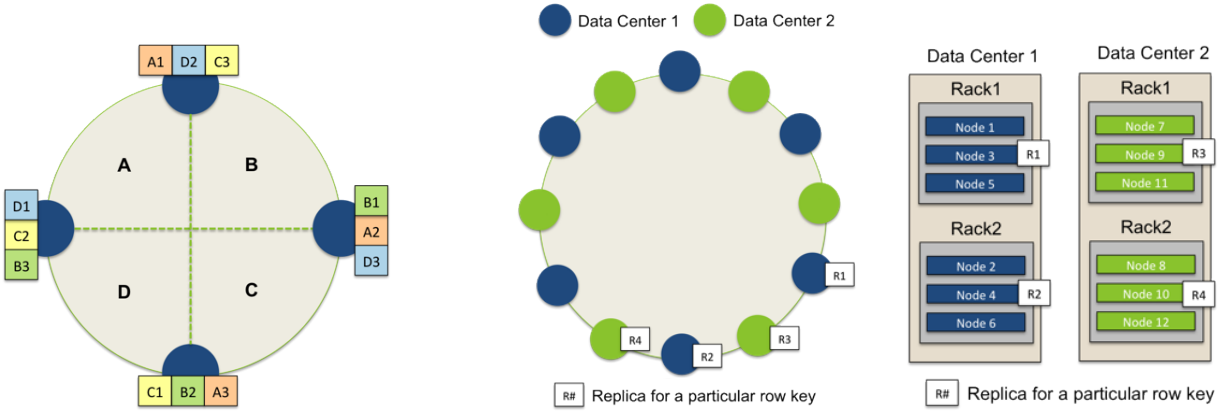
  - La otra opción es `NetworkTopologyStrategy`, adecuada cuando se tiene (o se prevé tener) el cluster desplegado a través de múltiples centros de datos. Esta estrategia especifica cuántas réplicas se desea en cada *data center*. `NetworkTopologyStrategy` intenta colocar réplicas en bastidores distintos, porque los nodos en el mismo rack (o agrupación física similar) pueden fallar al mismo tiempo debido a problemas de energía, refrigeración, o de red. Esta estrategia determina la colocación de la réplica, independiente dentro de cada centro de datos de la siguiente manera:

    - La primera réplica se coloca de acuerdo con el particionador (igual que con `Simple Strategy`).
    - Réplicas adicionales se colocan recorriendo el anillo en sentido horario hasta que se encuentre un nodo en un estante diferente. Si no existe tal nodo, réplicas adicionales se colocan en diferentes nodos en el mismo rack.

    En la figura 3.12 (b) y (c) se muestra el método de asignación con esta estrategia.

- `replication_factor` es cuántas copias se quieren del dato. El numero debería ser menor o igual que el número de servidores Cassandra disponibles. Si no hay bastantes servidores para las copias, los `INSERT` pueden fallar.

Figura 3.12: Replicación `SimpleStrategy` y `NetworkTopologyStrategy`



(a) `SimpleStrategy` (b) `NTStrategy`, Anillo (c) `NTStrategy`, Rack

Fuente: www.datastax.com/docs/1.0/cluster_architecture/replication

Si ahora se ejecuta:

```
DESC KEYSPACES;
```

Se obtienen los nombres de los *keyspaces* creados en la base de datos:

```
system system_traces prueba
```

Los espacios `system` y `system_traces` son creados por el sistema automáticamente. En el primero se incluye una serie de tablas que contienen detalles sobre los objetos de la base de datos Cassandra y la configuración del cluster. En el segundo es donde se almacena el seguimiento de una operación, cuando se activa por el usuario, para ver los pasos en la base de datos.

Si se quiere ver todo lo que hay en un *keyspace* concreto, para así ver cómo se han creado los elementos que hay en el, basta con introducir `DESC KEYSPACE <keyspace>;`.

Como sucede en las bases de datos relacionales, hay que indicar cual se quiere usar, en este caso, qué *keyspace* se quiere utilizar para realizar operaciones sobre el. La sintaxis que hay que utilizar para ello es: `USE prueba;`. De esta manera ya se está sobre el nuevo espacio creado.

Para crear una tabla (*column family* en el lenguaje propio de Cassandra) sobre el *keyspace*, se utiliza una notación como en las bases de datos de tipo relacional:

```
CREATE [TABLE|COLUMNFAMILY] usuarios (
```

```
firstname varchar,
lastname varchar,
email varchar,
organization varchar,
PRIMARY KEY (lastname));
```

Importante: a la hora de crear la tabla hay que escoger con cuidado qué se quiere que sea la clave primaria (`PRIMARY KEY`) de la tabla. Una característica especial de Cassandra es que, a la hora de utilizar la cláusula `WHERE`, sólo se puede poner la columna que sea clave primaria, sino dará error, de ahí que haya que escoger una columna que represente e identifique los datos que se van a contener.

Para ver los tipos de datos, junto con su descripción, que se pueden poner a la hora de crear o modificar columnas, consultar la tabla B.1 del apéndice B.1.

Insertar

Se procede a insertar unos pocos usuarios con los siguientes comandos:

```
INSERT INTO usuarios (firstname, lastname, email, organization)
  VALUES ('Daniel', 'Escudero', 'daniel@uva.es', 'Universidad');
INSERT INTO usuarios (firstname, lastname, email, organization)
  VALUES ('Carmen', 'Hernández', 'carmen@uva.es', 'Universidad');
INSERT INTO usuarios (firstname, lastname, email, organization)
  VALUES ('Antonio', 'Peláez', 'antonio@email.es', 'Everis');
```

Si se ejecuta: `SELECT * FROM usuarios;`, se obtiene:

```
 lastname   | email             | firstname | organization
------------+-------------------+-----------+--------------
     Peláez | antonio@email.es |   Antonio | Everis
   Escudero |    daniel@uva.es |    Daniel | Universidad
  Hernández |    carmen@uva.es |    Carmen | Universidad

(3 rows)
```

Buscar

Al igual que en bases de datos SQL, se pueden hacer consultas como buscar dentro de la tabla una entrada en concreto, o entradas según un parámetro, gracias a la clausula `WHERE`. Por ejemplo,

si con los datos que se han insertado, se quiere saber los datos de alguien que se apellide "Escudero", se buscaría de la siguiente manera:

```
SELECT * FROM usuarios WHERE lastname = 'Escudero';
```

Obteniendo como resultado:

```
 lastname   | email             | firstname | organization
------------+-------------------+-----------+--------------
   Escudero |    daniel@uva.es |    Daniel | Universidad

(1 rows)
```

Modificar

Tras ver como se insertan elementos en el sistema, se pasa a ver una variante, que es la modificación. Cassandra nunca lee antes de escribir, no comprueba si los datos ya existen al hacer un `INSERT`, por lo tanto `UPDATE` e `INSERT` sobreescriben las columnas de una entrada sin importar los datos ya almacenados.

```
UPDATE usuarios SET email = 'antonio@everis.es' WHERE lastname = 'Peláez';
```

Al hacer una búsqueda de la tabla (o del cambio que se ha realizado), se ve que el registro ha cambiado.

```
 lastname   | email              | firstname | organization
------------+--------------------+-----------+--------------
     Peláez | antonio@everis.es |   Antonio | Everis
   Escudero |     daniel@uva.es |    Daniel | Universidad
  Hernández |     carmen@uva.es |    Carmen | Universidad

(3 rows)
```

Eliminar

Para eliminar una entrada de la tabla se usa la sintaxis:

```
DELETE FROM usuarios WHERE lastname = 'Peláez';
```

Quedando la tabla ahora con dos registros. `SELECT * FROM usuarios;`

```
 lastname   | email             | firstname | organization
------------+-------------------+-----------+--------------
   Escudero |    daniel@uva.es |    Daniel | Universidad
  Hernández |    carmen@uva.es |    Carmen | Universidad

(2 rows)
```

Hasta aquí ya se conocen las operaciones básicas que se pueden realizar en Cassandra bajo la interfaz **CQL**, interfaz que permite hacer operaciones con los conocimientos que se tienen sobre la sintaxis de SQL, ya que no varía en nada en las operaciones elementales que se han visto. Se pueden realizar operaciones más completas, añadiendo cláusulas a las sentencias mencionadas anteriormente, como por ejemplo cambiar un parámetro de una instancia en CQL con el comando `ALTER`, crear índices o añadir más columnas a una tabla, entre otras cosas.

3.7. Cassandra desde Java

Como se ha comentado, Cassandra está implementado en el lenguaje de programación Java, y por tanto, en esta sección se va a dar a conocer como se puede acceder al servidor de Cassandra desde este lenguaje, a través de un cliente.

Clientes

Un cliente es una API que permite conectar y realizar operaciones contra Cassandra. Como ya se ha comentado, básicamente hay dos protocolos de acceso a Cassandra: **CQL3** y **Thrift**.

En los ejemplos que se mostrarán a partir de aquí, si no se dice lo contrario, se utilizará CQL3. Para ello se necesita el driver para la gestión y compilación correcta. Se puede descargar directamente desde el repositorio de Cassandra [23], añadirlo al compilar el código fuente de Java en línea de comandos, o con la siguiente dependencia:

```
<dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>1.0.6</version>
</dependency>
```

Una vez se tenga el driver configurado, el código para conectarse a un cluster Cassandra es el siguiente:

```
1 Cluster cluster1 =
    Cluster.builder().addContactPoints("172.30.210.11","172.30.210.12").build();
```

Se pueden ir añadiendo varias IPs o nombres de host, correspondientes a distintos servidores donde corran ejecutables de Cassandra dentro del módulo. En principio sólo es necesario dar una

IP y a partir de ella el driver será capaz de "interrogar" a Cassandra por todas las demás IPs del conjunto de servidores. Sin embargo, es buena idea dar más de una por si la única que se da está caída en ese momento; ahí el driver intentará conectarse a la siguiente.

Una vez que se tiene el *cluster*, hay que conectarse (dos maneras):

```
// Sin indicar keyspace
Session sesion = cluster1.connect();

// Indicando un keyspace
Session sesion = cluster1.connect("keyspace");
```

La opción sin *keyspace* es adecuada para crear por primera vez un *keyspace* y un conjunto de tablas asociadas a él. Una vez exista el *keyspace* y las tablas, se podría usar la conexión con el *keyspace*, ahorrándose así en las sentencias de `SELECT`, `INSERT`, `UPDATE` el poner el *keyspace* delante de cada tabla. Ahora lo que hay que hacer es crear un *keyspace* (por lo que se utilizará la primera opción de conexión al *cluster*) ya que se supone que no hay ninguno creado. Para ello, en la clase Java se escribe:

```
sesion.execute("CREATE KEYSPACE prueba " +
   "WITH replication = " +
   "{ 'class' : 'SimpleStrategy', 'replication_factor' : 1 }");
```

Se acaba de crear un *keyspace* llamado `prueba`. En la creación, se indica cómo se quiere que se distribuyan los datos en los distintos servidores y cuántas réplicas se quieren del dato. En el punto 3.6.1 ya se ha indicado qué se debe tener en cuenta.

El siguiente paso es crear una tabla.

```
session.execute(
      "CREATE TABLE prueba.canciones (" +
            "id uuid PRIMARY KEY," +
            "titulo text," +
            "album text," +
            "artista text," +
            "tags set<text>," +
            "data blob" +
            ");");
```

Ya que se ha utilizado la conexión en la que no se indicaba *keyspace*, es necesario poner éste delante del nombre de la tabla, `prueba.canciones`. El resto son columnas que se añaden a la tabla.

Una vez creado el *keyspace* y la tabla (*column family* en el lenguaje propio de Cassandra), se insertan datos. Se puede usar la misma sesión que ya hay abierta, la que no está asociada a ningún *keyspace*, o se puede abrir una sesión nueva asociada al *keyspace* `prueba`, para mayor comodidad. Si se usa la sesión existente que no está asociada a ningún *keyspace*, se debe poner delante de la

tabla el nombre del *keyspace*, como se ha hecho a la hora de crearla.

Para cambiar, se va a conectar con el método de *keyspace* existente, quedando la conexión y la inserción de datos de la siguiente manera:

```
sesion = cluster1.connect("prueba");
String uuid = UUID.randomUUID().toString();
sesion.execute("insert into canciones (id,title,album,artist,tags,data) "
   + " values ("
   + uuid
   + ",'titulo','album','artista',{'etiqueta1','etiqueta2'},0xffffffff);");
```

Si se fija en la forma de asignar un **ID**, hay que saber que en Cassandra no se generan automáticamente los ID, no existen secuencias o claves autoincrementales como en SQL; es el programa el encargado de generarlos. Se deben diseñar las *columns families* de acuerdo a las consultas que se quieran hacer, por lo que las claves deberían ser algún campo de valor único que tenga sentido para un usuario (como el DNI de una persona). Si se necesita un ID que no sea parte de uno de los campos del modelo de datos, lo habitual es usar un **UUID** [24], que se puede generar fácilmente de forma aleatoria con Java, por medio de la clase `UUID` y su método `randomUUID()`.

Una vez hecha la inserción de datos en la tabla, se puede actualizar el valor de alguna columna con la sentencia `UPDATE`. En Cassandra, en el `WHERE` debe ir **obligatoriamente** la `PRIMARY KEY` de la *column family*. Por eso es importante que se defina como clave primaria algo que forme parte del modelo de datos y tenga sentido desde el punto de vista del usuario.

```
sesion.execute("update canciones set titulo = 'nuevo titulo' where id = "+uuid);
```

Una vez que se tiene el *keyspace*, tabla y datos en esta, se pueden hacer búsquedas. El código en Java sería el siguiente, para buscar los elementos de una tabla:

```
// Suponemos que conocemos el uuid
ResultSet rs = sesion.execute("select * from canciones where id = "+uuid);
Iterator<Row> iterador = rs.iterator();
while (iterador.hasNext()){
   System.out.println(iterador.next().getString("titulo"));
}
```

El `ResultSet` que se obtiene no es el estándar de `java.sql.ResultSet`, sino uno específico del driver de Cassandra, `com.datastax.driver.core.ResultSet`.

Se crea un iterador y se van recorriendo las filas de resultado. Se puede pedir cada columna por su nombre, como en el ejemplo, los títulos. Si se quiere consultar por cualquier otra columna que no sea el **ID**, Cassandra obliga a tener creados índices secundarios para esas columnas que se quieren usar en el `WHERE`. Por ejemplo, en el momento de crear las tablas, si se prevee que se van a hacer consultas por la columna `titulo`, se puede crear un índice de esta forma:

```
sesion.execute("CREATE INDEX idx_titulo ON canciones (titulo)");
```

Con este índice ya es posible hacer consultas usando la columna `titulo` en la cláusula `WHERE`:

```
rs = sesion.execute("select * from canciones where titulo = 'nuevo titulo'");
iterador = rs.iterator();
while (iterador.hasNext()){
    Row fila = iterador.next();
    System.out.println(fila.getUUID("id")+" - "+fila.getString("titulo"));
}
```

Por último, la última operación básica que queda por realizar desde Java es el borrado de datos. La forma de hacerlo es muy parecida a las anteriores:

```
sesion.execute("delete from canciones where id = "+uuid);
```

3.8. Time Series Data

Dentro de las capacidades de Cassandra, se destaca en este punto una muy importante a mi criterio, como el uso de **series temporales de datos** (*time series data*) [26] para el análisis de negocios.

El uso de las series temporales de datos para analizar los negocios no es un movimiento nuevo. Lo que es nuevo es la capacidad de recopilar y analizar grandes volúmenes de datos en secuencia a gran velocidad para obtener la imagen más precisa que sea capaz de predecir los cambios futuros del mercado, el comportamiento de los usuarios, las condiciones ambientales, consumo de recursos, las tendencias de salud y mucho más.

Cassandra en este punto juega un papel muy importante ya que, como se ha comentado, es una plataforma superior de base de datos NoSQL para este tipo de retos en Big data. El modelo de datos de Cassandra, del que ya se ha hablado en el punto 3.3, es un excelente ajuste para el manejo de datos en secuencia, independientemente del tipo de datos o el tamaño. Al escribir datos en Cassandra, se ordenan y escriben secuencialmente en el disco. Al recuperar los datos por clave-fila y luego por rango, se obtiene un patrón de acceso rápido y eficiente debido a las mínimas solicitudes que se hacen al disco, de ahí que la serie temporal de datos sea un ajuste idóneo para este tipo de patrón. Cassandra permite a las empresas identificar las características significativas en sus series temporales de datos lo más rápido posible para tomar decisiones claras sobre los resultados futuros esperados.

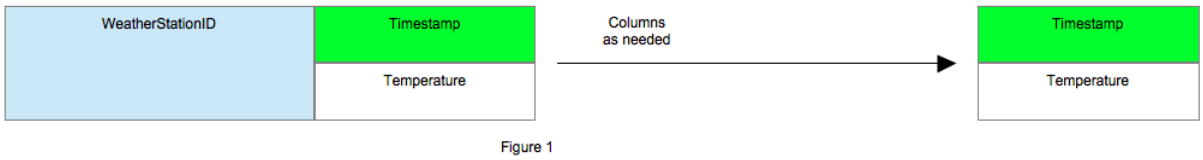
Pongamos un ejemplo para entender mejor esta característica que hace de Cassandra una base de datos más atractiva aún.

Patrón de series temporales 1 - dispositivo individual por fila

El modelo más simple para el almacenamiento de series temporales de datos es la creación de una gran fila de datos para cada fuente. En este primer ejemplo, se va a utilizar el ID de una

estación meteorológica como la clave de la fila. La fecha y hora de lectura será el nombre de la columna y la temperatura el valor de columna (figura 3.13). Dado que cada columna es dinámica, la fila crecerá según sea necesario para acomodar los datos.

Figura 3.13: Patrón 1 de series temporales



Fuente: planetcassandra.org/getting-started-with-time-series-data-modeling

```
CREATE TABLE temperature (
    weatherstation_id text,
    event_time timestamp,
    temperature text,
    PRIMARY KEY (weatherstation_id,event_time)
);
```

Con una simple consulta, se buscan todos los datos en una sola estación meteorológica:

```
SELECT event_time,temperature
FROM temperature
WHERE weatherstation_id='xxxxx';
```

O una consulta buscando datos entre dos fechas. Esto también se conoce como una 'rebanada'ya que leerá un intervalo de datos desde el disco:

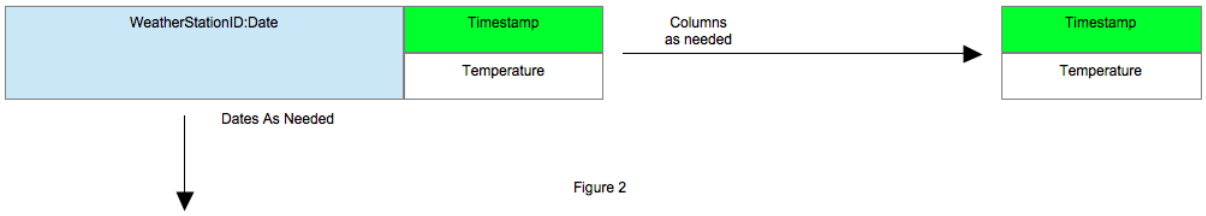
```
SELECT temperature
FROM temperature
WHERE weatherstation_id='xxxxx'
AND event_time > 'AAAA-MM-DD HH:MM:SS'
AND event_time < 'AAAA-MM-DD HH:MM:SS';
```

Patrón de series temporales 2 - partición del tamaño de la fila

En algunos casos, la cantidad de datos recogidos para un único dispositivo no es práctico para que quepan en una sola fila. Cassandra puede almacenar hasta dos mil millones de columnas por fila, pero si se estan almacenando datos cada milisegundo, no se podrá obtener el dato buscado en meses. La solución es utilizar un patrón llamado **partición de fila** mediante la adición de datos a la clave de la fila, para limitar la cantidad de columnas que se obtiene por dispositivo. Usando

los datos ya disponibles, se puede utilizar la fecha de la marca de tiempo y añadirla al ID de la estación meteorológica. Esto dará una fila por día por cada estación meteorológica, consiguiendo una forma fácil de encontrar los datos (figura 3.14).

Figura 3.14: Patrón 2 de series temporales



Fuente: planetcassandra.org/getting-started-with-time-series-data-modeling

```
CREATE TABLE temperature_by_day (
    weatherstation_id text,
    date text,
    event_time timestamp,
    temperature text,
    PRIMARY KEY ((weatherstation_id,date),event_time)
);
```

Hay que tener en cuenta la clave de la fila `(weatherstation_id,date)`. Cuando se hace esto en la definición de la clave primaria, la clave pasa a estar formada por los dos elementos. Ahora, cuando se inserten los datos, el grupo que forma la clave con todos los datos del tiempo en un solo día estará en una sola fila.

Para obtener todos los datos del tiempo para un solo día, se pueden consultar usando los dos elementos de la clave primaria:

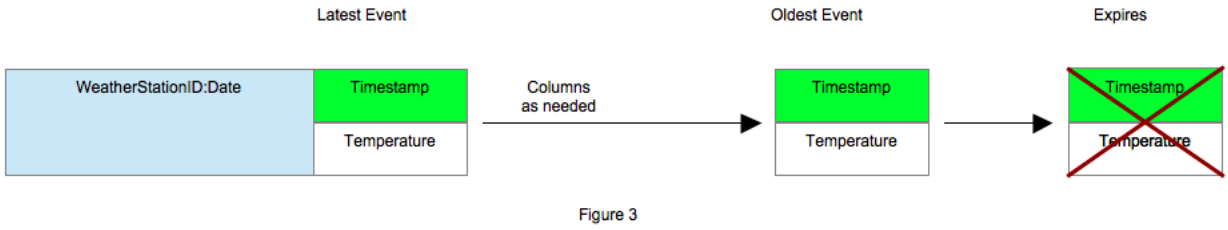
```
SELECT *
FROM temperature_by_day
WHERE weatherstation_id='xxxxx'
AND date='AAAA-MM-DD';
```

Patrón de series temporales 3 - columnas que expiran

Imaginemos que estamos utilizando estos datos para una aplicación de escritorio o móvil y sólo se quieren mostrar las últimas diez lecturas de temperatura. Los datos más antiguos ya no son útiles, por lo que se pueden eliminar de la página. Con otras bases de datos, habría que configurar un trabajo en segundo plano para limpiar los datos más antiguos. Con Cassandra, se puede

tomar la ventaja de una característica llamada *expiring columns*, donde los datos desaparecen tranquilamente después de una determinada cantidad de segundos (figura 3.15).

Figura 3.15: Patrón 3 de series temporales



Fuente: planetcassandra.org/getting-started-with-time-series-data-modeling

```
CREATE TABLE latest_temperatures (
    weatherstation_id text,
    event_time timestamp,
    temperature text,
    PRIMARY KEY (weatherstation_id,event_time),
) WITH CLUSTERING ORDER BY (event_time DESC);
```

A la hora de insertar los datos, hay que tener en cuenta el campo (TTL) donde se configura el tiempo en el que dato expirará. Por ejemplo, al insertar el dato:

```
INSERT INTO latest_temperatures(weatherstation_id,event_time,temperature)
    VALUES ('1234ABCD','2013-04-03 07:03:00','72F') USING TTL 60;
```

significa que a los 60 segundos ese dato desaparecerá.

Con esta propiedad se pueden hacer grandes cosas.

En definitiva, las series temporales de datos son uno de los modelos de datos más convincentes para Cassandra. Es un ajuste natural para el modelo de datos de una tabla grande y escala bien bajo una amplia variedad de cambios.

3.9. Otros comandos de CQL

En este apartado se van a detallar brevemente algunos de los comandos que se pueden utilizar en la base de datos, desde el editor CQL, y que son útiles para diversas operaciones, a parte de los que ya se han mencionado en el punto 3.6. Muchos de ellos son ya conocidos porque realizan las mismas operaciones que en bases de datos SQL.

<table>
<tr><th>Comando</th><th>Descripción</th></tr>
<tr><td>ALTER</td><td>Permite modificar un keyspace, una tabla, un usuario o un tipo añadiendo o quitando elementos de ese elmento a modificar. Por ejemplo, si se tiene una tabla y se quiere añadir una nueva columna, bastaría con poner: <code>ALTER TABLE tabla ADD columna tipo;</code></td></tr>
<tr><td>DROP</td><td>Permite eliminar, sin manera de recuperarlo, un índice, un keyspace, una tabla (y toda su información), un usuario, un tipo o un trigger. Por ejemplo, si se tiene una índice y se quiere borrar, bastaría con poner: <code>DROP INDEX IF EXISTS índice;</code></td></tr>
<tr><td>TRUNCATE</td><td>Elimina de manera irreversible e inmediata los datos de una tabla. Ejemplo: <code>TRUNCATE tabla;</code></td></tr>
<tr><td>GRANT/REVOKE</td><td>Comandos para dar o quitar permisos a un usuario. Por ejemplo, si se quiere dar todos los permisos a un usuario en un keyspace, bastaría con: <code>GRANT ALL ON keyspace TO usuario;</code>
La siguiente tabla muestra los permisos que se pueden dar o quitar:
<table>
<tr><th>Permisos</th><th>Comandos CQL</th></tr>
<tr><td>ALL</td><td>Todos los comandos</td></tr>
<tr><td>ALTER</td><td><code>ALTER KEYSPACE, ALTER TABLE, CREATE, INDEX, DROP INDEX</code></td></tr>
<tr><td>AUTHORIZE</td><td><code>GRANT, REVOKE</code></td></tr>
<tr><td>CREATE</td><td><code>CREATE KEYSPACE, CREATE TABLE</code></td></tr>
<tr><td>DROP</td><td><code>DROP KEYSPACE, DROP TABLE</code></td></tr>
<tr><td>MODIFY</td><td><code>INSERT, DELETE, UPDATE, TRUNCATE</code></td></tr>
<tr><td>SELECT</td><td><code>SELECT</code></td></tr>
</table>
</td></tr>
</table>

Tabla 3.4: Comandos para usar en el editor CQL

La especificación más completa y con ejemplos de los comandos que se acaban de detallar, y el resto de comandos que se pueden utilizar, están disponibles en la documentación oficial de la última versión de CQL proporcionado por DataStax [27].

Capítulo 4

Caso práctico

Resumen: *En los capítulos anteriores ya se ha conocido el origen de Cassandra, las bases de datos NoSQL, su sintaxis y qué se puede hacer con esta base de datos. En este último punto se va a llevar a cabo un caso práctico para poner en conocimiento del lector el funcionamiento de Cassandra y poner en práctica los aspectos explicados. De esta manera, se sacará partido al objetivo de Cassandra, almacenar grandes cantidades de información, en varios sitios y que sea fácil y rápido de obtener la información solicitada. Gracias a esto, se podrán sacar unas conclusiones para reflexionar sobre el futuro de este tipo de bases de datos.*

4.1. Introducción

Para el desarrollo práctico que se va a llevar a cabo en este capítulo, se necesita obtener grandes colecciones de información para introducir en la base de datos, al igual que configurar el entorno de acuerdo a las características tecnológicas de las que disponemos. Al ser un caso práctico, se comentará todo aquello que se ha utilizado y de dónde se ha obtenido la información que nos ayudará para llegar al éxito del mismo.

El fichero con la información que se va a importar en la base de datos que se va a crear a continuación, se ha obtenido de una Web pública oficial (**United States Census Bureau** [29]) donde se recogen datos de negocios del país que realiza estos estudios. Se ha escogido este fichero por tener gran cantidad de información, con algo más de 36.600 registros, requisito para las pruebas que se quieren realizar. Además la información de este fichero es muy variada, con 18 columnas, pudiendo así realizar búsquedas algo complejas, aunque el contenido en un principio no es relevante para las pruebas. Otra de las razones de la elección de este fichero es por estar en formato CSV (*comma-separated values*), algo que simplificará el trabajo de importación en el entorno. Una vez se tenga toda la información cargada en Cassandra, se realizarán diversos casos de prueba, en relación a la inserción de datos, búsquedas, creación de índices, borrado; todo esto analizando por debajo los mecanismos que establece el gestor en función de la operación, qué nodos participan o dónde se aloja la información.

Todas las pruebas que se van a realizar a continuación se harán sobre la interfaz CQL, por su sencillez en el lenguaje de consultas y su capacidad a la hora de operar.

4.2. Preparación del entorno

A continuación, se va a preparar el entorno de acuerdo a la tecnología que se dispone. Para realizar las pruebas que se han marcado, el mínimo de nodos que se va a habilitar es **tres**, para que así se pueda probar la propiedad de la distribución. En la tabla 3.1 se dan los detalles de las tres máquinas virtuales creadas donde se van a alojar cada uno de los tres nodos de la base de datos. En todas las máquinas está instalada la misma versión de Cassandra y configurado para funcionar, pero hay que hacer unos pequeños ajustes más, de manera concreta, para tener estos nodos comunicados, ya que hasta el momento no se "conocen" entre ellos.

Estos ajustes se van a hacer sobre el fichero de configuración de Cassandra, que según la configuración inicial se encuentra en `$HOME/cassandra/conf/cassandra.yaml`. Los campos que se van a modificar en este fichero son:

- `num_tokens: 256` (línea 27 aprox.): es el valor recomendado para todos los nodos. Si, por ejemplo, se pretendiera ofrecer solo la mitad de carga al primer nodo, por las razones que sean (menos capacidad de hardware y/o software, etc.), entonces a este se le daría el valor 128 y al resto de nodos, 256, el predeterminado. Como en este caso no hay ninguna restricción, ya que todas las máquinas son iguales, se deja el valor recomendado.

  Otra opción válida en la configuración, pero no simultánea a lo que se acaba de describir, es generar con un script la asignación de *tokens* según el número de nodos disponibles. Esto se debe configurar (comentando la variable anterior) en la variable `initial_token` (línea 33 aprox.), que sirve para configurar Cassandra para múltiples nodos. En este caso, se necesita saber de antemano cuántos nodos se van a utilizar, y calcular un número simbólico para cada uno. DataStax ha desarrollado un script en Python para calcular este número en función del número de nodos, y así asignar en la configuración de cada uno el valor correspondiente. Si se sabe con certeza que el número de nodos no va a variar, se puede utilizar esta opción, ya que incluir a posteriori más nodos en el cluster supondría volver a repetir este proceso para cambiar los valores.

  A continuación se detalla el proceso:

  - Descargar el script en Python que proporciona DataStax para el cálculo del número simbólico de cada nodo. Se puede encontrar en `raw.github.com/riptano/ComboAMI/2.2/tokentoolv2.py`, o en el apéndice C de este documento.
  - Cuando se tenga el script guardado, abrir un terminal y sobre el directorio donde se encuentra el fichero, darle permisos de ejecución (`chmod +x tokengentool.py`).

- Ahora se procede a calcular los números para cada nodo. Para ello, en el terminal, escribir el siguiente comando, pasándole por parámetro el valor 3, que es el número de nodos que en este caso se van a utilizar (`./tokengentool.py 3`).
- Los valores que devuelve el programa son:

```
{
    "0": {
        "0": 0,
        "1": 56713727820156410577229101238628035242,
        "2": 113427455640312821154458202477256070485
    }
}
```

  Y esos serán los números que se deben poner a cada nodo en el campo `initial_token`.

La configuración que se ha aplicado en este caso es la de poner valor a la variable `num_tokens`, por temas de sencillez en la configuración, ya que si se quiere añadir posteriormente otro nodo, no se tiene que recalcular el valor simbólico para cada uno de ellos de nuevo.

- `seeds` (línea 265 aprox.): aquí irán las IP's de los distintos nodos, separadas por comas. En este caso, la línea quedaría de la siguiente manera:

  `- seeds: ''192.168.1.3,192.168.1.4,192.168.1.5''`

  También se puede dejar la dirección IP de *localhost* de la máquina, aunque no es recomendable:

  `- seeds: ''127.0.0.1,192.168.1.4,192.168.1.5''`

- `listen_address: ip.de.la.maquina` (línea 371 aprox.): es recomendable cambiar *localhost* por la IP local de cada máquina en cada nodo.

  `listen_address: 192.168.1.3`

- `rpc_address: ip.de.la.maquina` (línea 412 aprox.): es recomendable cambiar *localhost* por la IP local de cada máquina en cada nodo.

  `rpc_address: 192.168.1.3`

Con estos cambios, más los que se hicieron anteriormente, bastaría para la complejidad de este caso de estudio, ya que se ha definido la carga de cada nodo, la comunicación de los nodos, y las direcciones de escucha.

Una vez hechos todos estos pasos, arrancar Cassandra en cada máquina (ver punto 9), y para ver que todos los nodos estan activos, escribir, en cualquiera, en el terminal `./bin/nodetool status`. En este caso, todo ha funcionado correctamente como se puede ver en la figura 4.1, y todos los nodos estan comunicados de manera correcta y listos para insertar información.

Figura 4.1: Ejecución de `./bin/nodetool status` para comprobar los nodos

```
cmaquina cassandra # ./bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load       Tokens  Owns    Host ID                               Rack
UN  192.168.1.3  58,7 KB    256     ?       18d66bce-40f7-441d-af89-7e02e0410d49  rack1
UN  192.168.1.4  61,33 KB   256     ?       b8ac7cb5-3a51-412b-81f8-667b0a1dcdee  rack1
UN  192.168.1.5  61,38 KB   256     ?       31eb26ab-632e-4a08-9213-55beb551d477  rack1
```

Si alguno de los nodos no estaría operativo, a la hora de comprobar el estado, se vería algo similar a lo de la figura 4.2, donde en este caso, los nodos de las máquinas B y C están fuera de servicio (DN).

Figura 4.2: Ejecución de `./bin/nodetool status` para comprobar los nodos (1)

```
amaquina@amaquina ~/cassandra $ ./bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load       Tokens  Owns    Host ID                               Rack
DN  192.168.1.5  ?          256     ?       31eb26ab-632e-4a08-9213-55beb551d477  rack1
DN  192.168.1.4  ?          256     ?       b8ac7cb5-3a51-412b-81f8-667b0a1dcdee  rack1
UN  192.168.1.3  98,37 KB   256     ?       18d66bce-40f7-441d-af89-7e02e0410d49  rack1
```

4.3. Importación de datos

Una vez preparado el entorno de trabajo y comprobado que todo funciona como se pretende, es momento de guardar información en la base de datos. Todas las pruebas que se hagan a continuación, salvo que se diga lo contrario, se van a hacer con los tres nodos activos, aunque con algunas de ellas no sería necesario tener la propiedad de la distribución activa, pero como ya se tiene configurado el entorno, se continúa así.

En primer lugar, y como ya se ha comentado anteriormente en el modelo de datos de Cassandra (3.4), se debe crear un espacio donde guardar toda esta información, el *keyspace*. Para ello, dentro del editor CQL, se crea un espacio de claves (figura 4.3).

Figura 4.3: Creación del *keyspace* donde guardar las *columns families*

```
Connected to Test Cluster at 192.168.1.5:9042.
[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 | Native protocol v3]
Use HELP for help.
cqlsh> CREATE KEYSPACE cassandratfg WITH replication = {'class': 'NetworkTopologyStrategy', '192.168.1.3': 3, '192.168.1.4': 3, '192.168.1.5': 3};
```

Se indica que el factor de replicación sea 3, igual al número de nodos de los que se dispone (se

pueden poner menos, pero nunca más de los nodos disponibles) en nuestro cluster, y la estrategia de distribución, como se ha descrito en el punto 3.6.1, se pone a `NetworkTopologyStrategy` porque hay varios nodos en distintas máquinas (o centros de datos).

Una vez creado este "contenedor", se procede a importar el fichero en formato CSV con toda la información. Para ello, primero se tiene que crear la tabla (*column family*) con el total de las columnas y el tipo de cada una, para que así el gestor identifique cada campo. Como ya se ha comentado, no todas las filas deben tener el mismo número de columnas con información, pero a la hora de crear la tabla antes de una importación, se deben definir todas las columnas ya que este método no es capaz de crear columnas a mayores y asignarles un tipo. Sin embargo, luego el gestor, internamente, para diversas operaciones, no tiene en cuenta esta estructura que estamos creando, ya que se guía por la clave única de cada registro.

Se abre el fichero CSV para conocer las columnas y el tipo de datos que forman cada una y a partir de ahí, se crea la tabla donde se guardará toda la información, teniendo el siguiente aspecto:

```
CREATE [TABLE|COLUMNFAMILY] IF NOT EXISTS cassandratfg.datos (
    policyID int,
    statecode varchar,
    county varchar,
    eq_site_limit double,
    hu_site_limit double,
    fl_site_limit double,
    fr_site_limit double,
    tiv_2011 double,
    tiv_2012 double,
    eq_site_deductible double,
    hu_site_deductible double,
    fl_site_deductible double,
    fr_site_deductible double,
    point_latitude double,
    point_longitude double,
    line varchar,
    construction varchar,
    point_granularity int,
    PRIMARY KEY (policyID)
);
```

Como se puede observar, a la hora de crear la tabla (*column family*), se puede usar cualquiera de las dos maneras que se muestra en la sintaxis. Se indica también que se cree si no existe una tabla con ese nombre. La clave primaria de la tabla (por donde se identificará cada registro) está formada por el campo `policyID`, que es único. Se ha escogido este esquema de datos ya que

es el que se ajusta a la información del fichero y es bastante sencillo de entender. Se podría haber creado una clave primaria compuesta o utilizar colecciones de datos en un campo, pero al no tener la información preparada de esa manera, se ha optado por la opción indicada.

Para crear esta tabla, se puede hacer directamente sobre el editor `cqlsh`, o desde fuera, teniendo en un fichero de texto plano este comando. Se va a realizar de la segunda manera, para ver otra forma de cargar scripts en el editor.

Se guarda el comando para crear la tabla en un fichero `.txt`, y desde un terminal normal, se escribe lo siguiente para crear la tabla en la base de datos, en el *keyspace* ya creado:

```
$ ./$HOME/cassandra/bin/cqlsh -k cassandratfg -f $HOME/createtable.txt
```

Si todo ha ido bien, la consola no devolverá ningún resultado, y se puede comprobar entrando en la base de datos, si la tabla está bien creada. Esto mismo se puede hacer desde dentro del editor `cqlsh` con el comando `SOURCE` y el mismo fichero, indicando su localización.

Ahora ya se puede cargar la información. Se va a utilizar el comando nativo `COPY` que permite tanto importar como exportar datos en formato CSV hacia y desde Cassandra de manera rápida y cómoda. La sintaxis del comando, para **importar** es la siguiente:

```
COPY table_name ( column, ...)
FROM ( 'file_name' | STDIN )
WITH option = 'value' AND ...
```

Para ver el uso de las opciones en la cláusula `WITH`, consultar la tabla B.2 en el apéndice B.2, para así cambiar la lectura del formato CSV, si así fuera necesario.

Escribir lo siguiente desde el editor `cqlsh` para cargar la información en la base de datos:

```
cqlsh:cassandratfg> COPY cassandratfg.datos (policyID, statecode, county,
  eq_site_limit, hu_site_limit, fl_site_limit, fr_site_limit, tiv_2011, tiv_2012,
  eq_site_deductible, hu_site_deductible, fl_site_deductible, fr_site_deductible,
  point_latitude, point_longitude, line, construction, point_granularity)
  FROM 'cargaStates.csv'
  WITH HEADER=TRUE AND DELIMITER=',';
```

Con esto se habrá cargado en nuestro espacio la información que estaba cargada en el fichero CSV. Para comprobarlo, basta con hacer una consulta básica sobre la tabla.

El número de registros cargados en la tabla `cassandratfg.datos` ha sido de 36.634 y el sistema una ver terminado el proceso, indica el tiempo que ha tardado en escribir esta información en la base de datos (figura 4.4).

Figura 4.4: Importación de datos en la tabla de la base de datos

```
Processed 36000 rows; Write: 417.08 rows/s
36634 rows imported in 1 minute and 26.787 seconds.
cqlsh:cassandratfg>
```

Ha tardado en importar toda la información en la tabla 1 minuto y 26.8 segundos, y lo ha hecho a poco más de 417 líneas por segundo, un ritmo bastante alto.

4.4. Casos de Pruebas

Una vez se encuentran los datos cargados en la herramienta, se va a proceder a realizar la siguiente batería de pruebas para ver el comportamiento que tiene Cassandra ante diferentes casos:

1. Comprobación de carga de cada nodo tras la subida de datos.
2. Seguimiento de la actividad ante diversas operaciones en la base de datos.
3. Búsquedas con valores exactos, con las cláusulas `AND`, `OR`, de rango (comparativa de tiempos).
4. Caida de nodos (provocada) y ver qué sucede con la información.
5. Cambio del factor de replicación y su comportamiento.
6. Seguridad.

4.4.1. Prueba 1: Comprobación de carga de cada nodo

En este primer caso de prueba, se va a dar a conocer el nivel de carga de los nodos configurados que forman el cluster en la base de datos, después de la importación masiva de datos sobre la tabla que se ha creado. Cabe recordar que cuando se ha hizo la configuración inicial del *datacenter* que alberga la base de datos, se indicó que los nodos tendrían una carga equitativa, ya que no existían restricciones de espacio en ciertos nodos. Por tanto, la carga en los tres nodos que forman la red en este caso, debe ser la misma.

Para ello, desde el terminal, basta con introducir el comando `./bin/nodetool status` y fijarse en la columna `Load` donde se indica la carga de cada nodo, tal como se indica en la figura 4.5.

Figura 4.5: Carga de los nodos que forman la base de datos

```
cmaquina cassandra # ./bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens  Owns    Host ID                               Rack
UN  192.168.1.3   7,33 MB    256     ?       18d66bce-40f7-441d-af89-7e02e0410d49  rack1
UN  192.168.1.4   7,33 MB    256     ?       b8ac7cb5-3a51-412b-81f8-667b0a1dcdee  rack1
UN  192.168.1.5   7,34 MB    256     ?       31eb26ab-632e-4a08-9213-55beb551d477  rack1
```

Esta información se puede obtener desde el terminal de cualquiera de los nodos. En este caso, la carga de cada uno es de 7,33 MB (el sistema ajusta el resto de la carga en el último nodo que forma la red si el reparto no es exacto), siendo una carga total de aproximadamente 22 MB.

Conclusión: Se observa que la configuración inicial en la que la carga debía ser igual para todos los nodos ha funcionado según lo esperado y que gracias a este reparto habrá un mayor rendimiento y tiempo de respuesta ya que no hay ningún nodo saturado ni con exceso de datos con respecto a otros.

4.4.2. Prueba 2: Seguimiento de la actividad de la base de datos

Para activar (o desactivar) el seguimiento de las peticiones de lectura/búsqueda o escritura, se puede utilizar el comando específico de Cassandra `TRACING`, que después de activarlo, registrará la actividad de la base de datos creando un fichero que permite ayudar al usuario y entender las operaciones internas de Cassandra y solucionar problemas de rendimiento. Durante 24 horas Cassandra guarda la información de seguimiento de las tablas, que están en el espacio de claves `system_traces`.

Esta será la prueba más importante de este proyecto, ya que se tendrá una visión de lo que sucede en la capa subyacente durante una operación, ver dónde consulta datos o dónde los guardas, cómo se comunican los nodos y las operaciones que realiza para obtener un alto rendimiento.

La información que se registra cuando se activa el seguimiento se almacena en las dos siguientes tablas, que están formadas por los campos que se detallan a continuación:

```
CREATE TABLE sessions (
    session_id uuid PRIMARY KEY,
    coordinator inet,
    duration int,
    parameters map<text, text>,
    request text,
    started_at timestamp
);

CREATE TABLE events (
    session_id uuid,
    event_id timeuuid,
    activity text,
    source inet,
    source_elapsed int,
    thread text,
    PRIMARY KEY (session_id, event_id)
);
```

La columna `source_elapsed` almacena el tiempo transcurrido, en microsegundos, antes de que el evento ocurra en el nodo origen. Para mantener la información de seguimiento, hay que copiar los datos de las sesiones y de las tablas de eventos en otra ubicación.

Seguimiento de una solicitud de escritura

En la tabla, en un seguimiento/trazado de una solicitud de escritura, se muestra:

- El coordinador (nodo desde el que se ejecuta la consulta) identifica los nodos de destino para la replicación de la fila.
- Escribe la fila en `commitlog` y `memtable`.
- Confirma la finalización de la solicitud.

Primero se activa el seguimiento desde el editor CQL con el siguiente comando:

```
cqlsh:tfgcassandra> TRACING ON;
```

Ahora se prueba a insertar una nueva fila en la base de datos; por ejemplo, con lo siguientes datos:

```
cqlsh:cassandratfg> INSERT INTO datos (policyID, statecode, county, line
        point_granularity) VALUES (0, 'ES', 'CASTILLA LEON', 'Community', 9);
```

Tras realizar la inserción, con todos los nodos de la base de datos funcionando, Cassandra proporciona una descripción de cada paso que se necesita para satisfacer la solicitud, los nombres de los nodos que se ven afectados, el tiempo de cada paso, y el tiempo total para la solicitud.

La siguiente porción de código muestra el valor que ha devuelto la ejecución de la inserción, para poder analizarlo:

```
Tracing session: 2bfb47c0-dc9a-11e4-bdc6-a16cc8128d83

 activity                                                  | timestamp                  | source      | source_elapsed
-----------------------------------------------------------+----------------------------+-------------+----------------
                                    Execute CQL3 query | 2015-04-06 22:19:04.8920 | 192.168.1.3 |              0
 Message received from /192.168.1.3 [Thread-7] | 2015-04-06 22:19:04.8380 | 192.168.1.4 |             94
   Appending to commitlog [SharedPool-Worker-1] | 2015-04-06 22:19:04.8440 | 192.168.1.4 |           7019
  Adding to datos memtable [SharedPool-Worker-1] | 2015-04-06 22:19:04.8450 | 192.168.1.4 |           7365
 Enqueuing response to /192.168.1.3 [SharedPool-Worker-1] | 2015-04-06 22:19:04.8450 | 192.168.1.4 |           7787
 Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 2015-04-06 22:19:04.8500 | 192.168.1.4 |          12142
                                    Parsing statement | 2015-04-06 22:19:04.8920 | 192.168.1.3 |            129
                  Preparing statement [SharedPool-Worker-1] | 2015-04-06 22:19:04.8920 | 192.168.1.3 |            352
 Determining replicas for mutation [SharedPool-Worker-1] | 2015-04-06 22:19:04.8930 | 192.168.1.3 |            843
 Sending message to /192.168.1.5 [WRITE-/192.168.1.5] | 2015-04-06 22:19:04.8960 | 192.168.1.3 |           4364
 Sending message to /192.168.1.4 [WRITE-/192.168.1.4] | 2015-04-06 22:19:04.8970 | 192.168.1.3 |           5157
   Appending to commitlog [SharedPool-Worker-1] | 2015-04-06 22:19:04.8970 | 192.168.1.3 |           5459
```

```
                Adding to datos memtable [SharedPool-Worker-1] | 2015-04-06 22:19:04.8980 | 192.168.1.3 |          5559
                Message received from /192.168.1.4 [Thread-5] | 2015-04-06 22:19:04.9120 | 192.168.1.3 |            --
Processing response from /192.168.1.4 [SharedPool-Worker-2] | 2015-04-06 22:19:04.9130 | 192.168.1.3 |            --
                Message received from /192.168.1.5 [Thread-7] | 2015-04-06 22:19:04.9250 | 192.168.1.3 |            --
Processing response from /192.168.1.5 [SharedPool-Worker-4] | 2015-04-06 22:19:04.9250 | 192.168.1.3 |            --
                Message received from /192.168.1.3 [Thread-7] | 2015-04-06 22:19:04.9790 | 192.168.1.5 |            84
                Appending to commitlog [SharedPool-Worker-1] | 2015-04-06 22:19:04.9820 | 192.168.1.5 |          3163
                Adding to datos memtable [SharedPool-Worker-1] | 2015-04-06 22:19:04.9820 | 192.168.1.5 |          3314
    Enqueuing response to /192.168.1.3 [SharedPool-Worker-1] | 2015-04-06 22:19:04.9830 | 192.168.1.5 |          3492
      Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 2015-04-06 22:19:05.0050 | 192.168.1.5 |         25842
                                        Request complete | 2015-04-06 22:19:04.8977 | 192.168.1.3 |          5733
```

Se puede ver en las trazas que devuelve que hay tres etapas distintas en una inserción simple como la que se acaba de hacer:

- El coordinador (quien ejecuta la consulta) se da cuenta de cuáles son los nodos activos en los que se puede hacer la réplica (en verde), y envía la solicitud a un nodo "al azar", siempre que se encuentren balanceados si la configuración es equitativa.
- La réplica (en amarillo) añade la fila a `commitlog`, que actúa como un registro de recuperación de fallos para los datos, y a continuación, lo agrega a la `memtable`, que es una caché en memoria con el contenido almacenado. Los datos en la `memtable` están ordenados por clave para que la recuperación posterior sea más rápida, y solo hay una `memtable` por *column family*.
- El coordinador recibe una confirmación por parte de la réplica y le dice al usuario que la solicitud se ha realizado correctamente.

Una operación de escritura nunca se considerará exitosa al menos hasta que los datos se hayan añadido al registro `commitlog`. Si la información no aparece aquí, la inserción no se efecutará.

Cuando la unidad de memoria lógica `memtable` está llena, los datos se escribirán ordenados y de forma secuencial en el disco en **SSTables** (Sorted String Table) [14], los cuales están compuestos por tres archivos:

- **Bloom Filter**: Para la optimización de lectura, determina si este SSTable contiene la clave solicitada.
- **Index**: Índice de la ubicación de los datos por la clave.
- **Data**: Los datos de la columna.

Conclusión: Tras la inserción en la base de datos, se confirma gracias a la traza que el gestor ha devuelto, que identifica los nodos activos del cluster y de manera aleatoria inserta en uno el dato, guardando en los ficheros de memoria la ubicación del dato y en el de consistencia el éxito de la operación e indicando al nodo coordinador que la inserción se ha realizado correctamente.

Seguimiento de una solicitud de búsqueda

En este caso se va a ver un trazado algo más complicado, se va a hacer una búsqueda en la tabla cuyo `policyID` sea 172534, por ejemplo.

```
cqlsh:cassandratfg> SELECT *
                    FROM datos
                    WHERE policyID = 172534;
```

La siguiente porción de código muestra el valor que ha devuelto la ejecución de la búsqueda, para poder analizarlo:

Tracing session: 7fa41240-dc9e-11e4-bdc6-a16cc8128d83

activity	timestamp	source	source_elapsed
Execute CQL3 query	2015-04-06 22:50:03.236000	192.168.1.3	0
Message received from /192.168.1.3 [Thread-7]	2015-04-06 22:50:03.212000	192.168.1.4	139
Executing single-partition query on datos [SharedPool-Worker-1]	2015-04-06 22:50:03.215000	192.168.1.4	2444
Acquiring sstable references [SharedPool-Worker-1]	2015-04-06 22:50:03.215000	192.168.1.4	2488
Merging memtable tombstones [SharedPool-Worker-1]	2015-04-06 22:50:03.215000	192.168.1.4	2596
Bloom filter allows skipping sstable 6 [SharedPool-Worker-1]	2015-04-06 22:50:03.215000	192.168.1.4	2766
Parsing select * from datos WHERE policyid = 172534; [SharedPool-Worker-1]	2015-04-06 22:50:03.237000	192.168.1.3	170
Preparing statement [SharedPool-Worker-1]	2015-04-06 22:50:03.237000	192.168.1.3	366
Sending message to /192.168.1.4 [WRITE-/192.168.1.4]	2015-04-06 22:50:03.241000	192.168.1.3	4832
Message received from /192.168.1.3 [Thread-7]	2015-04-06 22:50:03.284000	192.168.1.5	64
Executing single-partition query on datos [SharedPool-Worker-2]	2015-04-06 22:50:03.293000	192.168.1.5	9485
Acquiring sstable references [SharedPool-Worker-2]	2015-04-06 22:50:03.297000	192.168.1.5	13303
Merging memtable tombstones [SharedPool-Worker-2]	2015-04-06 22:50:03.299000	192.168.1.5	15439
Bloom filter allows skipping sstable 6 [SharedPool-Worker-2]	2015-04-06 22:50:03.301000	192.168.1.5	17455
Sending message to /192.168.1.5 [WRITE-/192.168.1.5]	2015-04-06 22:50:03.302000	192.168.1.3	65174
Partition index with 0 entries found for sstable 5 [SharedPool-Worker-1]	2015-04-06 22:50:03.341000	192.168.1.4	129056
Seeking to partition beginning in data file [SharedPool-Worker-1]	2015-04-06 22:50:03.343000	192.168.1.4	130812
Partition index with 0 entries found for sstable 5 [SharedPool-Worker-2]	2015-04-06 22:50:03.367000	192.168.1.5	83189
Seeking to partition beginning in data file [SharedPool-Worker-2]	2015-04-06 22:50:03.368000	192.168.1.5	84011
Skipped 0/2 non-slice-intersecting sstables, included 0 due to tombstones [SharedPool-Worker-1]	2015-04-06 22:50:03.391000	192.168.1.4	178772
Merging data from memtables and 1 sstables [SharedPool-Worker-1]	2015-04-06 22:50:03.391000	192.168.1.4	178825
Skipped 0/2 non-slice-intersecting sstables, included 0 due to tombstones [SharedPool-Worker-2]	2015-04-06 22:50:03.393000	192.168.1.5	108264
Merging data from memtables and 1 sstables [SharedPool-Worker-2]	2015-04-06 22:50:03.394000	192.168.1.5	110010
Read 1 live and 0 tombstoned cells [SharedPool-Worker-1]	2015-04-06 22:50:03.395000	192.168.1.4	182282
Enqueuing response to /192.168.1.3 [SharedPool-Worker-1]	2015-04-06 22:50:03.395000	192.168.1.4	182643
Sending message to /192.168.1.3 [WRITE-/192.168.1.3]	2015-04-06 22:50:03.395000	192.168.1.4	182978
Read 1 live and 0 tombstoned cells [SharedPool-Worker-2]	2015-04-06 22:50:03.396000	192.168.1.5	112211
Enqueuing response to /192.168.1.3 [SharedPool-Worker-2]	2015-04-06 22:50:03.396000	192.168.1.5	112409
Sending message to /192.168.1.3 [WRITE-/192.168.1.3]	2015-04-06 22:50:03.396000	192.168.1.5	112615
Message received from /192.168.1.5 [Thread-7]	2015-04-06 22:50:03.416000	192.168.1.3	179280
Processing response from /192.168.1.5 [SharedPool-Worker-3]	2015-04-06 22:50:03.417000	192.168.1.3	180890

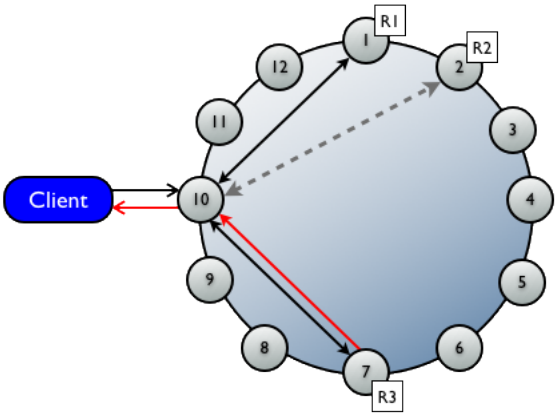
Message received from /192.168.1.4 [Thread-5]	2015-04-06 22:50:03.426000	192.168.1.3	--
Processing response from /192.168.1.4 [SharedPool-Worker-3]	2015-04-06 22:50:03.436000	192.168.1.3	--
Request complete	2015-04-06 22:50:03.417169	192.168.1.3	181169

Este proceso es considerablemente más largo ya que se hace una búsqueda secuencial por todo el clúster.

- El coordinador (quien ejecuta la consulta) establece las réplicas para consultar (amarillo) en cada uno de los nodos, hasta llegar al lugar donde se encuentre el dato, comprobando primero si la caché `memtable` aún contiene los datos.
- Si se encuentra un índice de partición que no contiene resultados de la búsqueda, se busca particionar al inicio del fichero de datos (rojo). Entonces Cassandra leerá **todos** los SSTables para esa *column family*.
- En caso de encontrar algún salto en la base de datos durante la búsqueda, como ha sucedido en este caso (azul), aprovecha para juntar esas partes (*merge*) y así para próximas búsquedas optimiza mejor el tiempo.
- El valor devuelto aparece en el nodo 192.168.1.4 y otra réplica en el 192.168.1.5 (verde).
- El coordinador recibe una confirmación por parte de la/s réplica/s (gris) y le dice al usuario que la solicitud se ha realizado correctamente.

Si sucede algún contratiempo durante la operación que pueda generar alguna inconsistencia, el gestor realiza una "lectura de reparación" (*read repair*) que actualiza las réplicas con la versión más reciente de los datos a leer. Por ejemplo, en un clúster con un factor de replicación 3, y un nivel de consistencia de lectura QUORUM, 2 de las 3 réplicas para la fila dada se ponen en contacto para cumplir con la solicitud de lectura. Suponiendo que las réplicas habían contactado con diferentes versiones de la fila, la réplica con la versión más reciente devolvería los datos solicitados (figura 4.6). En el fondo, la tercera réplica comprueba la coherencia con los dos primeros y si es necesario, la réplica más reciente emite una escritura en las réplicas fuera de fecha.

Figura 4.6: Proceso *read repair*



Fuente: docs.datastax.com/en/cassandra/1.2/cassandra/architecture/architectureClientRequestsRead_c.html

Conclusión: Como se puede ver en el caso de prueba, no pasa ni 1 segundo desde el inicio de la ejecución hasta obtener el resultado, sobre una búsqueda de entre más de 35.000 registros. Esto hace ver la potencia del gestor con los índices a la hora de consultar datos, pese al recorrido que hace sobre los nodos para comprobar que devuelve todos los datos, y pese a ser una operación más costosa que la de escribir en la base de datos. Con la traza se puede confirmar el proceso de que consulta en el fichero de memoria dónde se encuentra el dato a buscar y cuando lo encuentra, confirma al nodo coordinador del éxito de la operación.

Seguimiento de una solicitud de borrado

De las operaciones básicas en bases de datos, sólo queda ver qué es lo que sucede por debajo cuando se quiere borrar un dato de la base de datos, y cómo afecta a los nodos que forman el clúster. Por eso, en este caso se va a borrar un elemento de la base de datos, por ejemplo el que se ha insertado anteriormente.

```
cqlsh:cassandratfg> DELETE FROM datos
                    WHERE policyID = 0;
```

La siguiente porción de código muestra el valor que ha devuelto la ejecución del borrado, para poder analizarlo:

```
Tracing session: 20f0dca0-dd6c-11e4-b5f2-a16cc8128d83

 activity                                                 | timestamp                  | source      | source_elapsed
----------------------------------------------------------+----------------------------+-------------+----------------
                                    Execute CQL3 query | 2015-04-07 23:22:00.6830 | 192.168.1.3 |              0
          Message received from /192.168.1.3 [Thread-7] | 2015-04-07 23:22:00.6500 | 192.168.1.4 |             52
                               Appending to commitlog | 2015-04-07 23:22:00.6500 | 192.168.1.4 |            935
                             Adding to datos memtable | 2015-04-07 23:22:00.6510 | 192.168.1.4 |           1344
                  Enqueuing response to /192.168.1.3 | 2015-04-07 23:22:00.6510 | 192.168.1.4 |           1533
 Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 2015-04-07 23:22:00.6530 | 192.168.1.4 |           3858
          Message received from /192.168.1.3 [Thread-5] | 2015-04-07 23:22:00.6790 | 192.168.1.5 |             43
          Parsing DELETE FROM datos WHERE policyid = 0; | 2015-04-07 23:22:00.6830 | 192.168.1.3 |            320
                                    Preparing statement | 2015-04-07 23:22:00.6830 | 192.168.1.3 |            685
                   Determining replicas for mutation | 2015-04-07 23:22:00.6840 | 192.168.1.3 |           1323
                               Appending to commitlog | 2015-04-07 23:22:00.6850 | 192.168.1.5 |           5543
                             Adding to datos memtable | 2015-04-07 23:22:00.6850 | 192.168.1.5 |           5685
                  Enqueuing response to /192.168.1.3 | 2015-04-07 23:22:00.6850 | 192.168.1.5 |           5889
 Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 2015-04-07 23:22:00.6870 | 192.168.1.5 |           7360
 Sending message to /192.168.1.5 [WRITE-/192.168.1.5] | 2015-04-07 23:22:00.6910 | 192.168.1.3 |           8476
 Sending message to /192.168.1.4 [WRITE-/192.168.1.4] | 2015-04-07 23:22:00.6920 | 192.168.1.3 |           9629
                               Appending to commitlog | 2015-04-07 23:22:00.6920 | 192.168.1.3 |           9982
                             Adding to datos memtable | 2015-04-07 23:22:00.6930 | 192.168.1.3 |          10078
        Message received from /192.168.1.4 [Thread-10] | 2015-04-07 23:22:00.6990 | 192.168.1.3 |             --
                 Processing response from /192.168.1.4 | 2015-04-07 23:22:00.6990 | 192.168.1.3 |             --
          Message received from /192.168.1.5 [Thread-4] | 2015-04-07 23:22:00.7000 | 192.168.1.3 |             --
```

```
                Processing response from /192.168.1.5 | 2015-04-07 23:22:00.7010 | 192.168.1.3 |             --
                                      Request complete | 2015-04-07 23:22:00.6934 | 192.168.1.3 |          10379
```

Conclusión: El punto a destacar en esta traza tras borrar un dato de la base de datos es, que tras preparar la query para borrar, se determina el número de réplicas existentes en la base de datos (amarillo), para enviar un mensaje a los nodos de que tienen que ejecutar la query (rojo) y guardar el registro en la `memtable` antes de confirmar la operación.

Seguimiento de una solicitud de búsqueda con índice

Ahora se va a examinar el uso del seguimiento para diagnosticar problemas de rendimiento. Se crea un índice sobre una columna en la que no tiene mucho sentido por la *población* de datos que la forman.

Se crea un índice sobre la columna `COUNTY` para luego realizar una consulta sobre el valor que se ha insertado en el punto 4.4.2, donde se sabe que es el único con ese nombre sobre los más de 36.000 existentes, de los cuales muchos se repiten.

```
cqlsh:cassandratfg> CREATE INDEX ON datos (county);
```

Ahora se busca en la base de datos el valor que se ha insertado por esta columna:

```
cqlsh:cassandratfg> SELECT *
                    FROM datos
                    WHERE county = 'CASTILLA LEON';
```

Se dejan las trazas suficientes para centrarse en los pasos importantes de la operación de búsqueda con índice:

```
Tracing session: ec575070-de2c-11e4-a8b4-a16cc8128d83

 activity                                                          | timestamp           | source      | source_elapsed
-------------------------------------------------------------------+---------------------+-------------+----------------
                                        Execute CQL3 query | 2015-04-08 22:22:05 | 192.168.1.3 |              0
                                         Parsing statement | 2015-04-08 22:22:05 | 192.168.1.3 |             57
                                         Peparing statement | 2015-04-08 22:22:05 | 192.168.1.3 |            682
                                 Determining replicas to query | 2015-04-08 22:22:05 | 192.168.1.3 |           6623
                                                    [ ... ]
                                Sending message to /192.168.1.4 | 2015-04-08 22:22:05 | 192.168.1.3 |         105485

                             Message received from /192.168.1.3 | 2015-04-08 22:22:05 | 192.168.1.4 |             29
Executing indexed scan for (max(0), max(3074457345618258602)] | 2015-04-08 22:22:05 | 192.168.1.4 |           1844
   Executing single-partition query on datos.datos_county_idx | 2015-04-08 22:22:05 | 192.168.1.4 |           2692
                              Acquiring sstable references | 2015-04-08 22:22:05 | 192.168.1.4 |           2712
                                Merging memtable contents | 2015-04-08 22:22:05 | 192.168.1.4 |           2740
                            Scanned 36634 rows and matched 1 | 2015-04-08 22:22:05 | 192.168.1.4 |         112842
                            Enqueuing response to /192.168.1.3 | 2015-04-08 22:22:05 | 192.168.1.4 |         112861
                              Sending message to /192.168.1.3 | 2015-04-08 22:22:05 | 192.168.1.4 |         112938

                                          [ ... mismos pasos para 192.168.1.5 ... ]
                                            Request complete | 2015-04-08 22:22:05 | 192.168.1.3 |         291378
```

Conclusión: Cassandra tiene que escanear todas las filas para encontrar el `county` que se está buscando. No merece la pena crear un índice sobre una columna en la que todos los valores sean iguales, porque no se discrimina nada al hacer la búsqueda en el índice. Por lo tanto, se necesitan crear índices adecuados para obtener un rendimiento óptimo ya que si no es así, se puede conseguir el efecto contrario al deseado.

4.4.3. Prueba 3: Diferentes tipos de búsquedas

En Cassandra, al igual que sucede con otros gestores de bases de datos, se pueden realizar búsquedas sobre una tabla de diferentes maneras. Como ya se ha comentado, Cassandra no permite realizar búsquedas encadenadas de varias tablas, pero si se pueden realizar búsquedas por un valor exacto, por rango de valores, que se cumplan varias premisas o algunas (cláusulas `AND` y `OR`), comparando varias columnas, ordenando, filtrando o limitando resultados o utilizando alias, entre otras operaciones.

Estas opciones de búsquedas, la mayoría después de la cláusula `WHERE`, están sujetas a restricciones para así garantizar un buen rendimiento sobre la base de datos.

- **Operaciones condicionales no iguales en la *partition key***. Independientemente de la herramienta en uso de particionado, Cassandra no admite operaciones condicionales que no sean iguales que el contenido de las columnas de la clave de partición. Las operaciones que contienen *consultas por rango* estarían permitidas ya que incluyen esta restricción.
- **Consultar un tabla indexada**. Una consulta en una tabla indexada debe tener al menos una condición de igualdad en la columna indexada. Por ejemplo, si se crea un índice sobre una columna (`LINE`) y se hace una búsqueda en la tabla por esta columna, se puede utilizar el operador de igualdad (`=`) obteniendo resultados.
- **Consultas de rango**. Cassandra soporta las comparaciones "mayor que" y "menor que", pero para una clave de partición dada, las condiciones en la agrupación de la columna están restringidas a los filtros que permite Cassandra para seleccionar un orden contiguo de filas.

Por ejemplo, en la tabla que se está utilizando en este caso práctico, con clave primaria `PRIMARY KEY (policyID, county)` y se quiere hacer una consulta por rango; se podría hacer de la siguiente manera obteniendo un resultado:

```
cqlsh:cassandratfg> SELECT COUNT(*)
                    FROM datos
                    WHERE county = 'CLAY COUNTY'
                    AND policyID >= 119000
                    AND policyID < 119500 ALLOW FILTERING;
```

Si por necesidades del sistema, o por el volumen de datos, se necesita ahorrar el uso de memoria, se puede limitar el número de resultados a la hora de realizar una consulta, con la cláusula `LIMIT`:

```
cqlsh:cassandratfg> SELECT *
                    FROM datos
                    WHERE county = 'CLAY COUNTY'
                    LIMIT 10 ALLOW FILTERING;
```

En este caso, de los 365 resultados que coinciden con la búsqueda, se limita a 10 los valores mostrados al usuario, en orden de aparición en la base de datos.

El uso de la condición `IN` se recomienda en la última columna de la clave **sólo** si, en la consulta, todas las columnas precedentes de la clave tienen la condición de igualdad. Por ejemplo, se modifica la clave primaria de la tabla para que pase a estar formada por las siguientes columnas: `PRIMARY KEY (county, statecode, point_granularity)`. Una consulta que se podría realizar con la cláusula `IN` sería:

```
cqlsh:cassandratfg> SELECT *
                    FROM datos
                    WHERE county = 'CLAY COUNTY'
                    AND statecode = 'FL'
                    AND point_granularity IN (1,3);
```

Si la condición no se cumple, no se podrán realizar consultas usando la cláusula `IN`. Aún así, no se recomienda el uso de esta cláusua después del `WHERE` ya que puede afectar al rendimiento de la consulta en todos los nodos que forman el cluster, porque es muy probable que tenga que consultar en prácticamente todos. Se calcula que en un cluster formado por 30 nodos, con un factor de replicación igual a 3 y un nivel de consistencia `LOCAL_QUORUM`, si se utiliza la cláusua `IN`, el número de nodos que consultaría sería aproximadamente 20, mientras que si no se utiliza, se consultarían de 2 a 6 nodos únicamente, siempre dependiendo de cómo esté formada la clave.

Otra característica de las búsquedas es la comparación de columnas. Desde Cassandra 2.0.6 está disponible esta opción pudiendo agrupar la clave y columnas de agrupación y comparar la tupla de valores para obtener más filas. Por ejemplo:

```
cqlsh:cassandratfg> SELECT *
                    FROM datos
                    WHERE (county, statecode) = ('CLAY COUNTY', 'FL');
```

Por último, entre las características de búsquedas en Cassandra, queda mencionar una muy importante, como es la creación de índices. Cassandra no permite hacer búsquedas sobre columnas que no formen parte de la clave primaria de la tabla, o sobre columnas que no tengan un índice asociado. Esto está diseñado así para garantizar un óptimo rendimiento y una localización "inmediata" de la información que se está buscando. Al igual que se ha dicho que es muy importante definir las columnas que deben formar parte de la clave primaria, es muy importante saber sobre

qué columnas hay que crear un índice ya que esto puede jugar en contra del usuario en cuestiones de optimización. Para crear un índice sobre una columna, basta con:

```
cqlsh:cassandratfg> CREATE INDEX datos_line ON datos (line);
```

De esta manera, si ahora se quiere realizar una búsqueda con una condición sobre esa columna, con la creación de este índice, ya se puede hacer:

```
cqlsh:cassandratfg> SELECT COUNT (*)
                    FROM datos
                    WHERE line = 'Residential';
```

Conclusión: Con esto, se hace un repaso sobre los elementos más importantes para realizar búsquedas en Cassandra sobre una tabla. En el manual de DataStax [32] se recoge por completo todas las formas de consultar sobre la base de datos, con las restricciones y consejos existentes. A pesar de no poder realizar operaciones de búsqueda dentro de otras (*subqueries*), se confirma que con la variedad de tipos existente en Cassandra se pueden realizar búsquedas relativamente complejas en las que obtener información muy precisa.

4.4.4. Prueba 4: Caída de nodos

Una de las preguntas que puede surgir a la hora de ponerse a trabajar con Cassandra es que si la información está repartida entre todos los nodos que forman el cluster, qué pasa con la información que se pueda encontrar si uno o más nodos no están disponibles en ese momento. Pues bien, se va a llevar a cabo la prueba de realizar una consulta sobre la base de datos, de un dato que sabemos que se ha encontrado sobre un nodo, y se va a dejar no disponible para ver si encuentra la información y la devuelve al usuario.

La consulta que se ha realizado en el punto 4.4.2 ha indicado que el dato se ha encontrado en el nodo 192.168.1.4 y una réplica en el 192.168.1.5. Por tanto para este caso de prueba, no se va a iniciar el gestor Cassandra en el nodo 192.168.1.4 quedando sólo habilitados el 192.168.1.3 y 192.168.1.5 (figura 4.7).

Figura 4.7: Prueba de caida de nodos. Nodos habilitados

```
amaquina@amaquina ~/cassandra $ ./bin/nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens  Owns    Host ID                               Rack
UN  192.168.1.5   8,42 MB    256     ?       31eb26ab-632e-4a08-9213-55beb551d477  rack1
DN  192.168.1.4   ?          256     ?       b8ac7cb5-3a51-412b-81f8-667b0a1dcdee  rack1
UN  192.168.1.3   9,28 MB    256     ?       18d66bce-40f7-441d-af89-7e02e0410d49  rack1
```

Se sabe por cuando se hizo la carga masiva en la base de datos, que el número total es de 36.634 registros. Primero se comprueba que el número de elementos sea el mismo con solo dos nodos activos en el cluster. El valor devuelto es el esperado:

```
cqlsh:cassandratfg> SELECT COUNT(*) FROM datos ;

 count
-------
 36636

(1 rows)
```

Por tanto, ahora se prueba a realizar la misma consulta de la que se ha hablado al principio, activando previamente el seguimiento de la base de datos, para ver qué pasos sigue (se omiten las columnas de tiempo y tiempo transcurrido, ya que para el caso que se quiere analizar no aportan mucho):

```
Tracing session: b474e580-dd66-11e4-b5f2-a16cc8128d83

 activity                                                                  | source
---------------------------------------------------------------------------+------------
                                                    Execute CQL3 query | 192.168.1.3
                          Message received from /192.168.1.3 [Thread-5] | 192.168.1.5
                              Executing single-partition query on datos | 192.168.1.5
                                           Acquiring sstable references | 192.168.1.5
                                            Merging memtable tombstones | 192.168.1.5
                                 Bloom filter allows skipping sstable 6 | 192.168.1.5
                           Partition index with 0 entries found for sstable 5 | 192.168.1.5
                                  Seeking to partition beginning in data file | 192.168.1.5
                        Parsing select * from datos WHERE policyid = 172534; | 192.168.1.3
                                                        Preparing statement | 192.168.1.3
                        Sending message to /192.168.1.5 [WRITE-/192.168.1.5] | 192.168.1.3
                                    Executing single-partition query on peers | 192.168.1.3
                                           Acquiring sstable references | 192.168.1.3
                                            Merging memtable tombstones | 192.168.1.3
                         Partition index with 0 entries found for sstable 30 | 192.168.1.3
                                  Seeking to partition beginning in data file | 192.168.1.3
 Skipped 0/1 non-slice-intersecting sstables, included 0 due to tombstones | 192.168.1.3
                                  Merging data from memtables and 1 sstables | 192.168.1.3
                                          Read 0 live and 0 tombstoned cells | 192.168.1.3
 Skipped 0/2 non-slice-intersecting sstables, included 0 due to tombstones | 192.168.1.5
                                  Merging data from memtables and 1 sstables | 192.168.1.5
                                        Read 1 live and 0 tombstoned cells | 192.168.1.5
                        Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 192.168.1.3
```

```
                                    Message received from /192.168.1.3 [Thread-6] | 192.168.1.3
                                        Executing single-partition query on datos | 192.168.1.3
                                             Acquiring sstable references | 192.168.1.3
                                              Merging memtable tombstones | 192.168.1.3
                                   Bloom filter allows skipping sstable 6 | 192.168.1.3
                                              Key cache hit for sstable 5 | 192.168.1.3
                                        Enqueuing response to /192.168.1.3 | 192.168.1.3
                                    Message received from /192.168.1.5 [Thread-4] | 192.168.1.3
                                        Processing response from /192.168.1.5 | 192.168.1.3
                    Sending message to /192.168.1.3 [WRITE-/192.168.1.3] | 192.168.1.3
                                    Message received from /192.168.1.3 [Thread-7] | 192.168.1.3
                                        Processing response from /192.168.1.3 | 192.168.1.3
                                                                Request complete | 192.168.1.3
```

En la columna `source` se ve que solo estan las dos direcciones IP de los dos nodos que estan activos actualmente. Ahora se ve que la ejecución de la consulta la hace en ambos nodos (amarillo) ya que solo ha encontrado a uno de réplica. El dato lo encuentra en el nodo 192.168.1.5 (verde) y envía el mensaje al nodo coordinador para informar de que ha encontrado resultado. El coordinador entonces ejecuta la consulta para unir los datos en la `memtable` (rojo) y dar respuesta al usuario de que la transacción se ha realizado completamente.

Conclusión: Con esto se ve que, aunque alguno de los nodos no esté disponible en ese momento, gracias a la información que se almacena en los ficheros `sstables` físicamente, con el registro de la `memtable` se puede obtener mediante la replicación cualquier dato de la base de datos, manteniendo una alta optimización y una respuesta favorable para el usuario, sin este percartarse que hay menos nodos activos en el cluster.

4.4.5. Prueba 5: Cambio del factor de replicación

Una de las posibilidades con las que un administrador de base de datos se puede encontrar es que si el sistema funciona bien y la información que se almacena crece a un ritmo muy rápido, los nodos lleguen a su capacidad máxima y se vea limitada la base de datos, y por lo tanto su rendimiento. O que por una sobreestimación de espacio, sobre y no haga falta tener determinados nodos para no estar replicando sin necesidad (algo que puede ralentizar las operaciones). Pero gracias a la arquitectura de Cassandra no habría ningún problema a la hora de afrontar cualquiera de los dos escenarios para solventar el problema.

En el caso que se está presentando a lo largo de este capítulo, si se quiere cambiar el factor de replicación del *keyspaces* incluyendo dos nodos más (en total formarían cinco el cluster), bastaría con ejecutar la siguiente sentencia desde el editor de CQL:

```
cqlsh> ALTER KEYSPACE cassandratfg
```

```
WITH REPLICATION =
{'class' : 'NetworkTopologyStrategy', 'amaquina' : 3,
 'bmaquina' : 3, 'cmaquina' : 3,
 'dmaquina' : 3, 'emaquina' : 3};
```

Con esto lo que se hace es incluir los dos nodos nuevos, con el mismo factor de replicación que los otros que ya estaban. Una vez ejecutada esta sentencia, desde fuera del editor de CQL, lo que hay que hacer es "reparar" los nodos para que se actualicen con la información existente (obtener la `memtable`). Para ello, Cassandra ofrece un comando, que con lanzar en consola de comandos `./bin/nodetool repair cassandratfg` en cada uno de los nodos, basta para que todos esten actualizados a la nueva situación. Cuando se realicen inserciones nuevas en la base de datos, esa información se almacenará de manera aleatoria en estos nuevos nodos (siempre suponiendo que la carga está balanceada), hasta que los cinco, en este caso, esten nivelados.

Se debería hacer lo mismo si en vez de cambiar el factor de replicación se cambia la estrategia de replicación; en nuestro de `NetworkTopologyStrategy` a `SimpleStrategy`.

Para el caso de reducir el factor de replicación, la sentencia a ejecutar en el editor CQL tendría la misma estructura, pero esta vez, lo que hay que hacer es "limpiar" el *keyspace*, ejecutando `./bin/nodetool cleanup cassandratfg`. Hay que tener muy en cuenta los casos en los que se reduce el factor de replicación ya que se pueden sobrecargar nodos y perder rendimiento.

Conclusión: Se pueden generalizar estos cambios si se quiere incrementar o decrementar en `n` el factor (o estrategia) de replicación, gracias a la propiedad de escalabilidad flexible, la cual se explica en el punto 3.3.2, siempre y cuando se tengan en cuenta los consejos ya mencionados, como no poner un factor de replicación mayor al número de nodos del cluster, por ejemplo.

4.4.6. Prueba 6: Seguridad

Hoy en día el tema de la seguridad está muy valorado, y más en Internet. A los usuarios les importa cómo viaja la información a través de la red, si puede ser vulnerable, y este fáctor muchas veces hace que se use un determinado producto o no. En bases de datos distribuidas importa mucho saber si la información entre nodos viaja o no encriptada o si se pueden crear permisos, ya que estas características dan muchos puntos de ventaja para ser usadas por otros clientes.

Cassandra ofrece varias implementaciones de seguridad para que la información viaje encriptada, siempre y cuando se configure. Una de ellas es el **cifrado cliente-nodo**, mecanismo que protege los datos "en vuelo" de las máquinas cliente en un clúster de base de datos mediante el protocolo SSL (Secure Sockets Layer) [33]. Se establece un canal seguro entre el cliente y el nodo coordinador (quien ejecuta la operación). Para que esto se pueda llevar a cabo, se deben cumplir previamente dos requisitos:

- Todos los nodos deben tener todos los certificados SSL pertinentes de todos los nodos. En la documentación ofical de DataStax viene como crear dichos certificados [34].
- Para habilitar el cifrado SSL de cliente a nodo, se debe establecer a `enabled` la variable `client_encryption_options` en el fichero de configuración de Cassandra, `cassandra.yaml`.

Un ejemplo de configuración en el fichero sería:

```
client_encryption_options:
  enabled: true
  keystore: conf/.keystore ## Directorio donde guardar el fichero .keystore
  keystore_password: <keystore password> ## Contraseña utilizada cuando
                                         ## se genere el keystore
  truststore: conf/.truststore
  truststore_password: <truststore password>
  require_client_auth: <true or false>
```

Otro mecanismo de seguridad es el cifrado **nodo a nodo**, mecanismo que protege los datos transferidos entre nodos de un clúster mediante SSL (Secure Sockets Layer). Para poder llevar a cabo este mecanismo, hace falta tener implementados los dos requisitos mencionados anteriormente, pero esta vez, la variable del fichero de configuración de Cassandra que hay que activar es `server_encryption_options`. Un ejemplo de configuración en el fichero sería:

```
server_encryption_options:
   internode_encryption: <internode_option>
   keystore: resources/dse/conf/.keystore
   keystore_password: <keystore password>
   truststore: resources/dse/conf/.truststore
   truststore_password: <truststore password>
   require_client_auth: <true or false>
```

De estas dos maneras nos aseguramos que la información que viaja entre nodos, y de cliente a nodo, lo haga de manera encriptada garantizando la seguridad y privacidad de la información que se encuentra en la base de datos.

Ahora bien, si queremos definir cierta seguridad dentro de nuestra base de datos, se pueden crear usuarios y permisos sobre los *keyspaces* y tablas que existan, para garantizar que la información no se altere o borre por cualquiera.

En la base de datos sobre la que se está trabajando, se van a crear tres usuarios, uno con permisos de administrador, otro con permisos de lectura, escritura y borrado sobre la tabla de un *keyspace*, y el último con solo permisos de lectura sobre todos los *keyspaces*. Primero, en el fichero de configuración de Cassandra (`cassandra.yaml`), hay que indicar que la autenticación se realizará por contraseña, modificando la variable correspondiente (`authenticator: PasswordAuthenticator`)

y que las autorizaciones de permisos vendrán de usuarios Cassandra para almacenarlos en el *keyspaces* correspondiente, modificando la variable `authorizer` a `CassandraAuthorizer`. Una vez realizado este paso, sobre el editor de CQL, insertamos las siguientes sentencias (por comodidad, usuario y contraseña serán los mismos, aunque en un caso real nunca debería ser así):

```
cqlsh> CREATE USER daniel WITH PASSWORD 'daniel' SUPERUSER;
cqlsh> CREATE USER carmen WITH PASSWORD 'carmen';
cqlsh> CREATE USER manolo WITH PASSWORD 'manolo';
```

El usuario `daniel` tendrá la capacidad de administrar toda la base de datos con total libertad ya que tiene **todos** los permisos posibles. Es quien puede crear o borrar usuarios (a menos de que a otro usuario se le haya dado este privilegio) por ser *superusuario*. Para asignar los permisos a los otros usuarios, se utiliza el comando `GRANT` (si se quieren quitar permisos, el comando a utilizar es `REVOKE`). El usuario `carmen` será quien tenga los permisos para hacer operaciones de escritura, borrado o actualización sobre la tabla del *keyspace* sobre el que se está trabajando. Para ello, desde el administrador:

```
daniel@cqlsh> GRANT MODIFY ON cassandratfg.datos TO carmen;
```

El usuario `manolo` solo tendrá permisos de lectura sobre todos los *keyspaces*:

```
daniel@cqlsh> GRANT SELECT ON ALL KEYSPACES TO manolo;
```

Si ahora un usuario quiere entrar con sus datos de acceso, desde cualquiera de los nodos, lo haría de la siguiente manera desde un terminal de comandos:

```
$ ./$HOME/cassandra/bin/cqlsh 192.168.1.3 -u USUARIO -p PASSWORD
```

El fichero con los datos de acceso de los usuarios se almacenan en un directorio (`system_auth`), donde la contraseña aparece encriptada. Si este directorio se mueve de la ubicación inicial, los usuarios serán reseteados y solo quedará el *superuser* por defecto.

El usuario `manolo` accede con sus datos a la base de datos. Accede al *keyspace* que hay creado y prueba a hacer una operación de lectura sobre la base de datos. Como tiene permisos, devuelve un resultado.

```
manolo@cqlsh> USE cassandratfg;
manolo@cqlsh:cassandratfg> SELECT county, statecode FROM datos
                           WHERE policyID = 874333;

 county       | statecode
--------------+-----------
  CLAY COUNTY | FL

(1 rows)
```

Si ahora este mismo usuario prueba a hacer una inserción en la tabla, se encuentra que no puede por no tener permisos suficientes. Lo mismo sucedería si quisiera hacer otra operación que no sea `SELECT` sobre cualquier tabla de cualquier *keyspace*:

```
manolo@cqlsh:cassandratfg> INSERT INTO datos (policyID, statecode, county)
                           VALUES (3, 'ES', 'CASTILLA LEON');

code=2100 [Unauthorized] message="User manolo has no MODIFY permission on
<table cassandratfg.datos> or any of its parents"
```

Ahora accede el usuario `carmen` con sus datos a la base de datos. Accede al *keyspace* que hay creado y prueba a hacer las mismas operaciones (buscar e insertar) que ha hecho el usuario `manolo`. Se observa ahora `carmen` no tiene permisos para hacer búsquedas en la tabla `cassandratfg.datos`, pero si puede insertar datos sobre esta misma, tal y como se había contemplado.

```
carmen@cqlsh> USE cassandratfg;
carmen@cqlsh:cassandratfg> SELECT county, statecode FROM datos
                           WHERE policyID = 874333;

code=2100 [Unauthorized] message="User carmen has no SELECT permission on
<table cassandratfg.datos> or any of its parents"
```

Sobre el mismo *keyspace*, se va a probar si `carmen` puede insertar datos sobre otra tabla existente:

```
carmen@cqlsh:cassandratfg> INSERT INTO datostfg (id, datos)
                           VALUES (1, 'Cassandra base de datos');

code=2100 [Unauthorized] message="User carmen has no MODIFY permission on
<table cassandratfg.datostfg> or any of its parents"
```

La respuesta aquí es distinta, ya que no puede insertar datos en otra tabla que no sea `datos` dentro del *keyspace* `cassandratfg`. Para confirmar que todo funciona según los permisos asignados a este usuario, `carmen` prueba a insertar datos en otra tabla de otro *keyspace* diferente que hay en la base de datos:

```
carmen@cqlsh:cassandratfg> USE prueba;
carmen@cqlsh:prueba> INSERT INTO datosprueba (id, column1, edad)
                     VALUES (1, 'Value1', 23);

code=2100 [Unauthorized] message="User carmen has no MODIFY permission on
<table prueba.datosprueba> or any of its parents"
```

Se confirman las sospechas de que no puede insertar datos (o borrar o modificar) en otra tabla de cualquier lugar de la base de datos que no sea `datos` y se encuentre en el *keyspace* `cassandratfg`.

Mucho menos ninguno de los dos usuarios (`carmen` y `manolo`) podrá realizar operaciones de asignación o cambio de permisos (incluso sobre sí mismos), crear, modificar o borrar tablas, o crear o borrar índices sobre ellas. En la tabla 3.4 del capítulo 3 se reflejan los permisos en CQL que se pueden asignar, y lo que se puede hacer con cada uno de ellos.

Si un administrador de base de datos quiere saber los permisos que hay sobre un *keyspace* o una tabla, o los que tiene un usuario, puede hacerlo desde el editor CQL con la sentencia `LIST permission_name PERMISSION ON resource OF user_name`. Por ejemplo, si el administrador `daniel` quiere saber los permisos que hay sobre la tabla `cassandratfg.datos`, haría:

```
daniel@cqlsh> LIST ALL PERMISSIONS ON cassandratfg.datos ;

 username | resource                        | permission
----------+---------------------------------+------------
   carmen | <table cassandratfg.datos> |     MODIFY
   manolo |            <all keyspaces> |     SELECT

(2 rows)
```

Conclusión: Con este análisis de seguridad se ha dado una visión global de cómo se puede proteger la información contenida en la base de datos, y como se pueden dar permisos a usuarios según un rol para que así se pueda operar sobre ella con la certeza por parte del administrador de que no se va a vulnerar la información con acciones no permitidas.

Conclusiones y Trabajo Futuro

Conclusiones

Como resultado del estudio de investigación presentado, se pueden sacar varias conclusiones sobre el funcionamiento y la arquitectura del gestor de bases de datos Cassandra, cumpliendo así los objetivos iniciales de este trabajo fin de grado presentados en el punto 1.1, que eran el de adentrarse en esta nueva tecnología emergente en el mundo de las bases de datos, ver su potencial frente a otras y servir de introducción para futuros estudios. Ha quedado claro que Cassandra no es un sustituto de las bases de datos relaciones, pero que su utilización por grandes empresas que ofrecen gran variedad de contenidos al público está creciendo cada vez más por los resultados de rendimiento y tiempo de respuesta que oferta, algo que para los usuarios es muy importante.

Que Cassandra almacene la información en columnas es un concepto difícil de imaginar y manejar cuando uno esta acostumbrado al concepto de almacenamiento en filas de las bases de datos tradicionales. Esta organización de la información hace que la recuperación sea mucho más rápida ya que lo que busca es una clave que identifica la columna para obtener el valor buscado. Las operaciones por segundo en las búsquedas se ven reducidas notablemente, de ahí la importancia de destacar Cassandra como una base de datos basada en columnas clave-valor.

La otra característica importante que identifica a Cassandra es la distribución de la información, algo muy interesante que ayuda a no concentrar todos los elementos de la base de datos en un único punto, pudiendo así repartir la carga de los servidores que alojan la base de datos y poder aprovechar más los recursos de cada uno, sin preocuparse de la disponibilidad de la información en caso de que algún nodo no esté disponible gracias a la replicación y garantizando la seguridad en el viaje de la información por la red gracias a la encriptación que se puede configurar.

Mediante el caso práctico, se ha podido comprender el funcionamiento que hay por debajo al realizar operaciones sobre la base de datos implementada; el balanceo de carga en los nodos, ver a qué nodos se llama, dónde guarda la información, o cómo garantiza la consistencia, e incluso ver como no siempre se puede obtener un buen rendimiento por crear índices, algo que en principio puede suponer siempre una mejora.

A su vez, se pretendía que este estudio se presentara de forma clara y concisa, de manera que pueda ser utilizado en la docencia con el objetivo de que el alumno cuente también con una ligera

visión de lo que son las bases de datos no relaciones y cómo funcionan, y una visión más detallada de cómo es Cassandra de manera global.

Trabajo Futuro

Este proyecto ha servido para hacer una descripción completa de Cassandra, con el objetivo de servir de introducción a la tecnología y poder continuar realizando otros estudios partiendo con la base que aquí se ha presentado. Evidentemente, al ser un modelo de base de datos joven, hay muchas ramas por las que seguir investigando o para implementar, pero quizás las investigaciones que más interesa seguir desde este punto son: cómo diseñar un modelo de datos en Cassandra partiendo de un modelo de datos relacional. Este podría ser un nuevo proyecto por su dimensión ya que, como Cassandra no es una base de datos en la que se puedan relacionar tablas, habría que ver de qué manera y con qué mecanismos se puede construir una base de datos partiendo de una relacional. Otro punto por el que se puede seguir sería el estudio completo de la interfaz CQL, ya que está actualmente en continuo desarrollo con mejores de implementación y haciendo que las operaciones obtengan un rendimiento cada vez más alto.

Estas dos líneas serían muy interesantes para nuevos proyectos en los que se podría formar, juntando todos ellos, un estudio exhaustivo de Cassandra, sirviendo incluso a muchos para llevar a cabo tareas con esta base de datos.

Referencias bibliográficas

[1] Big data - Wikipedia en inglés. [En línea]. http://en.wikipedia.org/wiki/Big_data. (Última consulta: 17/12/2014). 1.1, 3.1

[2] Tablas de hash distribuidas - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Tabla_de_hash_distribuida. (Última consulta: 17/12/2014). 2.1

[3] Christof Strauch. NoSQL Databases. *Stuttgart Media University*, 2011. 2.1

[4] Bases de datos NoSQL. Qué son y tipos que nos podemos encontrar. [En línea]. http://www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf. (Última consulta: 11/12/2014).

[5] NoSQL - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/NoSQL. (Última consulta: 26/11/2014).

[6] Teoría de grafos - Wikipedia. [En línea]. http://goo.gl/7pUt. (Última consulta: 10/12/2014). 2.3.3

[7] Jonathan Ellis. NoSQL Ecosystem. [En línea]. http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/, 09/11/2009. (Última consulta: 15/12/2014). 2.2

[8] Cassandra - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Apache_Cassandra. (Última consulta: 17/12/2014). 3.1

[9] P2P - Wikipedia en inglés. [En línea]. http://en.wikipedia.org/wiki/Peer-to-peer. (Última consulta: 17/12/2014). 3.1

[10] Twitter. [En línea]. http://www.twitter.es. 3.1

[11] SSH (Secure SHell) - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Secure_Shell. (Última consulta: 23/12/2014). 3.2

[12] Descarga de Apache Cassandra. [En línea]. http://cassandra.apache.org/download/. (Última consulta: 18/12/2014). 1

[13] Web de Oracle. [En línea]. http://www.oracle.com. (Última consulta: 18/12/2014). 8

[14] Arquitectura de SSTables - [En línea]. http://wiki.apache.org/cassandra/ArchitectureSSTable. (Última consulta: 06/04/2015). 3.3.7, 4.4.2

[15] Thrift API. [En línea]. https://wiki.apache.org/cassandra/API10. (Última consulta: 23/12/2014). 3.5

[16] CQL (Cassandra Query Language). [En línea]. https://cassandra.apache.org/doc/cql/CQL.html. (Última consulta: 20/12/2014). 3.5, 3.6

[17] Protocolo de comunicación. - Wikipedia [En línea]. http://es.wikipedia.org/wiki/Protocolo_de_comunicaciones. (Última consulta: 30/03/2015). 3.5

[18] Remote Procedure Call. - Wikipedia en inglés. [En línea]. http://en.wikipedia.org/wiki/Remote_procedure_call. (Última consulta: 30/03/2015). 3.5

[19] Cassandra CLI. [En línea]. http://wiki.apache.org/cassandra/CassandraCli. (Última consulta: 02/03/2015). 3.5

[20] CQL vs. Thrift: Operations. [En línea]. http://www.datastax.com/dev/blog/cassandra-2-1-now-over-50-faster. (Última consulta: 06/04/2015). 3.5

[21] DataStax documentation. [En línea]. http://www.datastax.com/documentation/cql/3.1/cql/cql_intro_c.html. (Última consulta: 22/12/2014). 3.6, C

[22] CQL3. [En línea]. http://cassandra.apache.org/doc/cql3/CQL.html. (Última consulta: 23/12/2014).

[23] Descarga driver CQL3 para Linux. [En línea]. http://downloads.datastax.com/java-driver/cassandra-java-driver-2.0.2.tar.gz. (Última consulta: 23/12/2014). 3.7

[24] UUID (Universally Unique IDentifier) - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Universally_unique_identifier. (Última consulta: 23/12/2014). 3.7

[25] Snapshot - Wikipedia en inglés. [En línea]. http://en.wikipedia.org/wiki/Snapshot_(computer_storage). (Última consulta: 20/01/2015).

[26] Time series data (Series temporales de datos). [En línea]. http://planetcassandra.org/getting-started-with-time-series-data-modeling. (Última consulta: 02/03/2015). 3.8

[27] Comandos del editor CQL. [En línea]. http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cqlCommandsTOC.html. (Última consulta: 09/03/2015). 3.9

[28] Administrador de base de datos - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Administrador_de_base_de_datos. (Última consulta: 23/12/2014).

[29] Web de la descarga de datos. [En línea]. https://www.census.gov/econ/cbp/download. (Última consulta: 23/03/2015). 4.1

[30] Commitlog - [En línea]. https://wiki.apache.org/cassandra/Durability. (Última consulta: 06/04/2015).

[31] Memtable - [En línea]. https://wiki.apache.org/cassandra/MemtableSSTable. (Última consulta: 06/04/2015).

[32] Operaciones en búsquedas CQL. [En línea]. http://docs.datastax.com/en/cql/3.1/cql/cql_reference/select_r.html. (Última consulta: 13/04/2015). 4.4.3

[33] SSL, Secure Sockets Layer - Wikipedia. [En línea]. http://es.wikipedia.org/wiki/Transport_Layer_Security. (Última consulta: 28/03/2015). 4.4.6

[34] Preparación de los certificados SSL. [En línea]. http://docs.datastax.com/en/cassandra/1.2/cassandra/security/secureSSLCertificates_t.html. (Última consulta: 28/03/2015). 4.4.6

Apéndice A

Contenido del CD

En el CD-ROM que se adjunta se ha incluido parte del material que se ha utilizado para el correcto desarrollo de este trabajo fin de grado. A continuación, se detalla el árbol de directorios:

- **memoria.pdf**: memoria del trabajo fin de grado desarrollado.
- **Material adicional**: carpeta que contiene los ficheros adicionales utilizados para el desarrollo del trabajo fin de grado.
  - **planificacionTFG.mpp**: fichero elaborado con Microsoft Project 2013 en el que se recoge la planificación temporal del trabajo fin de grado.
  - **tokengentool.py**: script en Python utilizado para la obtención de los número simbólicos necesarios en la configuración del entorno Cassandra.
  - **createtable.txt**: script de creación de la tabla que se ha usado para cargar los datos.
  - **cargaStates.csv**: fichero que contiene los datos para la importación en la tabla creada en la base de datos.

Apéndice B

Documentación CQL para Cassandra 2.x

B.1. Tipos de datos CQL

A continuación, se muestra los distintos tipos de datos que podemos usar en la creación de columnas en Cassandra bajo el entorno CQL.

Tipo en CQL	Constantes	Descripción
`ascii`	strings	Cadena de caracteres US-ASCII.
`bigint`	integers	64 bits de longitud con signo.
`blob`	blobs	Bytes arbitrarios (sin validación), expresados en hexadecimal.
`boolean`	booleans	`true` o `false`.
`counter`	integers	Valor del contador distribuido (64 bits de longitud).
`decimal`	integers, floats	Decimal de precisión variable. Tipo Java. **Nota**: Cuando se trata de la moneda, es una buena práctica tener una clase de moneda que serializa desde y hacia un int o utilice el formulario Decimal.
`double`	integers, floats	De punto flotante de 64 bits IEEE-754. Tipo Java.
`float`	integers, floats	De punto flotante de 32 bits IEEE-754. Tipo Java.
`inet`	strings	Cadena de dirección IP en formato IPv4 o IPv6, utilizado por el conductor python-CQL y protocolos nativos CQL.
`int`	integers	32 bits enteros con signo.
`list`	n/a	Una colección de uno o más elementos ordenados.
`map`	n/a	Una matriz de estilo JSON de literales: literal: literal, literal: literal, ...
`set`	n/a	Una colección de uno o más elementos.

Sigue en la página siguiente.

text	strings	Cadena UTF-8 codificado.
timestamp	integers, strings	Fecha más el tiempo, codificado como 8 bytes.
timeuuid	uuids	Solo tipo 1 UUID.
tuple	n/a	Cassandra 2.1 y versiones posteriores. Un grupo de 2-3 campos.
uuid	uuids	Un UUID en formato estándar UUID.
varchar	strings	Cadena UTF-8 codificado.
varint	integers	Número entero de precisión arbitraria. Tipo Java.

Tabla B.1: Tipos de datos CQL

Fuente: www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_data_types_c.html

B.2. Opciones del comando COPY - CQL

En la siguiente tabla se muestran las distintas opciones que se pueden usar en el comando COPY para tratar un fichero CSV en la importación/exportación.

COPY Options	Valor predeterminado	Uso
DELIMITER	coma (,)	Establece el carácter que separa los campos que tienen caracteres de nueva línea en el archivo.
QUOTE	comillas (")	Establece el carácter que encierra el valor de un campo.
ESCAPE	barra invertida (\)	Establece el carácter que escapa usos literales del carácter QUOTE.
HEADER	true/false	Establece true para indicar que primera fila del archivo es un encabezado.
ENCODING	UTF8	Ajuste el comando COPY TO a cadenas UNICODE de salida.
NULL	cadena vacía	Representa la ausencia de un valor.

Tabla B.2: Opciones comando COPY

Fuente: www.datastax.com/documentation/cql/3.0/cql/cql_reference/copy_r.html

La opción de codificación (`ENCODING`) no puede utilizarse en la importación (`COPY FROM`). Esta tabla muestra que, por defecto, Cassandra espera que los datos CSV a tratar estén en campos separados por comas (,), registros separados por separadores de línea (una línea nueva, \r \n), y los valores de campo encerrados en comillas dobles (“ ”). También, para evitar la ambigüedad, escapar de una doble comilla usando una barra invertida dentro de una cadena encerrada en comillas dobles (“\” ”). De forma predeterminada, Cassandra no espera que el archivo CSV tenga encabezado con los nombres de columna en la primera línea. `COPY TO` (exportar) incluye el encabezado en la salida si `HEADER=true`. `COPY FROM` (importar) ignora la primera línea si `HEADER=true`.

Apéndice C

Script en Python para el cálculo del número simbólico

Script, en lenguaje Python, proporcionado por DataStax [21], para la obtención del número simbólico de cada nodo, en Cassandra, para éste posicionarse en el lugar correcto del espacio de almacenamiento.

```
#! /usr/bin/python

import copy
import json
import sys

global_data = {}
global_data['MAXRANGE'] = (2**127)

def calculate_tokens():
    """Sets the default tokens that each datacenter has to be spaced with."""

    tokens = {}
    for dc in range(len(global_data['datacenters'])):
        tokens[dc] = {}

        for i in range(int(global_data['datacenters'][dc])):
            tokens[dc][i] = (i * global_data['MAXRANGE'] /
                int(global_data['datacenters'][dc]))

    global_data['tokens'] = tokens

def two_closest_tokens(this_dc, this_token):
    """Returns the two closests tokens to this token within the entire
       cluster."""

```

```
25      tokens = get_offset_tokens()
26      lower_bound = 0
27      upper_bound = global_data['MAXRANGE']
28
29      for that_dc in tokens:
30          if this_dc == that_dc:
31              # Don't take the current datacenter's nodes into consideration
32              continue
33
34          that_dc = tokens[that_dc]
35
36          for that_node in that_dc:
37              that_token = that_dc[that_node]
38
39              if that_token <= this_token and that_token > lower_bound:
40                  lower_bound = that_token
41              if that_token > this_token and that_token < upper_bound:
42                  upper_bound = that_token
43
44      return lower_bound, upper_bound
45
46  def get_offset_tokens():
47      """Calculates what the tokens are with their calculated offsets."""
48
49      offset_tokens = copy.deepcopy(global_data['tokens'])
50      for dc in offset_tokens:
51          if dc == 0:
52              # Never offset the first datacenter
53              continue
54
55          # Apply all offsets
56          for node in offset_tokens[dc]:
57              offset = global_data['offsets'][dc] if dc in
                    global_data['offsets'] else 0
58              offset_tokens[dc][node] = (offset_tokens[dc][node] + offset) %
                    global_data['MAXRANGE']
59      return offset_tokens
60
61  def calculate_offsets():
62      """Find what the offsets should be for each datacenter."""
63
64      exit_loop = False
65      while not exit_loop:
66          exit_loop = True
67
68          tokens = get_offset_tokens()
69          for this_dc in range(len(tokens)):
70              if this_dc == 0:
```

```
71                  # Never offset the first datacenter
72                  continue
73
74              tokens = get_offset_tokens()
75              global_data['offsets'][this_dc] = global_data['offsets'][this_dc]
                    if this_dc in global_data['offsets'] else 0
76              previous_offset = global_data['offsets'][this_dc]
77              running_offset = []
78
79              # Get all the offsets, per token, that place each token in the
                    ideal spot
80              # away from all other tokens in the cluster
81              for this_node in range(len(tokens[this_dc])):
82                  this_token = tokens[this_dc][this_node]
83                  lower_bound, upper_bound = two_closest_tokens(this_dc,
                        this_token)
84                  perfect_spot = (upper_bound - lower_bound) / 2 + lower_bound
85                  this_offset = perfect_spot - this_token
86                  running_offset.append(this_offset)
87
88              # Set this datacenters offset to be an average of all the running
                    offsets
89              if len(running_offset):
90                  global_data['offsets'][this_dc] += sum(running_offset) /
                        len(running_offset)
91
92              # Vote on exiting the loop if this datacenter did not change it's
                    offset
93              if global_data['offsets'][this_dc] - previous_offset > 0:
94                  exit_loop = False
95
96  # ============================
97  # Main Starters
98
99  def print_tokens(tokens=False):
100     if not tokens:
101         tokens = get_offset_tokens()
102     # print 'Offsets: ', global_data['offsets']
103     print json.dumps(tokens, sort_keys=True, indent=4)
104
105     if 'test' in global_data:
106         calc_tests(tokens)
107
108 def run(datacenters):
109     global_data['offsets'] = {}
110
111     # Calculate the amount of datacenters in the beginning
112     # of the list that have no nodes
```

```
113     # Because the first DC remains stable
114     leading_blank_centers = 0
115     for datacenter in datacenters:
116         if not datacenter:
117             leading_blank_centers += 1
118         else:
119             break
120     datacenters = datacenters[leading_blank_centers:]
121 
122     global_data['datacenters'] = datacenters
123     calculate_tokens()
124     calculate_offsets()
125     returning_tokens = get_offset_tokens()
126 
127     # Add the preceding blank datacenters back in
128     if leading_blank_centers:
129         translated_tokens = {}
130         for i in range(leading_blank_centers):
131             translated_tokens[i] = {}
132         i += 1
133         for j in range(len(returning_tokens.keys())):
134             translated_tokens[i] = returning_tokens[j]
135             i += 1
136         returning_tokens = translated_tokens
137 
138     # print returning_tokens
139     return returning_tokens
140 
141 # ============================
142 
143 # ============================
144 # Tests
145 
146 def calc_tests(tokens):
147     import math
148     these_calcs = {}
149 
150     for this_dc in range(len(tokens)):
151         these_calcs[this_dc] = []
152         for node in range(len(tokens[this_dc])):
153             degrees = ((tokens[this_dc][node]) * 360 /
                    global_data['MAXRANGE']) + 180
154             radians = degrees * math.pi / 180
155 
156             center = global_data['graph_size'];
157             x2 = center + global_data['length_of_line'] * math.sin(radians);
158             y2 = center + global_data['length_of_line'] * math.cos(radians);
159             these_calcs[this_dc].append((x2, y2))
```

```
160
161     global_data['coordinates'].append(these_calcs)
162
163 def write_html():
164     html = """<!DOCTYPE html>
165 <html>
166 <body>
167
168 %s
169
170 </body>
171 </html>
172
173 """
174     default_chart = """
175     <canvas id="{0}" width="{2}" height="{2}" style="border:1px solid
        #c3c3c3;">
176         Your browser does not support the canvas element.
177     </canvas>
178     <script type="text/javascript">
179         var c=document.getElementById("{0}");
180         var ctx=c.getContext("2d");\n%s
181     </script>
182     """
183     default_chart_piece = """
184         ctx.beginPath();
185         ctx.strokeStyle = "%s";
186         ctx.moveTo({1},{1});
187         ctx.lineTo(%s,%s);
188         ctx.stroke();
189         ctx.closePath();
190     """
191     all_charts = ''
192     for chart_set in range(len(global_data['coordinates'])):
193         chart_index = chart_set
194         chart_set = global_data['coordinates'][chart_set]
195         chart_piece = ''
196         for dc in range(len(chart_set)):
197             for coordinates in chart_set[dc]:
198                 chart_piece += default_chart_piece %
                    (global_data['colors'][dc], coordinates[0], coordinates[1])
199         this_chart = default_chart % chart_piece
200         all_charts += this_chart.format(chart_index,
            global_data['graph_size'], global_data['graph_size'] * 2)
201     with open('tokentool.html', 'w') as f:
202         f.write(html % all_charts)
203
204 def run_tests():
```

```
205     global_data['test'] = True
206     global_data['coordinates'] = []
207     global_data['graph_size'] = 100
208     global_data['length_of_line'] = 80
209     global_data['colors'] = ['#000', '#00F', '#0F0', '#F00', '#0FF', '#FF0',
            '#F0F']
210     global_data['MAXRANGE'] = 1000
211 
212     tests = [
213         [1],
214         [1, 1],
215         [2, 2],
216         [1, 2, 2],
217         [2, 2, 2],
218         [2, 0, 0],
219         [0, 2, 0],
220         [0, 0, 2],
221         [2, 2, 0],
222         [2, 0, 2],
223         [0, 2, 2],
224         [0, 0, 1, 1, 0, 1, 1],
225         [6],
226         [3, 3, 3],
227         [9],
228         [1,1,1,1],
229         [4],
230         [3,3,6,4,2]
231     ]
232     for test in tests:
233         print_tokens(run(test))
234     write_html()
235 
236 # ============================
237 
238 if __name__ == '__main__':
239     if len(sys.argv) > 1:
240         if sys.argv[1] == '--test':
241             run_tests()
242             sys.exit(0)
243         datacenters = sys.argv[1:]
244     else:
245         print "Usage: ./tokentoolv2.py <nodes_in_dc> [<nodes_in_dc>]..."
246         sys.exit(0)
247     run(datacenters)
248     print_tokens()
```