MECANISMOS DE PLANIFICACIÓN DE ACCESOS A BASES DE DATOS NO RELACIONALES

Autor: David González Arriaza

Titulación: Grado en Ingeniería Informática

Especialidad: Tecnologías de la Información

Directora: Yolanda Becerra

Departamento: Arquitectura de Computadores (AC)

Fecha: 05/02/2014

Facultad de Informática de Barcelona (FIB)

Universidad Politécnica de Cataluña (UPC) - BarcelonaTech

Resumen

Como objetivo principal de éste trabajo, se ha implementado un módulo, para optimizar de forma automática las lecturas sobre la base de datos Apache Cassandra. Él módulo puede ser configurado a través de unos parámetros de rendimiento, lo cual ha permitido la realización de distintas pruebas en función de dichos parámetros. Los resultados que se han obtenido como consecuencia de las pruebas, han sido los esperados, puesto que se ha conseguido realizar peticiones a nodos específicos en función de los datos requeridos, que dependiendo de la topología de la red y la carga de ésta, con éste módulo, se podrían conseguir resultados muy beneficiosos. Además, se ha realizado una investigación sobre qué son las bases de datos NoSQL y en concreto, cómo funciona la base de datos Apache Cassandra, que es sobre la que se ha trabajado y sobre el query plan en las bases de datos relacionales.

Resum

L'objectiu principal d'aquest treball ha sigut realitzar la implementació d'un mòdul, per tal de poder optimitzar les lectures a la base de dades Apache Cassandra. El mòdul pot ser configurat mitjançant uns paràmetres de rendiment, els quals han permès la realització de diferents proves. Els resultats obtinguts a les proves han sigut els esperats, ja que s'ha aconseguit realitzar peticions a nodes específics en funció de les dades a consultar. Aquesta funcionalitat podria donar resultats molt avantatjosos per a casos en els quals es tingués una xarxa molt sobrecarregada. A més, s'ha realitzat un estudi sobre què son les bases de dades de tipus NoSQL i en concret, com funciona la base de dades Apache Cassandra, que és sobre la que s'ha treballat i també s’ha fet un estudi sobre el query plan a les bases de dades de tipus relacional.

Abstract

The main goal of the project has been the implementation of a module to automatically optimize the data queries that obtain reads from the Apache Cassandra database. The module can be configured with some performance parameters, which have allowed doing some tests. The results of the tests have been as it was expected. It has been seen that the queries were addressed to the nodes that were physically storing the data. The fact to be able to guide the queries to the responsible nodes would give profitable results in cases where the network is overloaded. Moreover, a study of what are NoSQL databases has been done, specifically how Apache Cassandra works because is the database used in this project. In addition, the query plan on relational databases has also been studied.

Índice de contenido

Índice de figuras

Índice de tablas

1. Introducción

El objetivo principal del proyecto es definir, diseñar e implementar un mecanismo que permita optimizar de manera automática la ejecución de accesos complejos a Apache Cassandra y medir el rendimiento de las ejecuciones realizadas.

Esto se hará mediante la implementación de un módulo intermedio entre la consulta y la base de datos, el cual recibirá las peticiones y automáticamente, dividirá la consulta según convenga para obtener la mayor eficiencia posible.

Además, se definirán e implementarán políticas sencillas que permitan evaluar el mecanismo en función de diferentes parámetros de rendimiento.

Éste mecanismo será beneficioso tanto para la persona que desee realizar una consulta de muchos datos, ya que tan solo deberá especificar cuál es la información que desea obtener y el mecanismo

automáticamente recuperará los datos, como para el investigador que desee comprobar qué consecuencias tiene en la ejecución, una política con una configuración de parámetros de rendimiento determinados.

1.1 Problema

Hoy en día la cantidad de información que circula y se almacena por la red crece casi de manera exponencial y los datos están más desestructurados, de ahí que surja la necesidad de encontrar nuevas formas de almacenar estos datos.

Hasta hace poco se solía hacer uso de bases de datos relacionales (SQL), que como bien indica su nombre, son bases de datos en las cuales sus tablas o schemas pueden estar relacionados entre sí. Este tipo de base datos requieren de una previa

creación de las tablas con sus correspondientes campos, a los cuales se les debe preasignar un tamaño, con el consecuente problema de que los datos después, pueden ser de tamaño variable y se esté perdiendo espacio. Por lo tanto, esto hace que las bases de datos SQL no sean adecuadas para datos que están desestructurados.

Además, la búsqueda de datos para las bases de datos de tipo SQL es bastante compleja, dado que la información por lo general suele estar distribuida en múltiples tablas interrelacionadas. Es por ello, por lo que se han dedicado tantos esfuerzos en desarrollar optimizadores que procesen la consulta y que la realicen de la manera más eficiente posible. A causa de la complejidad de búsqueda, las bases de datos SQL no son adecuadas para trabajar con grandes volúmenes de información, ya que las relaciones complejas dificultan la escalabilidad.

En el caso de las bases de datos NoSQL, las consultas son más sencillas, ya que los datos no

están relacionados como en el caso de SQL y aún no se ha visto la necesidad de optimizar los accesos.

Las bases de datos no-relacionales han sido diseñadas para gestionar de manera eficiente los accesos a la información de aplicaciones BigData, que suelen implementar muchos accesos pero requiriendo poca información cada uno de ellos. Sin embargo, el uso de esta tecnología se está extendiendo para servir también a aplicaciones que obtienen una gran cantidad de información en cada acceso. En este caso, el rendimiento de las aplicaciones puede ser muy diferente en función de cómo se organicen los accesos: ¿Es mejor hacer una sola petición que cargue toda la información? ¿Es mejor hacer varias? ¿Qué características del cluster hay que tener en cuenta?

Con la implementación del módulo se facilitará la ejecución de las pruebas necesarias para responder a estas preguntas.

1.2 Tareas

- **Estudio inicial:** esta tarea consistirá en hacer un estudio de cómo funciona la base de datos NoSQL Apache Cassandra. Se estudiará la manera en que esta base de datos almacena la información, qué mecanismos de acceso proporciona, el lenguaje y la sintaxis que se utiliza para consultar y almacenar datos en ella.

- **Diseño e implementación del módulo:** esta parte del proyecto se realizará una vez haya finalizado el estudio previo y se hayan adquirido unos conocimientos mínimos sobre Apache Cassandra. Consistirá en realizar el diseño y la implementación del módulo, que permita optimizar de forma automática el acceso a la base de datos.

El módulo recibirá la petición para la base de datos y automáticamente deberá partir esta petición, en función a los parámetros de rendimiento configurables, y pedir los datos a los nodos correspondientes, con la finalidad de hacerlo lo más eficiente posible.

- **Implementación de políticas:** una vez se haya acabado con la implementación del módulo se definirán e implementarán políticas sencillas que permitan evaluar el mecanismo en función de diferentes métricas de rendimiento.

- **Pruebas finales:** a medida que se van implementando las cosas, se irán haciendo pruebas que permitan observar que lo que se ha hecho funciona. Cuando todo esté ya finalmente implementado se harán pruebas que permitan ver los resultados que se

obtienen a causa de utilizar el módulo intermedio.

1.2.1 Gantt

Figura 1: Diagrama de Gantt

2. Estado del arte

En la siguiente sección se hablará sobre Apache Cassandra, que es la base de datos NoSQL sobre la que se ha decidido trabajar para realizar las investigaciones. Se ha hecho un estudio sobre cuál es la forma en la que almacena la información y sobre algunas de las características que tiene ésta base de datos.

Además, se ha hecho un breve estudio sobre el Query Plan en las bases de datos de tipo relacional, para saber cuáles son los pasos que realiza una petición, desde que la hace el usuario hasta que le llega el resultado.

2.1 NoSQL

NoSQL (Not only SQL) es un tipo de base de datos que rompe con los tradicionales patrones a los que nos tenían acostumbrados las bases de datos de tipo relacional.

Este tipo de base de datos es capaz de almacenar grandes volúmenes de información sin requerir una estructura fija para su guardado. Además, no hace uso de SQL que es el lenguaje usado por las BD relacionales.

Dentro del ámbito del NoSQL se puede encontrar un amplio abanico de bases de datos de este tipo, pero que difieren en la forma en la que almacenan los datos. Algunos ejemplos de guardado de datos son:

- Clave-valor
    - Permiten almacenar los datos sin la necesidad de definir un schema. Para una fila se puede almacenar un valor, con el valor que se desee.

Tan sólo se debe definir qué tipo de datos se almacenará.

- Documentales
  - Este tipo de base de datos son capaces de almacenar datos en formatos como XML, YAML, JSON y BSON, PDF entre otros. Son similares a las de clave-valor.

- Grafo
  - Este tipo de bases de datos funcionan muy bien para datos que están interrelacionados, formando de esta manera grafos.

NoSQL suele tener la habilidad de escalar horizontalmente con el aumento de la cantidad de datos, gracias a una buena distribución de los datos. Esta distribución a su tiempo hace que la información esté replicada y se minimicen los puntos de fallo.

2.2 Apache Cassandra

2.2.1 ¿Qué es?

Apache Cassandra es una base de datos de tipo NoSQL. Fue diseñada para soportar grandes cantidades de datos distribuidos por varios servidores, proporcionando de esta manera alta disponibilidad de los datos sin ningún punto fijo de fallo. [1]

Esta base de datos nació como una fusión del modelo de datos BigTable de Google y la arquitectura de Amazon Dynamo.

Los datos se almacenan por clave-valor y la consistencia de los mismos se puede ajustar. Hay un parámetro de replicación que el administrador puede modificar para elegir en cuantos nodos quiere que la información este replicada.

Es una base de datos descentralizada, es decir, todos los nodos del cluster tienen un mismo rol. De ahí que se diga que no tiene ningún punto fijo de fallo. La información está distribuida por todos los nodos que forman el cluster pero no hay ningún nodo master ya que cualquiera de ellos puede satisfacer cualquier petición. [3]

Los datos se replican automáticamente entre los nodos para evitar tiempos en fuera de servicio a causa de la caída de alguno de los servidores.

Se puede tener la información distribuida desde en un solo nodo, hasta en múltiples nodos en distintos DataCenters.

Además, Cassandra incorpora un lenguaje, CQL, a través del cual permite hacer consultas de los datos de una forma muy similar a SQL.

2.2.2 Protocolo Gossip

Los nodos que forman el cluster utilizan un protocolo interno llamado “gossip”. Este protocolo es de tipo peer-to-peer y permite saber a cada nodo la información y el estado de los demás nodos del cluster.

Dicha información se la van intercambiando periódicamente(cada segundo), dónde se define la información tanto del propio nodo cómo la de los nodos que éste conoce.

2.2.3 Características

- **Particionador:** es el encargado de decidir cómo se distribuyen los datos a través de los nodos. Es una función de hash, la cual calcula el token para una clave y según el resultado generado, le

asigna esa clave a un nodo o a otro. La asignación dependerá también de los rangos de tokens que tenga asignado cada nodo.

Algunos de los particionadores que incorpora la base de datos son:

- Murmur3: es un particionador que distribuye uniformemente los datos basado en un valor hash Murmur(no criptográfico). Crea un valor hash de hasta 128 bits dada una clave.
- Random: distribuye los datos basando en valores hash MD5 comprendidos entre 0 y 2^127.
- Por byte: ordena lexicográficamente, por los bytes de las filas.

- **Factor de replica:** permite especificar sobre cuántos nodos se quiere tener la información replicada. Dos estrategias de replicación:
  - Simple: para un sólo DataCenter. Se almacena

la primera réplica en un nodo elegido por el particionador. Las demás se almacenan en el sentido de las agujas del reloj.

- Red: para cuando la información está distribuida en varios DataCenter. Permite elegir cuantas replicas se desean en cada centro.

- **Snitch:** tiene información sobre la topología del cluster y se encarga de decidir el mejor camino para una petición. Los tipos de snitch son:
  - Simple: no es capaz de reconocer información sobre racks ni de DataCenters(DC). Es apropiado sólo si se trabaja en un único DC.
  - RackInferring: es capaz de determinar la localización del nodo, distinguiendo entre racks y DC. Esto es posible porque la información viene dada por la IP del nodo:

    10.168.35.57, sería:

- 168: DC
- 35: rack
- 57: nodo

- EC2snitch: utilizado para clusters formados por máquinas de Amazon EC2.

2.2.4 Almacenamiento de la información

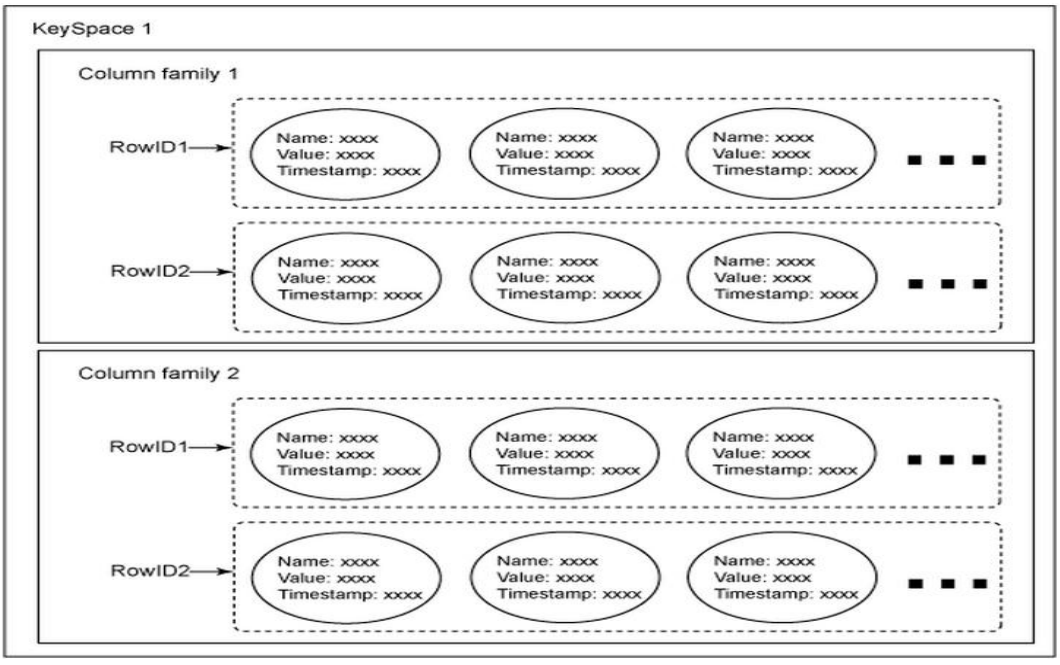

Figura 2: Ejemplo de almacenamiento de datos [6]

En la figura de arriba se puede observar un ejemplo de cómo estarían organizados los datos. Se pueden distinguir 4 partes:

- **Columnas:** es la unidad más básica del modelo de datos de Cassandra, cada columna consiste en un nombre, un valor y un timestamp.

- **Filas:** son una colección de columnas y están representadas por un nombre.

- **Column family:** son conjuntos de filas y están representadas por un nombre.

- **Keyspace:** son conjuntos de column family y también están representadas por un nombre.

2.2.5 Escrituras

Las escrituras en Cassandra pueden involucrar a varios nodos. Se realizan de la siguiente manera:

Se envía le escritura a cualquier nodo de la base de datos. Éste sabe cuál es el nodo responsable para dicha escritura, gracias al protocolo gossip, con lo que redirige la petición al nodo o nodos responsable de ella.

Se hace la petición de escritura, ésta se añade al commit log, el cual se encuentra en disco, lo que garantiza la durabilidad y además lo añade al memtable (memorias cache)[1], que está en memoria.

Cuando la memtable alcanza un tamaño máximo, se hace un flush a la sstable que está en disco.

Una vez se hace el flush de la mmtable a la sstable, se borra las correspondientes entradas del commit log, puesto que esta tarea ya no está pendiente de realizarse.

1 Hay tantas memtables como column families

La representación gráfica sería:

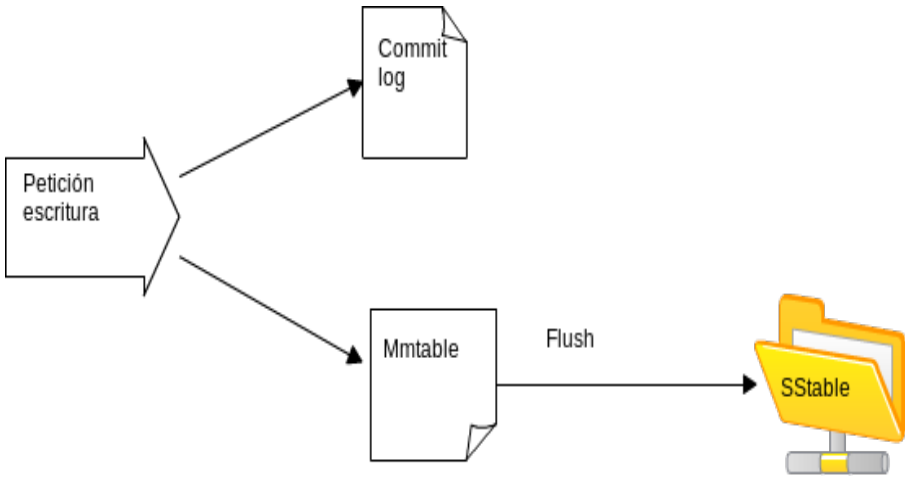

Figura 3: Estructuras modificadas en la escritura

La sstable es inmutable y para cada una de ellas, Cassandra genera en memoria las siguientes estructuras de datos:

- Una lista de filas y su posición dentro del fichero de datos.
- Un subconjunto de la lista de filas.

Dependiendo del nivel de consistencia, no se le devuelve el Ack al peticionario, hasta que un mínimo de nodos hayan realizado la escritura satisfactoriamente.

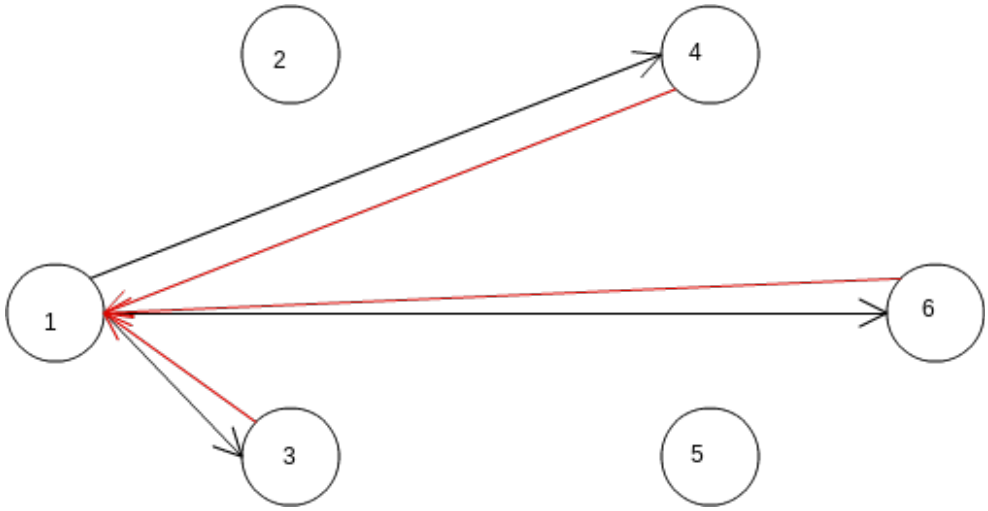

Figura 4: Petición de escritura

En la imagen de arriba, se muestra cómo sería una petición de escritura si tuviese un nivel de consistencia 3. El nodo 1 recibe la petición del usuario y propaga la escritura a los nodos 4, 3 y 6, los cuales son los responsables. Hasta que el nodo 1 no recibe el Ack de los tres nodos responsables, no le envía la confirmación al usuario, ya que de otra forma no se estaría garantizando el nivel de consistencia especificado.

Una vez se han actualizado las estructuras se devuelve una respuesta al peticionario.

En caso de que ocurriera un fallo antes de que se haya hecho un “flush” de éstas estructuras a disco, la información se perdería incluso habiéndola escrito ya en los respectivos nodos, porqué el sistema no estaría notificado de que dicha acción se ha llevado a cabo.

2.2.6 Borrados

Cuando se produce un borrado, no se borran directamente los datos, sino que a esa columna se le añade un valor especial (tombstone) para marcar su nuevo estado y es al cabo de un tiempo determinado, (gc_grace_seconds) que se puede modificar, cuando se borran de manera permanente los datos.

2.2.7 Lecturas

Las lecturas son más complicadas que las escrituras dada la gran cantidad de información que se puede almacenar en la base de datos.

Cuando se realiza una petición de lectura lo que ocurre es:

Para la column family sobre la que se hace la petición, se deben de mirar todas sus memtables y sstables en busca de las columnas que forman la fila pedida.

Cassandra hace un merge de todos los valores que tiene que devolver, ya que se puede dar el caso de que tuviese que devolver 4 columnas y cada una de ellas estuviese en una sstable distinta.

Se dispone de un filtro (bloomfilter) el cual permite saber si ese valor se encuentra en esa fila, aunque cabe la posibilidad de que se den falsos positivos, aunque no, falsos negativos.

Al igual que en las escrituras, el nivel de consistencia determina cuantos nodos son los que tienen que responder para poder confirmar la lectura al usuario. En caso de que los datos que entregan las réplicas difieran, se le entregará al usuario el valor con el timestamp mas reciente.

2.2.8 Distribución de los datos entre nodos

Cada nodo dentro del cluster tiene un token que se le asigna al iniciarse por primera vez el cluster, y que se le puede cambiar posteriormente. Cassandra genera un valor hash para cada clave de los datos. Dependiendo del valor resultante y comparándolo con los tokens de cada nodo, se determina cual será el nodo responsable de almacenar ese dato. Ejemplo:

Generación del hash:

Primary key	Murmur3 hash value
jim	-2245462676723223822
carol	7723358927203680754
johnny	-6723372854036780875
suzy	1168604627387940318

Figura 5: Valor hash generado por el particionador [7]

Rangos de tokens de los nodos:

Node	Murmur3 start range	Murmur3 end range
A	-9223372036854775808	-4611686018427387903
B	-4611686018427387904	-1
C	0	4611686018427387903
D	4611686018427387904	9223372036854775807

Figura 6: Tokens de cada nodo [7]

Distribución de valores según rangos:

Node	Start range	End range	Primary key	Hash value
A	-9223372036854775808	-4611686018427387903	johnny	-6723372854036780875
B	-4611686018427387904	-1	jim	-2245462676723223822
C	0	4611686018427387903	suzy	1168604627387940318
D	4611686018427387904	9223372036854775807	carol	7723358927203680754

Figura 7: Asignación de la fila en función del token [7]

2.2.9 Seguridad

Cassandra ofrece algunas características en cuanto a seguridad como son:

- Encriptación de la información entre el cliente y Cassandra y entre nodos usando Secure Socket Layer(SSL).

- Cuentas de usuario. Permite crear usuarios con sus respectivas contraseñas.

- Permisos. Una vez creado un usuario, se pueden definir de qué permisos dispone dentro de la base de datos.

2.2.10 Ventajas y desventajas

- Escalabilidad lineal añadiendo nodos.
- Fácil replicación y distribución de los datos.
- Esquemas de diseño flexibles.
- Compresión de los datos almacenados (hasta un 80%).
- No permite transacciones ACID ni hacer JOINs.
- No hay claves foráneas y las claves son inmutables.

2.3 Query plan

La información en las bases de datos de tipo SQL, por lo general, suele estar distribuida en múltiples tablas. Estas tablas hacen uso de campos para hacer referencia a relaciones que tienen con otras tablas de la misma base de datos y en global, forman un conjunto relacionado de información. Debido a que la búsqueda de información en este tipo de bases de datos puede implicar una compleja relación de múltiples campos, se han dedicado muchos esfuerzos a realizar optimizadores que lleven a cabo todas estas operaciones de la forma más óptima posible.

En el caso de las bases de datos de tipo NoSQL, el almacenamiento de los datos es mucho más flexible, con la consecuencia de que no se crean las asociaciones complejas comentadas en el párrafo anterior. Por ello, aún no se ha tenido la necesidad de realizar ningún optimizador para las bases de datos NoSQL.

2.3.1 RDBMS [8]

Cuando un usuario o una aplicación genera una consulta a una base de datos, hace una petición en la que especifica cuál es el resultado que quiere obtener pero no la manera en la que se debe de hacer.

Un query plan es un conjunto de pasos que se realizan para acceder a la información de una base de datos.

Las bases de datos relacionales disponen de un optimizador de consultas, el cual se encarga de escoger la mejor opción entre las disponibles para realizar una consulta de la manera más óptima.

La base de datos lo que hace es recibir la consulta, la parsea y la transforma en un árbol algebraico, el cual representa la estructura de la consulta. Acto seguido dicho árbol lo recibe el optimizador, el cual intenta hacer las siguientes optimizaciones:

- Semántica: se encarga de reescribir la sentencia SQL en otra equivalente y más eficiente.

  El objetivo de este paso es encontrar consultas que tengan formas incorrectas o se contradigan (resultados null). En este paso aún no se tiene en cuenta el coste de ejecutar las consultas generadas.

- Sintáctica: transforma la consulta de SQL a álgebra relacional y busca un orden de operaciones que minimicen el intercambio de datos.

  Se puede dar el caso de que para una misma consulta, se generen varias secuencias algebraicas. En ese caso el optimizador se queda con la menos costosa en términos heurísticos. También puede representar la petición en forma de árbol, donde los resultados van fluyendo hacia los nodos superiores (el más alto, el root, devuelve el resultado de la consulta), dónde cada nodo intermedio representa una operación

(joins, proyecciones, uniones) y dónde las hojas representan una búsqueda en disco (acceso a las tablas).

El objetivo es que fluya el mínimo de información posible hacia los nodos superiores.

- Física: tiene en cuenta estructuras físicas (índices, clusters...), paths de acceso y algoritmos que implementan las operaciones, con el fin de decidir basándose en unos costes cual es el más efectivo.

  La entrada que recibe esta fase es el árbol generado en la parte de optimización sintáctica y transforma éste árbol en un árbol de procesos. Para cada árbol de procesos que se genera, el optimizador hace una estimación del coste que tendría su ejecución, ya que ejecutar cada plan para evaluar su coste no tendría sentido alguno.

Para la estimación de costes se tiene en cuenta:

- Reunir información estadística sobre datos almacenados.
- Dado un operador en el plan de ejecución e información estadística, para cada sub plan:
  - Información estadística sobre la salida.
  - Coste estimado de ejecutar el operador (nodo intermedio).

Los pasos que se siguen en ésta parte son:

- Generación de alternativas: se tienen en cuenta todas las alternativas existentes para la ejecución de cada operando (nodo intermedio).

- Cardinalidad y estimación del tamaño de los resultados intermedios: se estima el tamaño de los resultados de las subconsultas y la frecuencia con la que se distribuyen esos resultados.

- Estimación del coste para cada algoritmo y ruta de acceso: mediante fórmulas aritméticas se calcula el coste de los accesos.

- Elegir la mejor opción y generar el plan de acceso.

Finalmente se ejecutan las operaciones y se devuelve el resultado al usuario. La representación gráfica sería:

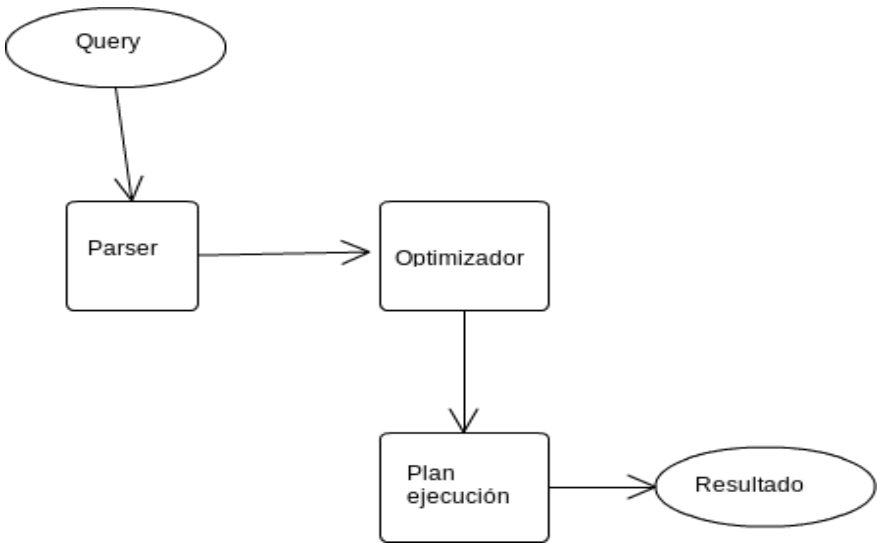

Figura 8: Proceso optimización consulta en RDBMS

Cada consulta que resuelve el optimizador, genera una secuencia de accesos y de operaciones entre los elementos a tratar, del cual se almacena el orden en el que se han realizado. Esto luego le sirve para ordenar el resultado.

2.3.2 DDBMS [8]

En este tipo de bases de datos lo que predomina es el tiempo de comunicación, ya que el coste de enviar información de un sitio a otro dependiendo del tamaño puede ser muy elevado. Es por eso que este tipo de optimizadores a diferencia de los centralizados que ponen énfasis en reducir accesos a disco, éste lo haga en reducir al mínimo las comunicaciones.

En sistemas distribuidos, se puede ganar en reducción de costes haciendo uso del paralelismo, puesto que un proceso se puede encargar de buscar una información X en un nodo I mientras otro busca una

información Y en un nodo J.

Las optimizaciones que sigue este optimizador son:

- Semántica: es la misma que en el centralizado. Intentar generar sentencias SQL más eficientes respecto a una dada.

- Sintáctica: igual que en el modelo centralizado pero se analiza además dónde está ubicada la información y qué fragmentos de la consulta se pueden borrar (reducción). Se borran aquellos que vayan a formar relaciones nulas.

  Se generan nuevos árboles (bushy trees), los cuales permiten realizar ejecuciones paralelas.

- Física: esta parte se divide en dos pasos:
  - Optimización global: decide dónde se debe de ejecutar la operación e inserta primitivas de comunicación en el plan de ejecución.

- Optimización local: igual que el optimizador de las BD centralizadas.

De los pasos que realiza el optimizador en las bases de datos SQL, que se han podido ver, se podría hacer uso en NoSQL, de la optimización global, de la parte física del sistema distribuido. Ésta parte es importante si se tiene la base de datos distribuida en varios nodos, ya que el tiempo de comunicación entre ellos puede suponer un gran coste para la ejecución de la consulta en caso de tener redes congestionadas. Por lo tanto, sería interesante poder realizar la consulta de forma que el nodo que la reciba sea autosuficiente para poder dar una respuesta.

3. Módulo

La primera parte de éste apartado se centra en dar una visión general del módulo, de cómo está formado el internamente y cómo intervienen cada una de las partes en un ejemplo de ejecución.

En la segunda parte se entra en detalle, en cómo se han realizado cada una de las partes vistas en la primera parte para que funcionen.

3.1 Descripción

El módulo recibe una petición y la optimiza automáticamente en función de unos parámetros de rendimiento. Esta optimización consiste en

dividir la consulta en fragmentos más pequeños, del tamaño que decida la política, y después, se reparte el trabajo entre los threadQ para que la consulta se ejecute de forma concurrente. El trabajo se repartirá de una forma u otra en función de los parámetros de rendimiento que decida la política[2].

Ejemplo:

Supongamos que el módulo recibe una consulta de un total de 100 filas y de 100 columnas. La política decide que se deben utilizar 10 filas y 10 columnas para cada subconsulta, con lo que tendríamos 10 bloques de filas y otros 10 bloques de columnas para asignar a los threadQ. En función del resto de parámetros de rendimiento que decide política, el siguiente paso consistiría en repartir de una forma u otra. Supongamos que se ha decido repartir el trabajo iterando por columnas.

2 Ver parámetros y funcionamiento en el apartado 3.2.1 Parámetros de política

Ahora el módulo empieza a repartir bloques de 10 filas y columnas a cada threadQ, es decir, al primero le data las filas y columnas de la 1 a 10, al segundo las filas y columnas de la 11 a la 20 y así hasta que todo el trabajo esté repartido.

Los threadQ, una vez reciban las filas y columnas que deben recuperar de la base de datos, irán realizando concurrentemente las consultas y almacenando los resultados para posteriormente devolverlos.

3.2 Diseño

El objetivo es diseñar un módulo que sea capaz de recibir peticiones y automáticamente, lanzarlas contra la base de datos, con una previa optimización en función de unas métricas de rendimiento.

Estas métricas de rendimiento implican:

- Poder dirigir una consulta a los nodos que realmente almacenan esa información, es decir, que físicamente son los que la tienen.

- Poder elegir el orden en el que se realiza la consulta, es decir, si se recibe una lista de filas y una de columnas a consultar, elegir si la iteración se debe de hacer por filas, o bien por columnas.

- Decidir si se le debe de dar prioridad a la localidad de los datos, que es el primer punto explicado, o bien a las filas que se deben de consultar y por lo tanto, hacer un balanceo de carga. [3]

- Decidir el número total de threadQ que se deben de utilizar para realizar la consulta.[4]

3 Explicado en el apartado de la 3.1.1 Parámetros de política
4 En el apartado 3.3.4 Política se explica cómo se decide que número de threadQ utilizar

El módulo está dividido en 4 partes para poder cumplir con los requerimientos de los parámetros de rendimiento y también por las siguientes razones:

- Poder modificar la forma en la que se obtiene la consulta sin que esto implique realizar cambios en el “núcleo” del módulo, que es la clase mechanism. Para ello se ha diseñado la clase queryAnalyzer.

- Poder modificar la política para las pruebas de rendimiento, que implican la modificación de parámetros, sin que implique realizar cambios sobre mechanism. Para ello se ha diseñado la clase policy.

- Poder explotar al máximo el grado de paralelismo de la máquina, para intentar reducir los tiempos de ejecución. Para ello se ha creado la clase threadQ.

El esquema gráfico del módulo sería el siguiente:

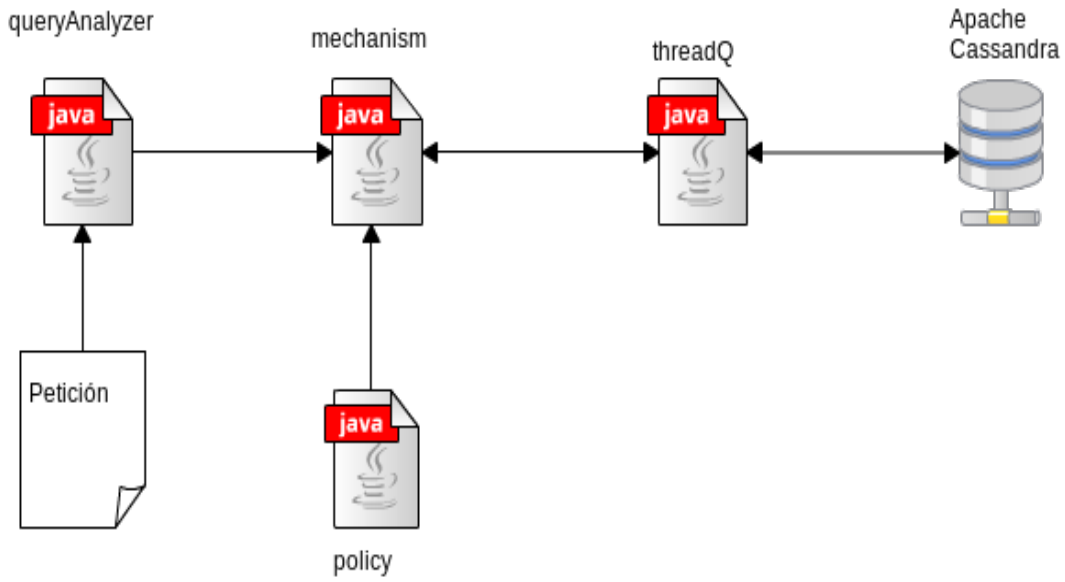

Figura 9: Diseño del módulo

En caso de que un usuario quiera realizar una petición a la base de datos haciendo uso del mecanismo, se le pedirá que introduzca una keyspace sobre la que quiera realizar la petición, una column family, una lista de filas y una lista de columnas (este último es opcional). La interacción con el mecanismo se realizará a través de una terminal.

El queryAnalyzer es la clase encargada de leer la consulta, separar las partes diferenciables (keyspace, column family, lista de filas y columnas) y de hacérsela llegar a la clase mechanism.

La clase mechanism es la encargada de recibir los datos que le pasa la clase queryAnalyzer, de interpretar los parámetros de rendimiento que decide la clase policy que se deben de utilizar, de preparar la consulta en función de los parámetros, de repartir el trabajo para las clases threadQ y de recoger los resultados.

La clase threadQ recibe órdenes de la clase mechanism. Se le indica que datos debe leer de la base de datos, hace la consulta y le devuelve los resultados obtenidos a la clase mechanism. La interacción entre el threadQ y la base de datos se realiza mediante el uso de una API basada en Thrift[5], haciendo uso de métodos que ésta proporciona para realizar las lecturas.

5 Ver en apartado 3.3.1 Hector

Finalmente, cuando la clase mechanism ha recibido los resultados, se los muestra al usuario.

3.2.1 Parámetros de política

En esta sección se pretende dar una descripción gráfica de como funcionan algunos de los parámetros de rendimiento para que éstos no creen confusión.

Los parámetros número de filas y columnas por subconsulta y orden de iteración funcionan como se puede observar en el siguiente dibujo:

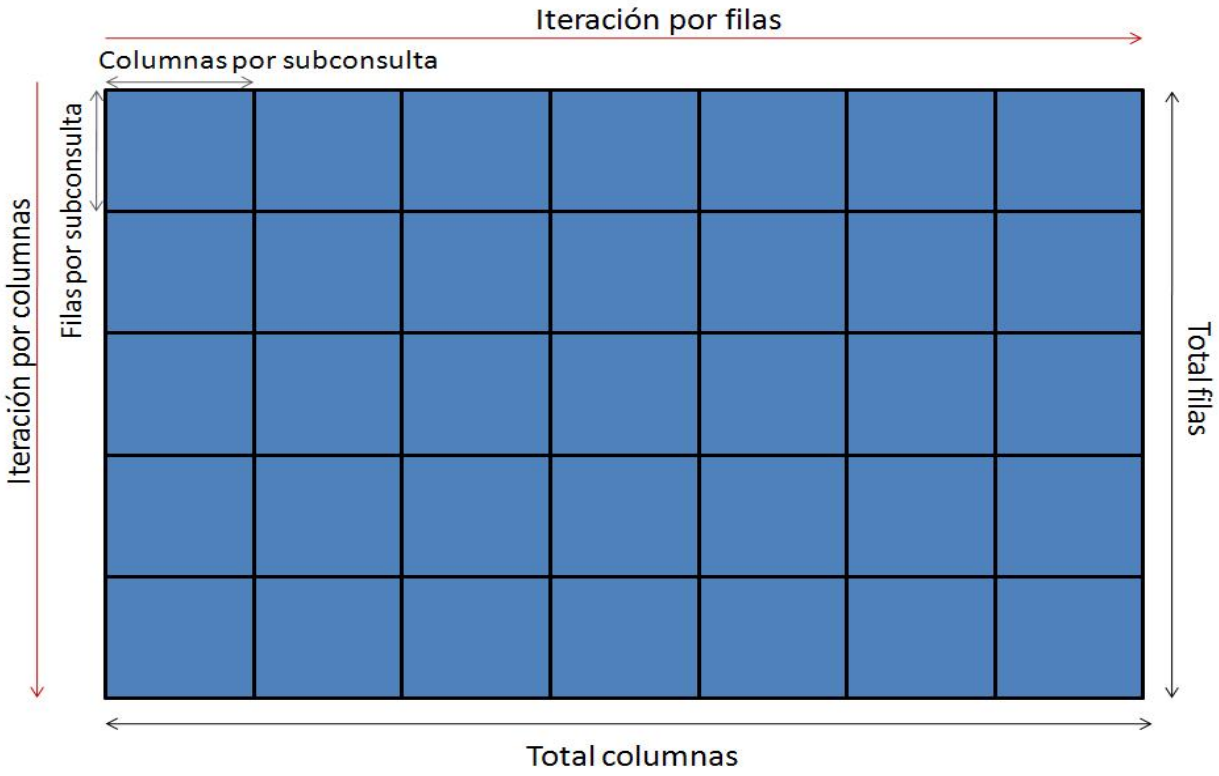

Figura 10: Parámetros de política 1

Dadas n filas y m columnas totales, el número de filas y columnas por subconsuta serán un subconjunto que representen desde 1 fila y 1 columna hasta n filas y m columnas.

En cuanto al orden de iteración, si se elige iteración por columnas lo que se hace es ir consultando las mimas columnas para todas las filas y una vez recorridas todas las filas, se mueve la ventana de columnas. En cambio, si se elige iteración por filas, lo que se hace es consultar todas las columnas de

una misma fila y después se cambia de fila.

El funcionamiento del parámetro token aware se puede observar en la siguiente figura:

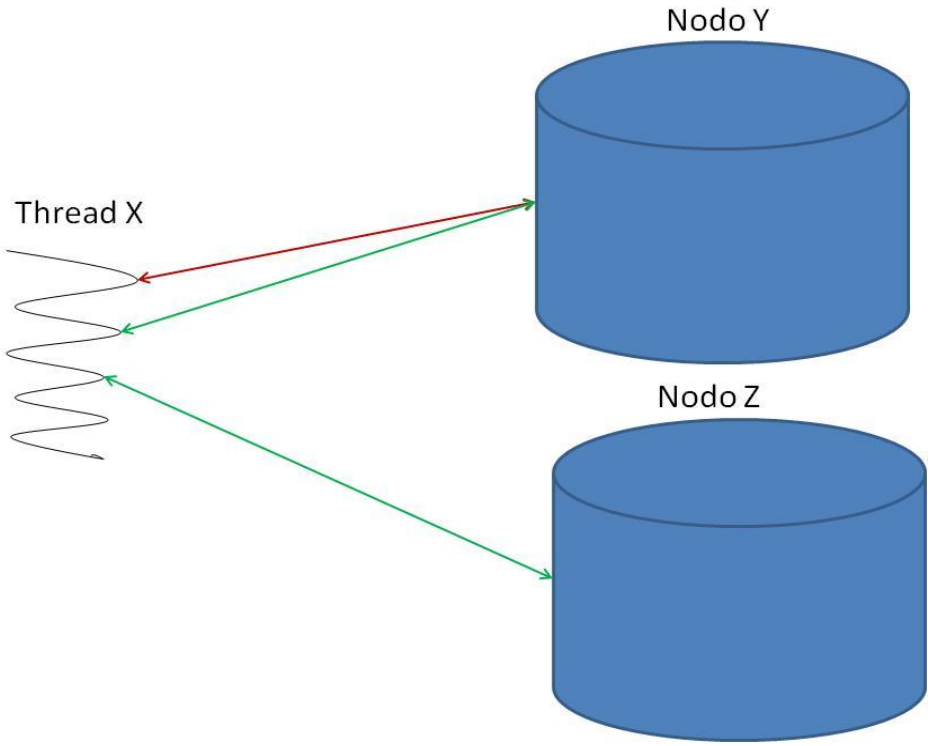

Figura 11: Parámetros de política 2

Las líneas de color verde representarían una consulta de unas filas que físicamente almacena el nodo Y, pero que al no elegirse la opción de utilizar token aware, al threadQ se le pasan las direcciones de los dos nodos que forman la base de datos y se

deja que la API decida a que nodo preguntar. En cambio, la línea de color rojo representaría una consulta haciendo uso de la opción de token aware, donde se iría a preguntar al nodo que físicamente almacena los datos.

El parámetro prioridad, lo que hace es decidir si dar prioridad al token aware o bien a las filas y por lo tanto se hace un balanceo de carga. Es decir, si a un nodo le corresponden 19 filas y a otro 3, se hace un balanceo de carga, para que cada uno de ellos se haga cargo de 11.

3.2.2 Ventajas

Las ventajas que ofrece el módulo son:

- Realizar consulta de datos indicando una lista de filas y de columnas(opcional).

- Posibilidad de modificar la forma en la que introducir la consulta de forma rápida y sencilla.

- Posibilidad de incorporar diferentes políticas, para evaluar consecuencias en el tiempo de ejecución de distintos parámetros, de forma rápida y sencilla.

- Ejecución de las consultas en paralelo. Para consultas de mucha información es muy importante poder alcanzar el máximo grado de

paralelismo posible para reducir tiempos.

- Posibilidad de lanzar consultas a los nodos que almacenan la información. Esta opción nos permite minimizar el tráfico de red, que en casos en los que se tenga una red congestionada puede ser de mucha utilidad.

3.2.3 Simplificaciones

Inicialmente se había incorporado la opción de realizar consultas mediante rangos, es decir, se le indicaba una fila de inicio, una fila de fin, una columna de inicio y una columna de fin y se debía realizar la petición. Por simplicidad finalmente se opto

por realizar las consultas mediante listas y desestimar los rangos, ya que al darse la posibilidad de que no fueran rangos continuos podía darse el caso de realizar lecturas de valores inexistentes o bien de valores no deseados.

Para la consulta de los datos, se le debe indicar a la estructura que ofrece la API qué tipo de datos son tanto filas, como columnas, como resultados. Lo que se ha hecho para evitar tener que consultar la meta-información de la column family, ha sido que la política indique el tipo de datos que son.

3.3 Análisis económico

El coste para la realización del diseño del módulo y de la política es el mostrado en la siguiente tabla:

Tarea	Horas	Precio/hora	Total(€)
Diseño módulo	30	8,5	255
Diseño políticas	20	8,5	170
Coste total	**50**	**8,5**	**425**

Tabla 1: Coste del diseño del módulo y políticas

El coste para la implementación del módulo y de las políticas es el mostrado en la siguiente tabla:

Tarea	Horas	Precio/hora	Total(€)
Implementación módulo	213	8,5	1810,5
Implementación políticas	92	8,5	782
Tests	46	8,5	391
Coste total	**351**	**8,5**	**2983,5**

Table 2: Coste implementación del módulo y políticas

3.4 Implementación

Para la implementación del módulo se ha hecho uso de una API llamada Hector.

3.4.1 Hector

Hector es una API implementada en java basada en Thrift, que permite trabajar con Cassandra. Thrift es una interfície que permite crear servicios para múltiples lenguajes. Además tiene una estructura interna (pila) bien diferenciada, en la que se pueden distinguir, el código de entrada, el servicio que se genera a partir del código, el protocolo utilizado y la capa de transporte.[1]

Una de las ventajas que ofrece Thrift es que la capa de transporte y la de protocolo, se pueden modificar sin la necesidad de tener que volver a compilar el código. [4]

Algunas de ventajas que ofrece Hector son: [2]

- Agrupación de conexiones que permiten mejorar el rendimiento y la escalabilidad (no hace falta abrir una nueva conexión para cada operación).

- Monitorización y control de JMX.

- Balanceo de carga. Solo implementa dos políticas. Una de ellas es Round Robin y la otra es la de pedir al nodo con menos conexiones activas.

- Reconexión automática de nodos caídos.

- Autodescubrimiento de nuevos nodos.

- Soporte para “failover”, es decir, redirigir peticiones a nodos redundantes cuando el nodo responsable cae. Ofrece soporte para modos de operación:
  - Fail fast: devuelve el error al cliente sin volver a intentarlo.
  - En caso de fallo, prueba con un nodo mas elegido al azar.
  - En caso de fallo, prueba todos los nodos que son conocidos.

3.4.2 Inicialización

En la inicialización del mecanismo, se determina cual es el particionador que está usando la base de datos. Para ello se inicializa un cluster, con el método *getOrCreateCluster* que proporciona la clase *HFactory* de Hector, con un nodo y se llama al método *describePartitioner*, que proporciona la clase *Cluster.* El método *describePartitioner* devuelve el nombre del particionador utilizado con el formato: *"org.apache.cassandra.dht.NOMBREPARTICIONADOR"*.

Una vez determinado el particionador se crea una instancia de éste, utilizando el método *lookForPartitioner*, con el que posteriormente se generaran los token de cada clave, para saber en qué nodo esta almacenada, en caso de que se quiera explotar la localidad de los datos.

3.4.3 Lectura

Para la realización de la lectura, se crea una instancia de la clase queryAnalyzer, pasándole un nodo del cluster, con la que se obtienen los parámetros introducidos por el usuario. Lo que extrae de la consulta introducida son:

- La keyspace sobre la que se trabajará.

- La column family sobre la que se desea realizar la consulta de los datos.

- La lista de filas que se desean consultar.

- Por último, la lista de columnas que se desean consultar de cada fila.(Este parámetro es optativo)

Entre la lectura de la keyspace sobre la que se desea trabajar y la lectura de la consulta, se inicializan dos estructuras. Una de ellas es una lista con todos los nodos que forman la base de datos. Se obtiene sus IPs correspondientes para poder dirigir las consultas posteriormente a esos nodos en concreto.

Saber cuántos son los nodos permite saber cuántos threadQ crear. Cómo máximo se crearán tantos como nodos haya en el cluster (depende de las consultas que se tengan que realizar), en caso de que se quiera explotar la localidad de los datos.

Para saber cuántos son los nodos, se llama al método *getNodesIptokens,* dentro del cual se crea una consulta en CQL, a la keyspace system y a la tabla peers. De ahí se extraen las IPs de los nodos, a través del campo peer.

Una vez se obtienen las IPs de los nodos y se sabe cuántos son, se pasa a obtener los tokens de cada uno de ellos. Para obtenerlos se llama al método describeRing, que proporciona la clase Cluster, que dado un keyspace, devuelve los tokens de cada uno de los nodos. Los tokens obtenidos se guardan en una lista para poder hacer las comparaciones posteriormente.

Una vez hechos los pasos descritos anteriormente, se devuelven los parámetros introducidos por el usuario al mecanismo.

Ahora el mecanismo sabe cuántos son los nodos que forman el cluster, cuáles son sus correspondientes direcciones y los tokens de cada uno de ellos. Con esto se crea una lista de clusters individuales, donde cada cluster esta únicamente formado por un único nodo. Además se crea un cluster con todos los nodos. Esto se hace para utilizar unos u otros dependiendo de si

se quiere explotar la localidad de los datos o no. Es decir, en caso de que la política decida que se debe utilizar token aware (explotar la localidad de los datos), a los threads encargados de realizar una consulta, se le asignarán las filas y columnas correspondientes y el nodo al que debe ir a preguntar, que será el que físicamente tenga los datos.

En caso de que la política decida que no se debe utilizar token aware, se le asignara el cluster con todos los nodos, para que automáticamente se dirija esa consulta al nodo que la API decida que es el más conveniente.

Cuando se ha finalizado la inicialización de los clusters, se inicializa la política, la cual recibe el número de filas y el número de columnas totales a consultar.

Dependiendo de la política de la que se esté haciendo uso y del número de filas y columnas recibidas, ésta decidirá:

- El número de filas y columnas que se debe utilizar para cada subconsulta.

- Si se debe utilizar token aware o no.

- En caso de no utilizar token aware, el número de threadQ a utilizar como máximo.

- Si se le da prioridad a token aware o a las filas.

- El orden de iteración en el que se deben consultar los datos. Puede ser por filas, o bien por columnas.

- El tipo de dato que son las filas.

- El tipo de dato que son las columnas.

- El tipo de dato que son los resultados.

En caso de que la política indique que sí se debe tener en cuenta token aware, entonces las peticiones se deben realizar sobre los nodos que almacenen físicamente esos valores.

Una vez la política le ha indicado al mecanismo el tipo de dato que son las filas y las columnas (en caso de que haya), el mecanismo procede a una transformación de las listas que ha recibido del queryAnalyzer. Las transforma al tipo de objeto que le haya indicado la política. Ésta transformación se realiza para evitar excepciones

a la hora de leer los resultados, ya que se deben introducir en la consulta datos del tipo que realmente son en la base de datos.

El siguiente paso es preparar las consultas en función de si se usa token aware o no. Para ello se hace uso de un ArrayListMultimap, que proporciona el paquete *com.google.common.collect* el cual permite almacenar varios valores para una misma fila.

Para cada fila que ha introducido el usuario, se genera el token de la fila con el método getToken del particionador del que se esté haciendo uso. Una vez se tiene el token, se utiliza un algoritmo de búsqueda binaria, que proporciona la clase Arrays, para buscar dentro de la lista de tokens, a que nodo se corresponde el token generado de la fila y se almacena esa fila dentro del ArrayListMultimap, en la posición devuelta por el algoritmo.

De esa forma se consigue tener en cada posición del Multimap las filas que tienen que ir a un nodo correspondiente. Es decir, en la posición *“i”* del Multimap estarán almacenadas todas aquellas filas que se deban consultar al nodo *“i”* y así con todas las demás filas.

En caso de que la política decida que no se debe utilizar token aware, todas las filas se almacenaran en la primera posición del Multimap.

Cuando se tienen las filas distribuidas para cada nodo, en caso de que la política haya decidido dar prioridad a las filas en vez del token aware, se realiza una función de balanceo sobre las listas generadas para cada nodo.

Para realizar éste balanceo de carga lo que se ha hecho es lo siguiente:

Se mira para cada nodo cuantas filas tiene y se guarda cual es el que tiene más, cual es el que tiene menos y cuál es la diferencia de filas entre

ambos. Una vez se sabe el nodo destino, el nodo origen y la cantidad de datos a mover, se coge una sublista, de tamaño la mitad de la diferencia y se mueve del nodo origen al nodo destino y se eliminan del nodo origen. Cabe destacar que siempre se coge la sublista desde el final de lista. Con esto se explota al máximo la localidad de los datos, dado que las últimas filas pueden ser las que se han recibido de otros nodos y no se estarían moviendo las que se han asignado por localidad a ese nodo.

Ésta tarea se realiza hasta que la diferencia entre las filas de los nodos sea de una fila como máximo.

Con todo el trabajo anterior realizado, es hora de empezar a mandar trabajo a los threadQ.

Para ello se crea un vector de threadQ. Para el caso en el que se utilice token aware, el vector será del tamaño de los nodos que formen el cluster. En caso

de que la política haya decido no utilizar token aware, se crearán cómo máximo tantos como la política decida. Aunque cabe remarcar, que es posible, que no todos se llegan a inicializar, dependerá del número de consultas que se tengan que realizar.

Se envía cada posición del Multimap a otra función (orderWork), que es la que se encarga de partir el trabajo en trozos más pequeños para cada threadQ. (Recordar que cada posición del Multimap se correspondía a las consultas a un nodo específico en caso de utilizar token aware, o bien estaban todas en la primera posición si no se utilizaba token aware.)

La función orderWork lo que hace es:

- Determinar si la política ha decidido iterar por filas o por columnas.

- Partir las listas recibidas en tantas filas y tantas columnas como haya decidido la política.

- Determinar si se debe de tener en cuenta token aware.

- En caso de que aun queden threadQ sin inicializar:
  - Asignarle la keyspace y la colum family que haya introducido el usuario.
  - Asignarle una sublista de filas y columnas, dependiendo del valor de filas y columnas que haya decidido la política.
  - En función de si se utiliza token aware, asignarle un cluster específico al threadQ (con un solo nodo) o uno con todos los nodos si no se utiliza token aware.

    - Notificarle los tipos de datos para filas, columnas y resultados.

- En caso de que ya se haya hecho uso de todos los threadQ, lo que se hace es:
    - Esperar a que el threadQ acabe de realizar la consulta que se le haya asignado anteriormente.
    - Asignarle una nueva lista de filas.
    - Asignarle una nueva lista de columnas.
    - Asignarle un nuevo cluster.
    - Ejecutar el threadQ.

Cuando se hayan asignado todas las filas y todas las columnas, lo que se hace es esperar a que todos los threadQ finalicen sus correspondientes consultas.

Una vez han acabado, se recopilan todos los resultados correspondientes de cada threadQ y se le devuelve al usuario.

También se ha implementado la posibilidad de que la consulta del usuario solo necesite estar compuesta por una lista de filas. El proceso es el mismo que cuando se incluyen columnas hasta el paso de distribuir las filas y las columnas. Aquí tan solo se le asignan filas, sin columnas a cada threadQ.

Se tienen en cuenta las mismas comprobaciones: si es token aware y si ya se han consumido todos los threadQ. Con las filas asignadas, cada threadQ lee de la BD todas las columnas de cada fila que se le hayan asignado.

3.4.4 Política

La política implementada se inicializa con el número de filas y el número de columnas. En caso de no utilizar columnas, el parámetro número de columnas será 0. Los parámetros que decide son:

- El número de filas y columnas que decide son estáticas. Se introducen manualmente.

- El token aware. Para decidir si utilizar token aware o no, lo que hace la política es verificar si todos los nodos que forman el cluster están activos o no. En caso de que detecte que alguno de ellos está en estado DOWN, no se tendrá en cuenta el token aware.

- En función del token aware se determinan dos parámetros:
  - En caso de que no se utilice, la política debe decidir cuantos threadQ, como máximo, podrá crear el mecanismo. Hay que recordar que en función de las consultas a realizar se usarán un número de threadQ u otro.

    Para tomar la decisión de los threadQ, se consulta el número de CPUs de las que dispone la máquina y se crearán tantos como CPUs.
  - En caso de que se tenga en cuenta el token aware, el número de threadQ será como máximo el número de nodos que formen la BD. Aparte, se debe decidir si se tiene que dar prioridad a las filas o al token aware. Esto se debe decidir para evitar desbalanceos de carga entre nodos.

En caso de que se le dé prioridad al token aware, porque se quiera minimizar el tráfico de datos por la red, ya sea por una mala distribución de los nodos o porque existe un cuello de botella en algún sitio, los nodos se quedarían con las filas sin la redistribución.

- El orden de iteración. Éste parámetro decide si se debe iterar por filas o por columnas. La configuración de éste parámetro es manual.

- El tipo de dato que son las filas, columnas y valores. Esto es necesario para poder realizar la consulta correctamente sin que se produzcan excepciones.

3.4.5 TreadQ

Los threadQ, son una extensión de la clase Thread. Para inicializarlos hay que pasarles:

- Una keyspace.
- Una column family.
- Una lista de filas y otra de columnas (en caso de que se utilicen columnas).
- Un cluster.
- Los tipos de datos que son las filas, columnas y resultados.

Según si se utilizan columnas o no, el threadQ ejecuta dos posibles métodos para recopilar la información de la BD:

- En caso de que no se especifiquen columnas, se crea un MultigetSliceQuery de la clase Hfactory, al que se le pasa el keyspace, la column family, las filas en formato de lista y no se le especifica ninguna columna de inicio ni de fin. Como se tiene el tipo de dato que son tanto filas, columnas y resultado, se definen 3 serializadores para cada uno y se le pasan como argumento al MultigetSliceQuery.

  Esto lo que hace es recuperar todas las columnas que tiene cada fila introducida.

- En caso de que se especifiquen las columnas, lo que se hace es iterar para cada fila introducida, sobre las columnas especificadas y se recuperan los valores de la base de datos. Al igual que en

caso anterior, se le pasa el tipo de dato de las filas y las columnas. Para la recuperación del resultado, se ejecuta un método u otro en función del tipo de dato que sea.

Los valores que se obtienen de cada consulta se guardan en una estructura propia de cada threadQ, que posteriormente le devolverá al mecanismo.

4 Tests

Se han realizado dos tipos de pruebas. Una de ellas ha sido funcional, en la que se ha comprobado que lo que se ha implementado funcionaba como era esperado y la otra prueba ha sido de rendimiento, en la que se han realizado varios experimentos para poder observar cual es la sobrecarga añadida por el mecanismo y los distintos parámetros de rendimiento. Sin embargo, ni el cluster, ni el tamaño de la base de datos, ni el tamaño de la consulta son lo suficientemente grandes como para que la evaluación de las políticas sea representativa.

Las pruebas se han realizado sobre dos entornos. Uno de ellos ha sido en local, simulando distintos nodos en la misma máquina y el otro entorno ha sido en un sistema distribuido real.

4.1 Local

Para realizar las pruebas en local del módulo que se ha implementado, se ha hecho uso de una librería de código abierto en Python[9], la cual facilita la creación de nuevos clusters en la misma máquina, así como añadir y quitar nodos según se quiera.

En local se han realizado varias pruebas funcionales, que obligasen al mecanismo a realizar todas las posibles consultas que podría realizar un usuario cualquiera.

Los resultados obtenidos en las pruebas funcionales locales han sido correctos, en cuanto a los nodos a los que se ha ido a buscar la información y los resultados leídos, sin embargo, de los resultados para las pruebas de rendimiento en este entorno, no se han podido apreciar grandes diferencias, dado que todos los nodos residían en la misma máquina.

4.2 Distribuido

Las otras pruebas se han realizado sobre una máquina del departamento de Arquitectura de Computadores.

La gestión de la máquina se ha realizado de forma remota. Se le ha instalado y configurado Cassandra para poder realizar las pruebas en un sistema distribuido.

La base de datos está formada por un total de 2 nodos, con estrategia de replicación Simple Strategy, que es la indicada cuando todas las máquinas se encuentran en el mismo data center. Esta estrategia permite al particionador que sea el que elija la primera réplica y las demás irán en el sentido de las agujas del reloj. Además se ha escogido que el factor de réplica sea de 1, que implica que tan solo hay una copia de las filas en un nodo.

Para las pruebas en la base de datos distribuida, se ha generado un “Poblador”, con el que indicándole un rango de inicio y fin, introduce fin-inicio filas de datos y de columnas en Cassandra.

Esto se ha hecho para poder simular consultas de grandes volúmenes de información.

Además, también se han generado scripts para la gestión. Uno de ellos permite generar peticiones de filas y columnas aleatorias dentro de un rango específico. El otro lanza la ejecución del módulo con todas las posibles variantes en cuanto a parámetros de rendimiento se refiere y hace la recopilación de los tiempos de ejecución de cada uno.

Se han realizado distintos bloques de pruebas de rendimiento, dentro de los cuales se han ido modificando los parámetros de rendimiento, para poder observar cómo afectan éstos en el tiempo de consulta. En todos los bloques la consulta ha sido sobre un **total de 3000 filas y 3000 columnas**.

Los tiempos que se han medido son:

- **Tiempo de preparación**: es el tiempo que tarda el módulo desde que se lee la consulta hasta que ha realizado la preparación de las filas. Esto es hasta que se hayan distribuido las filas por el Multimap y en caso de hacer uso de token aware y dar prioridad a las filas, hasta que se haya hecho el balanceo de carga.

- **Tiempo de espera**: es el tiempo que tarda el módulo desde que se ha acabado de realizar la preparación, hasta que se obtiene el último resultado del último thread.

- **Tiempo de tratamiento**: es el tiempo que se tarda en mostrar todos los resultados por pantalla.

- **Tiempo medio de threadQ**: es el tiempo medio que tardan los threadQ en realizar todo el trabajo que se les ha asignado.

Los parámetros orden de iteración, token aware y prioridad, se han modificado para formar todas las combinaciones posibles. Recordar que el parámetro prioridad, solo se tiene en cuenta cuando se hace uso de la opción token aware, para decidir si dar prioridad a token aware o bien a las filas y hacer un balanceo de carga de éstas entre los nodos.

Las combinaciones son:

- Token aware, iteración por filas, prioridad a token aware.

- Token aware, iteración por columnas, prioridad a token aware.

- No token aware, iteración por filas.

- No token aware, iteración por columnas.

- Token aware, iteración por filas, prioridad a filas.

- Token aware, iteración por columnas, prioridad a filas.

Los tiempos obtenidos, se han calculado haciendo la mediana de un total de 10 tiempos para cada caso, es decir, para cada caso con un número determinado de filas y columnas por subconsulta, token aware, iteración por filas y prioridad a filas se ha medido 10 veces y así con todos los demás. Además, las mediciones se han realizado alternadas, por si la carga de la máquina en ese momento fuese muy alta y la ejecución fuera más

lenta, no solo afectase a los tiempos de un test en concreto.

Para mostrar el rendimiento del mecanismo en función del número de filas y columnas por cada subconsulta, se han realizado dos pruebas. Una de ellas ha consistido en realizar subconsultas de 100 filas y 100 columnas sobre el total de 3000 filas y columnas, donde se ha podido observar el elevado coste de tiempo que se obtiene al desgranar tanto el volumen de datos. La otra ha consistido en no partir el total de datos en subconsultas, es decir utilizar 3000 filas y 3000 columnas para subconsultas, donde se ha podido observar que los tiempos se han reducido significativamente respecto al caso de 100 filas y columnas.

Los resultados para el caso de 100 filas y columnas, son los que se muestran a continuación:

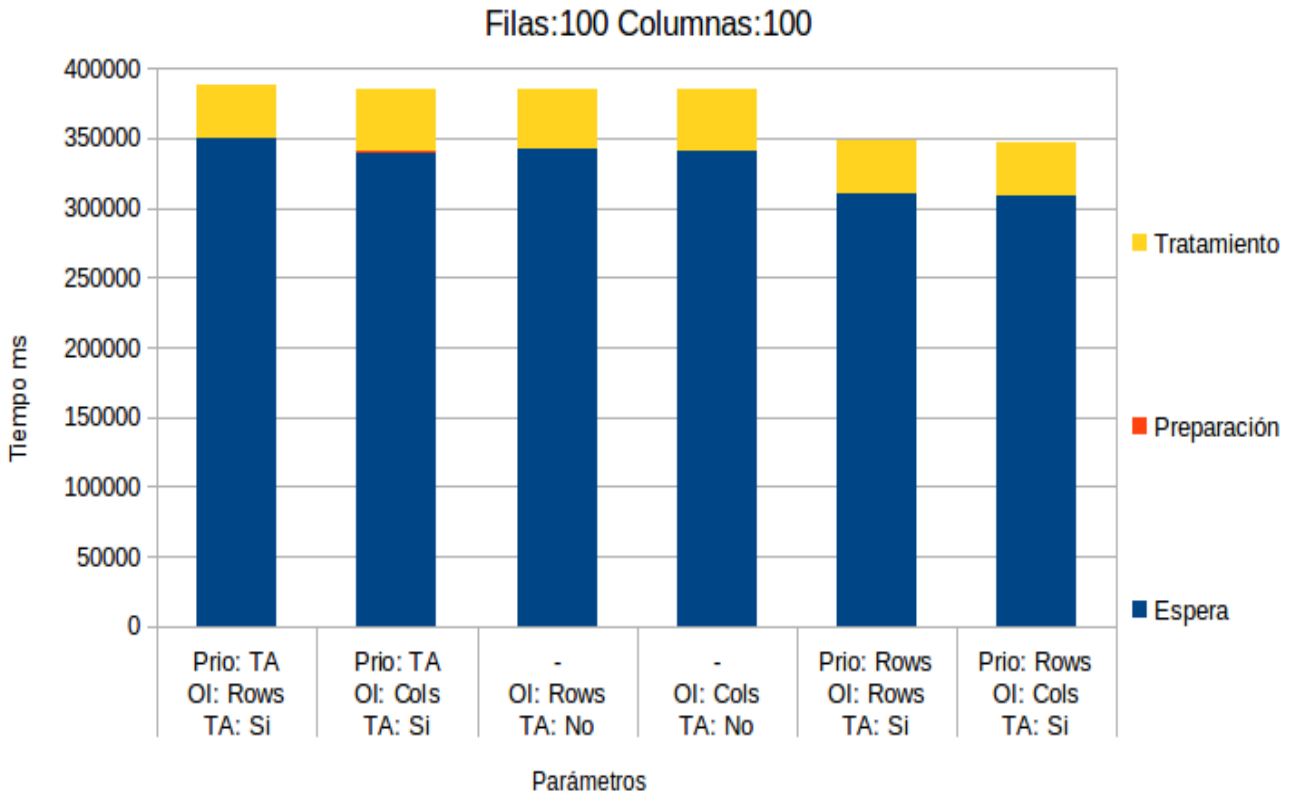

Figura 12: Tiempos rendimiento. Filas:100 y columnas: 100

Se ha podido observar que:

El tiempo de preparación es mínimo en todos los casos. El máximo ha sido de 156 ms para los casos de utilizar token aware y dar prioridad a las filas.

El tiempo de tratamiento ha sido bastante constate, con una media de unos 40 segundos para todos los casos.

El tiempo que más ha predominado, es el tiempo de espera, dónde el máximo se ha dado para el caso en el que se ha utilizado token aware, iteración por filas y prioridad a token aware, con un total de unos 350 segundos aproximadamente. El mínimo se ha dado cuando se ha uso de token aware, iteración por columnas y se ha dado prioridad a token aware, con un total de 308 segundos aproximadamente.

El tiempo total de ejecución más bajo se ha dado en el caso dónde también se ha obtenido el mínimo en el tiempo de espera, que ha sido al utilizar token aware, iteración por columnas y prioridad a filas, con un total de 346 segundos aproximadamente.

Los resultados para el caso de utilizar 3000 filas y columnas para cada subconsulta ha sido:

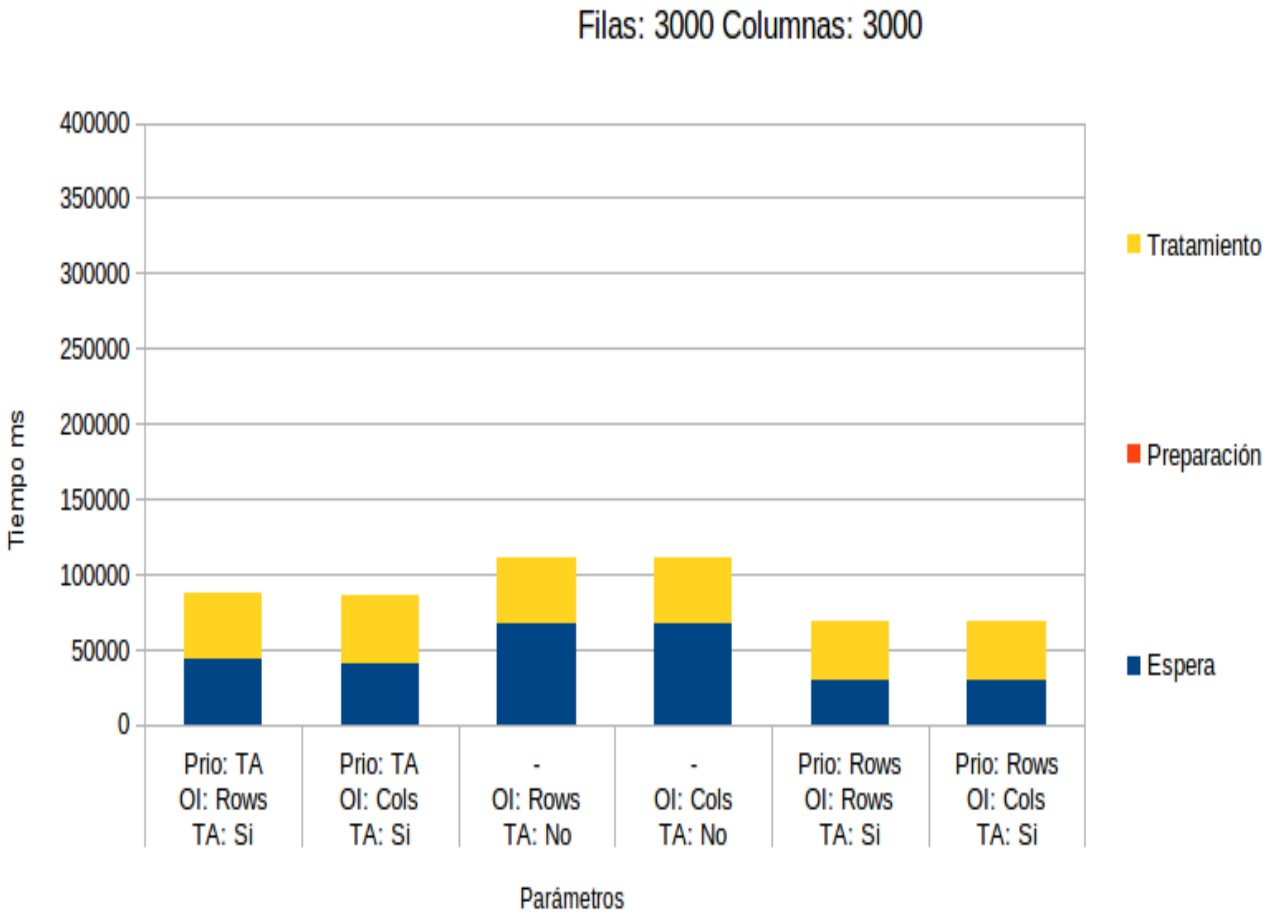

Figura 13: Tiempos rendimiento. Filas: 3000 y columnas: 3000

En este bloque de pruebas se han obtenido tiempos de espera significativamente más bajos respecto a los obtenidos en el caso de 100 filas y columnas. Han sido de unos 30 segundos para los casos en los que se ha utilizado token aware y se ha dado prioridad a las filas, de unos 42 segundos para los que se ha dado prioridad a token aware y de unos 67 segundos para los casos en los que no

se ha utilizado token aware.

Los tiempos de tratamiento y de preparación, sin embargo, han sido prácticamente constantes respecto al caso de 100 filas y 100 columnas.

Los tiempos totales de ejecución más bajos se han dado para los casos en los que se ha utilizado token aware y se ha dado prioridad a las filas, con un tiempo total de ejecución de unos 68 segundos.

Una vez vistas las diferencias de tiempos entre ambas gráficas, se puede decir que el tiempo de preparación y de tratamiento dependen del total de los datos a tratar y no del tamaño de las subconsultas. La única diferencia apreciable en el tiempo de preparación está en los casos en los que no se utiliza token aware. En estos casos la preparación de las filas es más simple y por lo tanto, los tiempos de preparación, por lo general, suelen ser la mitad respecto a los casos en los que

se utiliza token aware.

En cuanto al tiempo de espera, se puede decir que cuanto menos se divida el total de datos a tratar, menor será este tiempo, ya que la repartición de trabajo para los threadQ será mucho más rápida.

En cuanto a los tiempos medios de threadQ, los resultados obtenidos para el caso de utilizar 100 filas y columnas, son los representados por la siguiente figura:

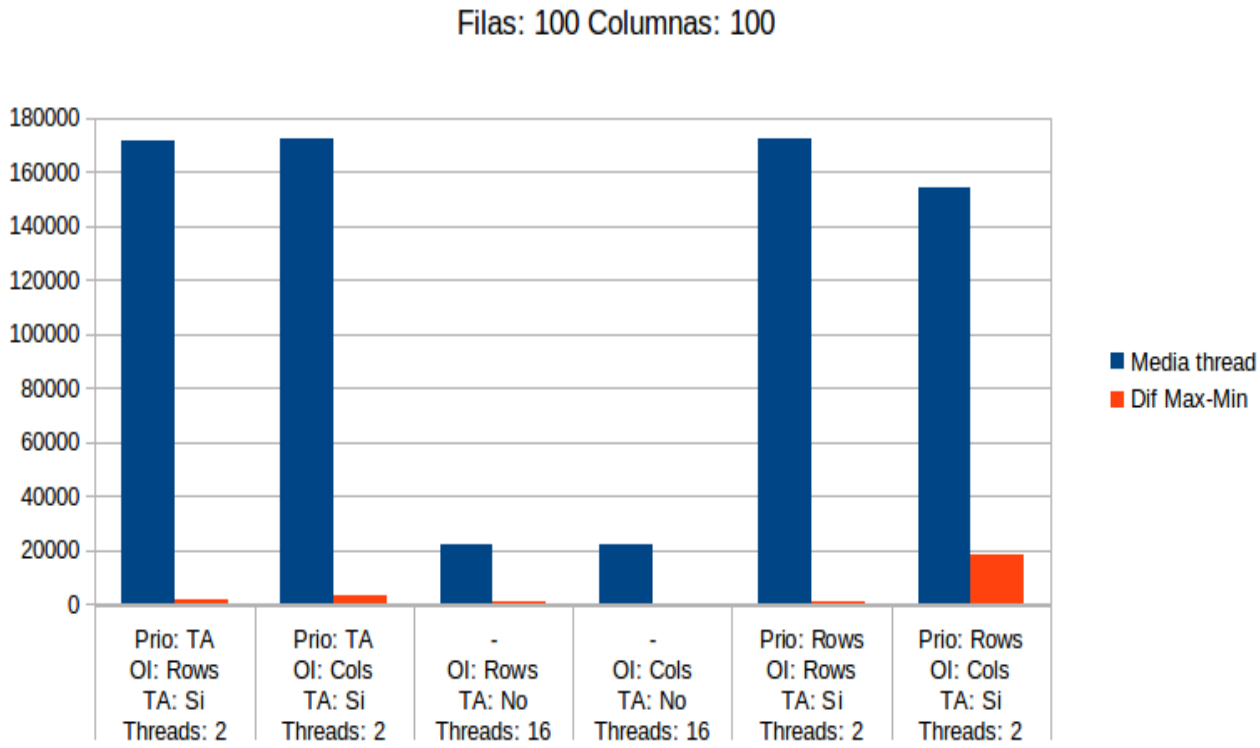

Figura 14: Media threadQ. Filas: 100 y columnas: 100

Los tiempos medios de consulta de los threadQ han sido más elevados para los casos en los que se ha hecho uso de token aware, ya que el número de threadQ era de 2, en cambio para los casos en los que no se ha utilizado token aware, el número de threadQ ha sido de 16 y por lo tanto, la media ha dado bastante menor.

El trabajo entre threadQ ha sido bastante equitativo. Se puede observar en las columnas representadas de color naranja.

En cuanto a los tiempos medios de threadQ obtenidos en el caso de utilizar 3000 filas y columnas, los resultados son:

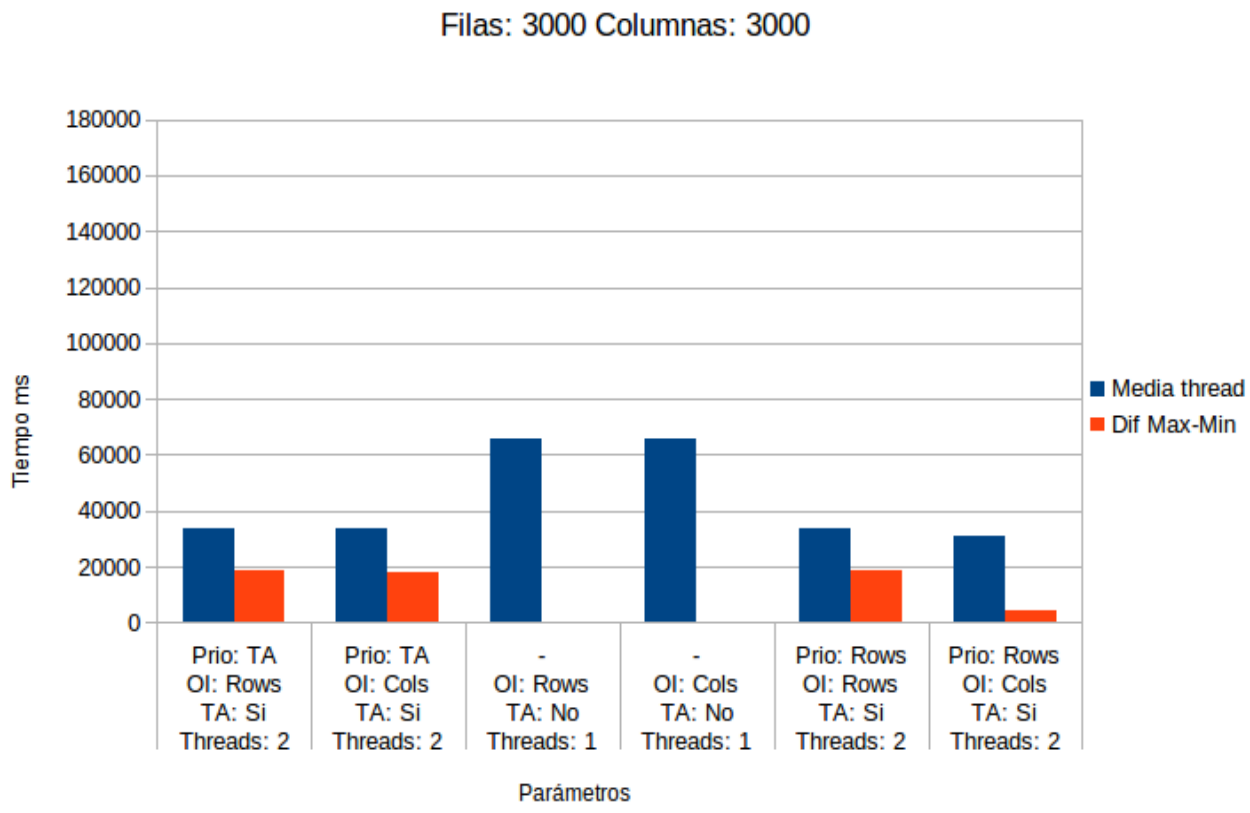

Figura 15: Media threadQ. Filas: 3000 y columnas: 3000

Se puede observar que los tiempos medios de threadQ para los casos en los que no se utiliza token aware han sido los más altos. Esto es debido a que solo se ha hecho uso de 1 threadQ a causa de que el número de filas y columnas por subconsulta ha sido de 3000, es decir, el total. Para los demás casos, se ha reducido a causa del incremento en el número de columnas a 3000.

Una vez vistos los tiempos de threadQ para el caso de 100 filas y columnas y 3000 filas y columnas, se puede decir que los tiempos medios son más bajos cuanto mayor es el número de threadQ a utilizar, ya que el trabajo se reparte más.

A continuación se muestran otros dos bloques de pruebas de rendimiento. El primer bloque de pruebas ha sido realizar una consulta con 100 filas y 1500 columnas para cada subconsulta. Los parámetros de rendimiento se han configurado con las combinaciones comentadas al inicio del capítulo.

Los tiempos son los mostrados en la siguiente figura:

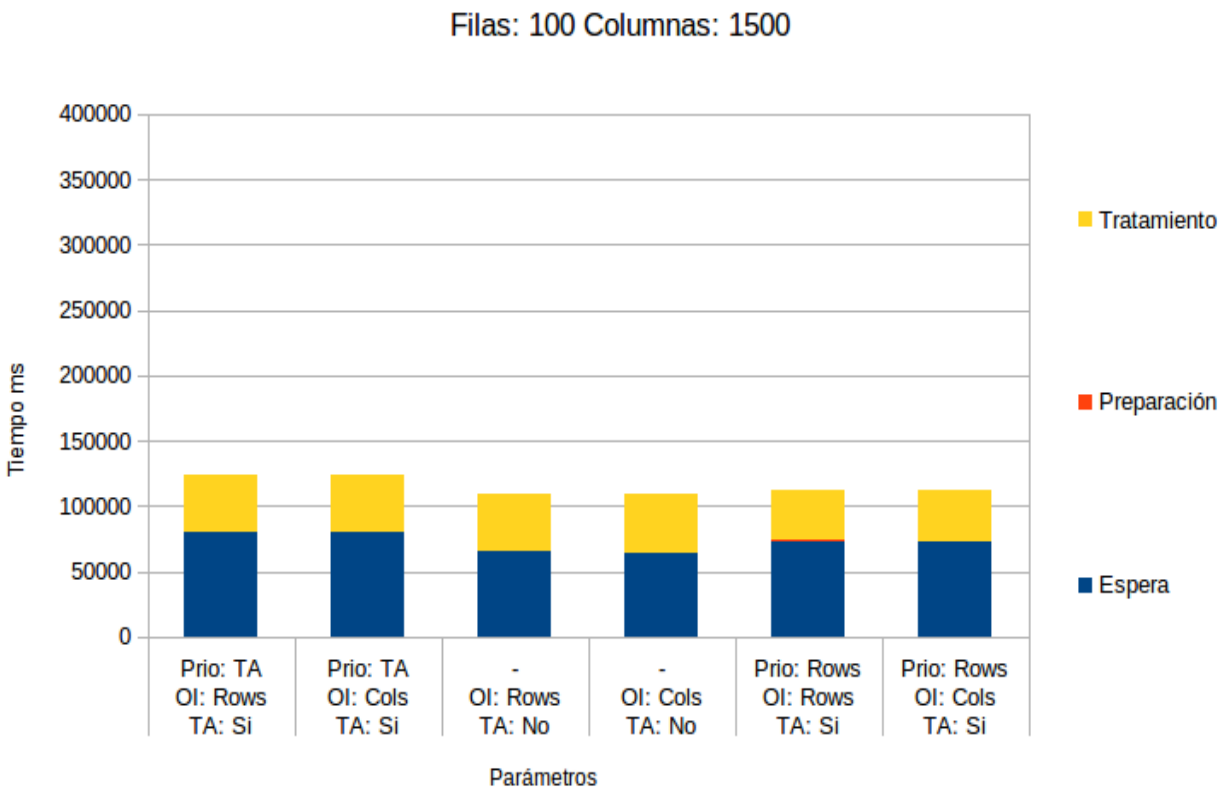

Figura 16: Tiempos rendimiento. Filas: 100 y columnas: 1500

Los tiempos de tratamiento al igual que los tiempos de preparación, se han mantenido prácticamente constantes, ya que el volumen total de datos sigue siendo el mismo que para los dos ejemplos mostrados anteriormente.

De nuevo el tiempo predominante ha sido el de espera, pero esta vez ha sido significativamente inferior respecto al caso en el que se han utilizado 100 columnas. Esto ha sido a causa de que los

bloques a repartir de trabajo eran mayores, han pasado de 100 columnas a 1500 columnas, y ha tenido una repercusión directa en el tiempo.

Los tiempos de ejecución total mínimos se han dado para los casos en los que no se ha utilizado token aware, con un total de unos 108 segundos aproximadamente.

Los tiempos medios de los threadQ han sido los siguientes:

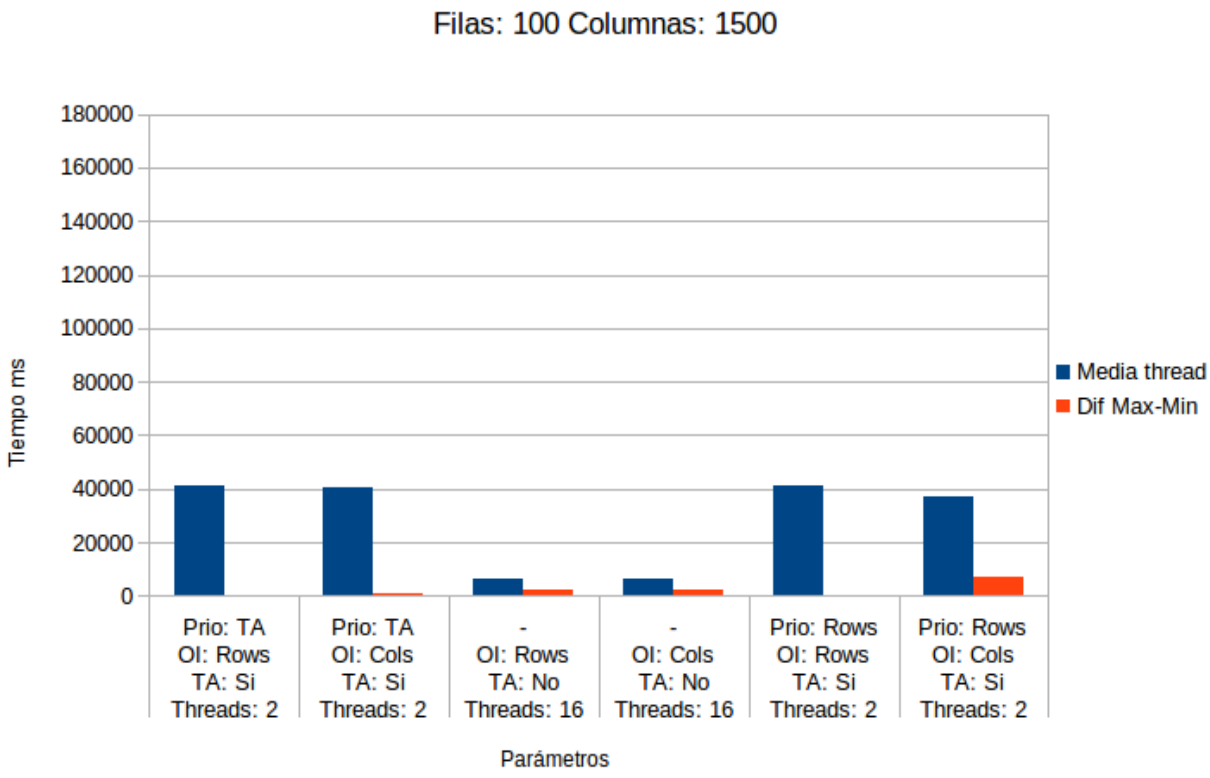

Figura 17: Media threadQ. Filas: 100 y columnas: 1500

En este caso la media de tiempos ha sido significativamente inferior que en el caso en el que se han utilizado 100 columnas. La diferencia es del orden de unos 120 segundos para los casos en los que se ha utilizado token aware y de unos 20 segundos para los casos en los que no se ha utilizado token aware, y el trabajo ha vuelto a ser bastante equitativo entre threadQ.

Para el siguiente bloque, el cual consiste en 100 filas y 3000 columnas por cada subconsulta, se han obtenido los resultados mostrados en la siguiente figura:

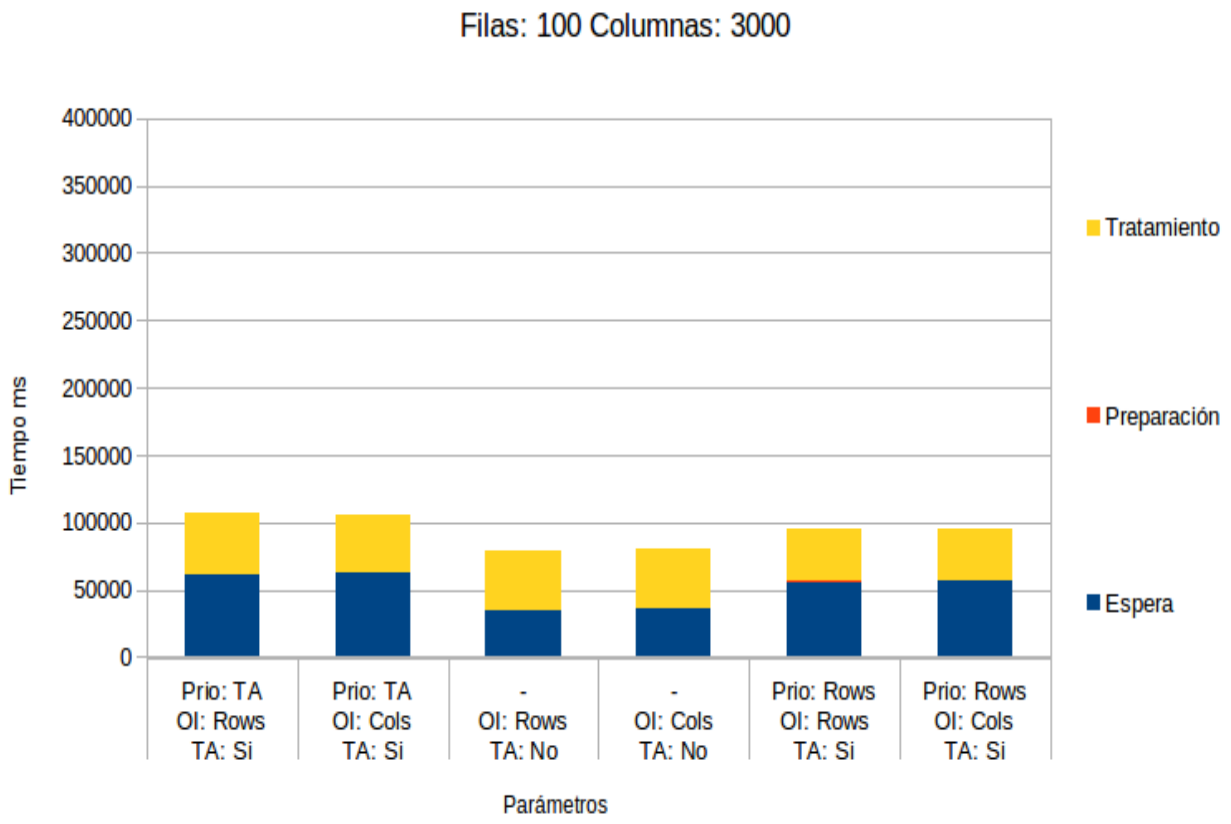

Figura 18: Tiempos rendimiento. Filas: 100 y columnas: 3000

En este caso los tiempos de tratamiento y preparación han vuelto a ser prácticamente iguales que para los dos casos anteriores.

En cuanto al tiempo de espera, ha sido inferior que el caso anterior. De nuevo el incremento en el número de columnas, esta vez a 3000, ha tenido una repercusión directa sobre el tiempo. Las reducciones de tiempo han sido aproximadamente de unos 30 segundos para los

casos en los que no se ha utilizado token aware y de unos 20 segundos para los casos en los que sí se ha utilizado.

Para este caso los tiempos totales de ejecución mínimos se han dado para las opciones en las que no se ha hecho uso de token aware, con un total de 80 segundos aproximadamente.

En cuanto a los tiempos medios de threadQ, los resultados han sido los siguientes:

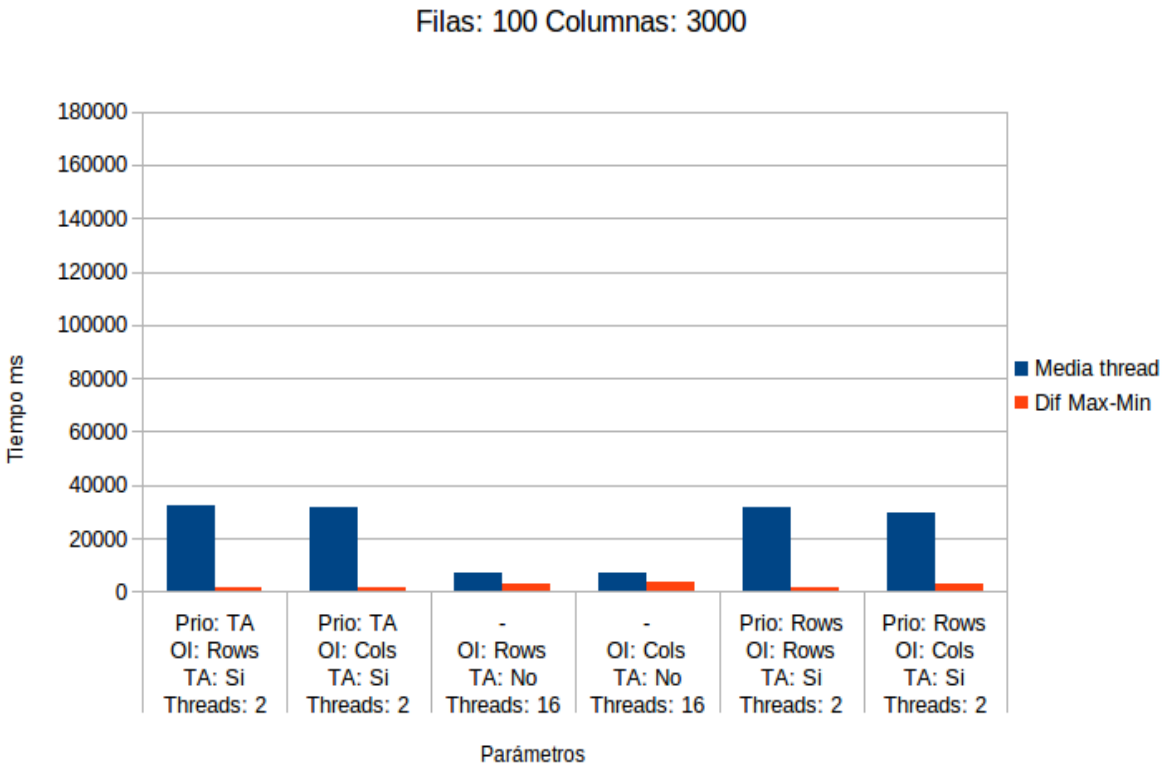

Figura 19: Media threadQ. Filas: 100 y columnas: 3000

Se ha conseguido reducir la media en unos 10 segundos para los casos en los que se ha utilizado token aware. Para el caso en el que no se ha utilizado token aware el tiempo ha sido aproximadamente el mismo.

Después de observar los resultados obtenidos en las distintas pruebas de rendimiento, se puede decir que los tiempos de preparación y de tratamiento dependen directamente del volumen de datos total a tratar, independientemente del tamaño de las subconsultas.

Por lo general, cuantos más threadQ se tengan, menores serán los tiempos de espera ya que se explota más el trabajo concurrente. El número de filas y columnas a utilizar para cada subconsulta también tiene un impacto directo sobre el tiempo de espera, dándose tiempos más bajos cuando el número de filas y columnas son más elevados. El hecho de que éstos sean los factores

determinantes para la reducción de tiempo en las pruebas realizadas, viene relacionado con el pequeño tamaño de la base de datos y de la consulta a realizar.

Al estar, la base de datos, formada únicamente por dos nodos, la opción de token aware ha tenido poca influencia. Se obtendrían mejores resultados en clusters formados por un número más elevado de nodos, ya que el número de threadQ sería más elevado y por lo tanto, se reducirían significativamente los tiempos de espera.

5. Conclusiones

Se ha desarrollado un mecanismo para optimizar automáticamente, en función de unos parámetros de rendimiento, las consultas a la base de datos Apache Cassandra. Los resultados obtenidos en las pruebas funcionales han sido los esperados. Se ha conseguido explotar la localidad de los datos, es decir, que el mecanismo pregunte directamente a los nodos, que físicamente almacenan los datos.

Como ventaja sobre el cliente de Apache Cassandra, cabe destacar que permite realizar la consulta de múltiples filas, pudiéndole indicar cuáles son las que desean ser consultadas. Lo mismo sucede con las columnas. Además de la ya comentada explotación de la localidad de los datos.

Se ha visto que los tiempos de espera se pueden reducir significativamente utilizando valores para las filas y las columnas, a utilizar para cada subconsulta, próximos al volumen total de datos. Además, la concurrencia también es un factor determinante en el tiempo de espera, dándose tiempos menores cuando el número de threadQ es más elevado.

Los tiempos de preparación y de tratamiento van directamente ligados al volumen total de datos a tratar.

En el caso de que se hiciera uso de dicho módulo en un sistema en una red con cuellos de botella, o una red muy sobrecargada en la que interesara minimizar las comunicaciones, los resultados serían muy positivos. Con la opción del token aware, se lograría minimizar el tráfico de red, ya que cada nodo podría responder sin necesidad de preguntar a los demás, y por lo tanto los tiempos

de ejecución serían menores.

El trabajo futuro que se debería realizar, consistiría en hacer que los threadQ trabajasen exclusivamente para un nodo en caso de utilizar token aware. De esta forma se evitarían posibles interferencias entre ellos en caso de hacer consultas a un mismo nodo. Con esta mejora se podrían reducir los tiempos de threadQ y en consecuencia los tiempos de espera.

6. Bibliografía

[1] Apache Cassandra, Wikipedia

http://en.wikipedia.org/wiki/Apache_Cassandra

[2] Hector Client for Apache Cassandra, Datastax, 2010, Documento PDF

www.datastax.com/sites/default/files/hector-v2-client-doc.pdf

[3] Apache Cassandra in Action, Jonathan Ellis, Transparencias

http://assets.en.oreilly.com/1/event/55/Apache%20Cassandra%20in%20Action%20Presentation.pdf

[4] Apache Thrift, Andrew Prunicki, 2009, Página web

http://jnb.ociweb.com/jnb/jnbJun2009.html

[5] Apache Cassandra, Web oficial

http://cassandra.apache.org/

[6] Consider the Apache Cassand DB, Srinath Perera, 2012, Documento PDF

http://www.ibm.com/developerworks/library/os-apache-cassandra/os-apache-cassandra-pdf.pdf

[7] Apache Cassandra 2.0 Documentation, Datastax, 2013, Documento PDF

www.datastax.com/documentation/cassandra/2.0/pdf/cassandra20.pdf

[8] Distributed Query Optimization, Alberto Abelló y Oscar Romero, UOC, Documento PDF

[9] A script to easily create and destroy an Apache Cassandra cluster on localhost, Sylvain Lebresne, Github, Página web

www.github.com/pcmanus/ccm

[10] The SimpleThreads Example, Oracle, Página web

www.docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

[11] Hector: A high level Java client for Apache Cassandra, Web oficial

www.hector-client.github.io/hector/build/html/index.html