Facultad de Ciencias

Especificación y ejecución de benchmarks sobre el gestor de datos distribuido y escalable Cassandra para aplicaciones de uso intenso de datos en la Industria 4.0

Benchmark specification and execution on Cassandra, a distributed and scalable data base, for intensive-data applications in the Industry 4.0

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Ricardo Dintén Herrero

Directora: Marta Elena Zorrilla Pantaleón

Junio 2019

Agradecimientos

En primer lugar, me gustaría dar las gracias a mis padres y amigos todo el apoyo que me han dado a lo largo de estos cuatro años de grado. También a mi novia, que ha estado a mi lado apoyándome y animándome siempre que lo he necesitado e incluso cuando no.

Además, me gustaría dar las gracias al departamento de ingeniería informática y electrónica de la Universidad de Cantabria, que me ha permitido, a través de una beca de colaboración, participar con ellos en un proyecto de investigación y, me ha proporcionado un lugar para desarrollar este trabajo y acceso al laboratorio de ISTR de la facultad de ciencias. Y quiero dar las gracias, en especial, a Marta Zorrilla, que me ha ayudado y orientado a lo largo del desarrollo de este proyecto y la colaboración con el grupo de investigación.

Resumen

En la actualidad, estamos inmersos en la llamada cuarta revolución industrial o también conocida industria 4.0. Esta tiene por objeto revolucionar la industria de la fabricación y producción gracias a la digitalización de los equipos del entorno industrial, la computación en la nube, la integración de los datos y los avances tecnológicos de los sistemas de producción y fabricación. Las plataformas de tercera generación, basadas en arquitecturas centradas en el dato y la computación en la nube, son propuestas para vertebrar este cambio tecnológico. El grupo de investigación ISTR de la Universidad de Cantabria está desarrollando una propuesta arquitectónica denominada P3forI4 basada en los frameworks del ecosistema Apache. En este trabajo, se estudia y evalúa el gestor de bases de datos distribuido y escalable Apache Cassandra para comprobar su idoneidad y su comportamiento para formar parte de la capa de persistencia de esta arquitectura. Para ello se diseñan y realizan un conjunto de pruebas (benchmark) con distintas configuraciones y cargas de trabajo para conocer las condiciones límite de operación y extraer conclusiones sobre consideraciones de diseño para su adecuación como sistema de persistencia para aplicaciones reactivas y entornos con consideraciones de tiempo real.

Palabras clave:benchmark, NoSQL, big data, industria 4.0

Abstract

Nowadays we are immersed in the fourth industrial revolution, also known as Industry 4.0. This is aimed at revolutionizing the manufacturing and production industry thanks to the digitalization of the equipment of the industrial environment, the cloud computing, the integration of the data and the technological advances of the production and manufacturing systems.The thrid generation platforms based on data-centric architectures and cloud computing are proposed to vertebrate this technological change. ISTR, a research group of the University of Cantabria, is currently developing an architectural proposal named P3forI4 based on the Apache ecosystem frameworks. In this project, we study and evaluate Cassandra, a distributed and scalable data base, to check its suitability and its behavior to be considered as an element of the persistence layer of this architecture. Therefore, we design and perform a set of benchmarks with different configurations and workloads to know the limits of the system and draw conclusions about design guidelines to adapt it as a persistance system for reactive applications in real-time environments.

Key words: benchmark, NoSQL, big data, industry 4.0

Índice

Índice de figuras

Índice de *scripts*

Glosario

Big data: término que alude al almacenamiento y la gestión de una cantidad elevada de datos que se generan a gran velocidad y que presentan gran variedad (datos estructurados, semi-estructurados y no estructurados).

Benchmark: conjunto de pruebas que se ejecutan bajo diferentes configuraciones y cargas de trabajo para establecer comparaciones y medir rendimientos.

Computación en la nube: se refiere a la práctica de utilizar servidores remotos interconectados alojados en Internet para almacenas, gestionar y procesar información.

Ecosistema: conjunto de tecnologías relacionadas entre sí y que comparten un fin común.

Escalabilidad: capacidad de un sistema de crecer, puede ser tanto de manera horizontal (aumentando el número de nodos), como vertical (aumentado la potencia de cómputo del nodo).

Familia de columnas: paradigma bajo el cuál se catalogan gestores de base de datos cuya estructura se basa en un conjunto de tuplas clave-valor, donde la clave se asigna a un valor que es un conjunto de columnas.

IIoT: acrónimo de Internet Industrial de las cosas, un concepto que se refiere a la conexión entre personas, datos y máquinas relacionados con la actividad industrial.

IoT: acrónimo de Internet de las cosas, un concepto que se refiere a la conexión entre objetos físicos como sensores o máquinas e Internet.

Latencia: tiempo de respuesta ante una petición. Es el tiempo que discurre desde que un cliente lanza una petición contra un servidor hasta que recibe la respuesta.

Partición: segmento de datos agrupados en un mismo nodo de un clúster.

Plataforma de tercera generación: son aquellas en las que la carga de trabajo proviene de dispositivos móviles inteligentes, soportados por una infraestructura en la nube. Está basado en cuatro pilares: *big data* y *analitycs*, computación en la nube, computación móvil y social.

Sistema ciber-físicos: es todo aquel dispositivo que integra capacidades de computación, almacenamiento y comunicación para controlar e interactuar con un proceso físico.

Throughput: número de operaciones por segundo que ejecuta un cliente o, que son atendidas por un servidor. En este documento se usa la segunda acepción.

1 Antecedentes y objetivos

Este trabajo fin de grado se desarrolla en el marco del proyecto del plan Nacional "Sistemas Informáticos Predecibles y Confiables para la Industria 4.0"(TIN2017- 86520-C3-3 R) del grupo de investigación Ingeniería de Software y Tiempo Real (ISTR) de la Universidad de Cantabria. Dentro de este, se está desarrollando una propuesta de plataforma de tercera generación [11] para dar soporte a la Industria 4.0.

El término Industria 4.0 hace referencia a la cuarta revolución industrial (ver Ilustración 1), una nueva etapa en la organización y el control de la cadena de valor a lo largo del ciclo de vida de un producto [12]. La Industria 4.0, por tanto, persigue la transformación del modelo de automatización tradicional con estructura piramidal a un modelo basado en servicios interconectados que combina la tecnología operacional (OT) y las tecnologías de la información (IT) [13]. Su objetivo es alcanzar la operación requerida con una mejora cualitativa en la automatización y optimización de los procesos industriales, satisfaciendo además, la creciente demanda global de productos personalizados. Este modelo se basa en la ubiquidad y conectividad de los datos, las personas, los procesos, los servicios y los sistemas ciber-físicos que intercambian y explotan la información generada en cada nivel de la arquitectura para conseguir una producción descentralizada y adaptable a los cambios en tiempo real.

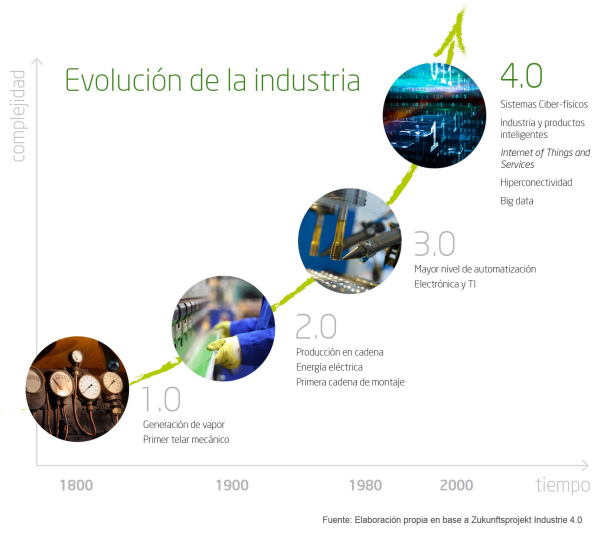

Ilustración 1: Evolución de la industria [6]

En este escenario, un entorno industrial 4.0 estará compuesto de un gran número de fuentes (sensores, aplicaciones, sistemas ciberfísicos, etc.) que generarán un ingente volumen de datos que debe ser almacenado, procesado y analizado con diferentes restricciones temporales para aportar valor. El Big Data y la computación en la nube están actualmente considerados como habilitadores [13] para la construcción del ecosistema industrial del futuro, ya que son capaces de gestionar la variedad, velocidad y volumen de datos por

medio de arquitecturas altamente distribuidas y escalables que pueden redimiensionarse de manera dinámica para procesar cargas de trabajo en cuasi tiempo real.

Hay varios frameworks para procesar big data en una plataforma estándar, por ejemplo, construidas con herramientas del ecosistema Apache tales como Kaftka, Spark, Storm, etc. Sin embargo, no están optimizadas para emplearse directamente en un dominio industrial. Por esto, dentro de este proyecto nacional, se propone una arquitectura de referencia llamada P3forI4, diseñada bajo las siguientes premisas:

- Debe procesar los datos procedentes del entorno de modo reactivo y descentralizado.
- La distribución, heterogeneidad y escalabilidad de los recursos computacionales requeridos debe concebirse con un conjunto de servicios de *middleware*, con el objetivo de seguir una única estrategia para su gestión.
- La monitorización es una tarea esencial en este tipo de sistemas, puesto que su gestión se basa en estrategias dinámicas definidas por el nivel de uso de recursos y el estado de las operaciones.

La arquitectura se formula en base a tres principios básicos: El dato como servicio (DaaS), la plataforma como servicio (PaaS) y la monitorización como servicio (MaaS).

Actualmente se han definido los siguientes servicios (ver ilustración 2): servicio de serialización proporcionado por AVRO, servicio de distribución proporcionado por Zookeeper, servicio de comunicación proporcionado por Kafka, servicio de planificación proporcionado por Spark, servicio de persistencia proporcionado por Mem-SQL (en memoria), Apache Cassandra (basado en familias de columnas) y Neo4J(basado en grafos) y servicio de seguridad a través de mecanismos proporcionados por Zookeeper para el acceso a información compartida.

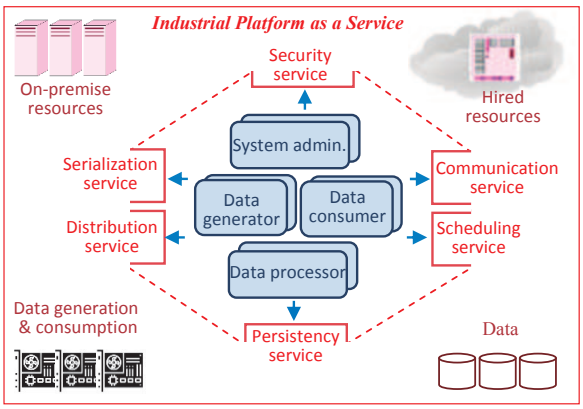

Ilustración 2: Plataforma industrial como servicio

Este trabajo surge como parte del desarrollo del servicio de persistencia. En él se pretende analizar el rendimiento de Cassandra, que forma parte del ecosistema Apache, para adaptarlo a las necesidades de la industria.

En este trabajo y teniendo en cuenta el marco de referencia en el que se realiza se pretenden alcanzar los siguientes objetivos:

- Analizar de rendimiento de bases de datos Cassandra bajo diferentes cargas de trabajo y en diferentes configuraciones del clúster.
- Estudiar de rendimiento de escrituras vs lecturas para diferentes estrategias de diseño de tablas en Cassandra.

- A partir de las conclusiones obtenidas de las pruebas realizadas para satisfacer los objetivos anteriores, establecer una serie de reglas o directrices que sirvan para guiar el proceso de diseño de una base de datos Cassandra, en general, y para una aplicación en un entorno de Industria 4.0 en particular.

2 Gestor de datos distribuido y escalable Apache Cassandra

Cassandra es un sistema gestor de bases de datos NoSQL escalable y de código abierto. Cassandra está pensado para manejar grandes cantidades de datos de tipo estructurado, semi-estructurado y no estructurado a lo largo de un conjunto de nodos interconectados. Sus principales características son la disponibilidad, la escalabilidad lineal y la simplicidad funcional a través de varios servidores sin puntos de fallo únicos. Está basado en estructuras de datos simples para ofrecer flexibilidad y tiempos de respuesta bajos.

Cassandra basa su arquitectura en el uso de familias de columnas para almacenar los datos. Estas son similares a las tablas que se conocen de las bases de datos relacionales, pero a diferencia de estas, cada fila puede tener un número diferente de columnas. Además, no implementa restricciones semánticas ni integridad referencial. Por tanto, hace que las bases de datos sean más flexibles aunque relegando a las aplicaciones el control de la información que almacenan.

Los datos se distribuyen en particiones, que el propio sistema gestiona de manera automática y transparente al usuario, para asegurar que los datos quedan repartidos entre todos los nodos que conforman el sistema. Estos nodos se comunican mediante un protocolo P2P (Peer to peer), es decir, cualquier nodo puede contestar peticiones o realizar la tarea de coordinador. Además se puede configurar el nivel de replicación de los datos para que estén disponibles en varios nodos y mejorar la tolerancia ante fallos del sistema. Estas arquitectura favorece la disponibilidad, otra de las características que se han mencionado anteriormente, y los tiempos de respuesta bajos, ya que no es un único nodo el que tiene que recibir todas las peticiones.

En cuanto a la escalabilidad lineal, Cassandra se ha diseñado para funcionar de manera que cuando se necesite atender un mayor número de peticiones sea suficiente con aumentar el número de nodos, como se representa en la ilustración 3.

Otro de los principios de Cassandra relacionado con la disponibilidad es que se puede escribir siempre que haya al menos un nodo en el que almacenar la réplica. Esto ocurre por dos motivos. El primero es que la escrituras no se almacenan directamente en el disco, si no

Ilustración 3: Escalabilidad lineal de Cassandra[3]

que primero se guardan en memoria, en lo que se llama *Mem-Table*. Cuando la *Mem-Table* se llena se vuelca en las *SS-Table*, que son las estructuras de datos almacenadas en disco (*flush*) y posteriormente, son compactadas (compactación) para mejorar el rendimiento de las lecturas. Y el segundo de los motivos, es que no comprueba la existencia de la fila, si no que se produce lo que se conoce como *upsert*. Esto significa que se escribe la fila al final de la *Mem-Table* y, en caso de que ya existiera alguna con la misma clave, la sustituye.

A continuación, se explica el proceso de instalación de Cassandra, los ficheros de configuración para su despliegue en un clúster de varios nodos y las consideraciones de diseño básicas en relación con las bases de datos relacionales.

2.1. Instalación de Cassandra

Para la instalación de Cassandra existen dos opciones comúnmente usadas: instalación mediante paquetes e instalación *tarball*. En este caso se ha utilizado la instalación por paquetes. Para ello hay que añadir un nuevo repositorio a la aplicación apt y una clave para el mismo, actualizar y, por último, instalar. Como es una tarea que se debe repetir en cada nodo se ha llevado a cabo utilizando el siguiente *script*:

Script 2.1: Instalación y despliegue de Cassandra: script de instalación de Cassandra mediante paquetes

```
#!/bin/bash
# 1. Añadir el repositorio
echo "deb http://www.apache.org/dist/cassandra/debian 311x main"
    | sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
# 2. Añadir keys para el repositorio
curl https://www.apache.org/dist/cassandra/KEYS | sudo apt-key
   add -
sudo apt-key adv --keyserver pool.sks-keyservers.net --recv-key
   A278B781FE4B2BDA
# 3. Instalar con apt-get
sudo apt-get update
sudo apt-get install cassandra
```

2.2. Configuración del clúster

Antes de comenzar el proceso de configuración de un clúster, conviene aclarar los siguientes conceptos (ver Ilustración 4):

- **Nodo**: es donde se almacenan los datos. Es el elemento básico en la infraestructura de Cassandra.
- **Clúster**: es el conjunto de todos los nodos intercomunicados. Un clúster puede almacenar varios *keyspaces*.
- **Datacenter**: es una colección de nodos físicos o virtuales que constituyen un anillo Cassandra. A este nivel tiene lugar la replicación de los datos.
- **Rack**: es una subvidisión de nodos de nivel inferior al *datacenter*. Sirve para agrupar los nodos de un *datacenter* de forma que las réplicas se distribuyan en nodos pertenecientes a distintos *racks*.

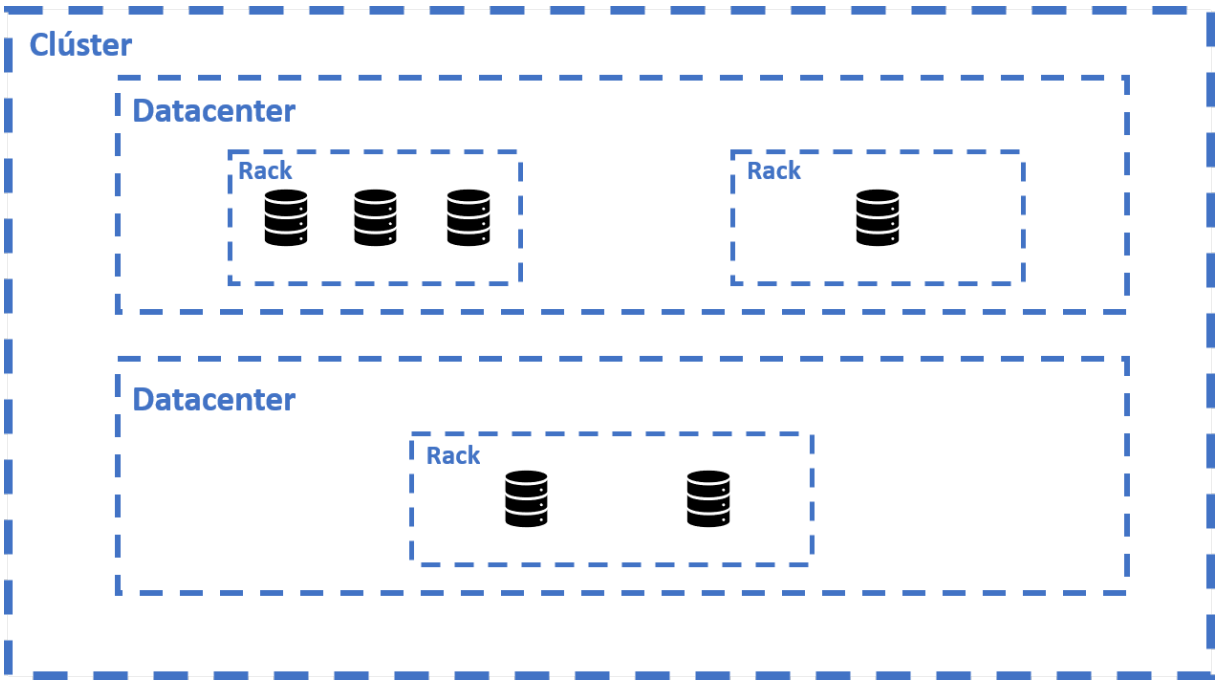

Ilustración 4: Ejemplo de estructura de un clúster en Cassandra

Una vez entendidos los conceptos y con Cassandra ya instalado en cada nodo, es el momento de configurar el clúster.Para ello es necesario detener las instancias de Cassandra (si estuvieran ejecutándose) y ajustar una serie de parámetros que se encuentran en los ficheros de configuración *cassandra.yaml*[5] y *cassandra-rackdc.properties*. Según el tipo de instalación que se haya realizado: por paquetes o *tarball*, se encontrarán en el directorio */etc/cassandra* o en *CASSANDRA_HOME/conf* respectivamente.

En primer lugar hay que modificar el fichero *cassandra.yaml*. En ese fichero se deben establecer los siguientes parámetros:

- **cluster_name**: Este parámetro es el nombre con el que se identificará el clúster. Puede ser cualquier nombre, pero deberá ser el mismo en todos los nodos.
- **seeds**: Este parámetro contiene las direcciones IP de los nodos 'semilla'. Estos son nodos conocidos que contienen información sobre la topología del clúster. Todos los nodos contienen esta información pero estos son conocidos por todos para agilizar el proceso de incorporación de nuevos nodos al clúster. Al igual que el cluster_name debe ser igual en todos los nodos. Este parámetro solo se debe cambiar al introducir un nodo en el clúster, una vez incorporado Cassandra lo gestiona de manera automática.
- **listen_address**: es la dirección IP del nodo Cassandra. Se utiliza para comunicaciones internas (de Cassandra a Cassandra). Este parámetro es diferente en cada nodo.
- **rcp_address**: es la dirección IP en la que Cassandra escuchará las peticiones de los clientes a través del protocolo CQL. Este parámetro es diferente en cada nodo.
- **endpoint_snitch**: es el "snitch" usado por Cassandra. El snitch es la estrategia que utiliza el clúster para conocer la estructura lógica establecida, esto es, en que datacenter y rack se encuentra cada nodo. Existen diferentes tipos de snitch que pueden utilizarse en función de las necesidades que se tengan. Para sistemas en producción se recomienda usar el snitch *GossipingPropertyFileSnitch*.

Una vez hecho esto, si se ha escogido el *GossipingPropertyFileSnitch* se deben cambiar los parámetros del fichero *cassandra-rackdc.properties*. Los parámetros son:

- **dc**: Este parámetro indica en que datacenter se encuentra el nodo. Como ocurría con el parámetro *cluster_name*, puede tomar cualquier valor, pero tiene que ser común en todos los nodos que formen parte del mismo datacenter.

- **rack**: Este parámetro indica el rack en el que se encuentra el nodo dentro de un datacenter. Al igual que en el parámetro *dc*, puede tomar cualquier valor siempre que sea común a todos los nodos de un mismo rack.

Por último, cuando ya se ha terminado de modificar todos los ficheros hay que eliminar el contenido del directorio *var/lib/cassandra/data/system* para que los cambios tengan efecto. Y también el fichero *cassandra-topology.properties*, ya que si este fichero existe Cassandra los toma como referencia y queremos que genere uno nuevo con los datos del fichero *cassandra-rackdc.properties* modificado previamente. Para ello, habrá que ejecutar los siguientes comandos:

```
rm -fr /var/lib/cassandra/data/system/*
rm /etc/cassandra/cassandra-topology.properties
```

Con esto el clúster queda configurado y solo es necesario iniciar el servicio Cassandra en cada nodo, comenzando por los que hemos designado como *seeds*. Cuando todos están iniciados se puede comprobar que todo funciona correctamente mediante el comando

```
nodetool status
```

Ejemplo de configuración

Para la ejecución de los benchmarks que se describirán en capítulos posteriores ha sido necesario configurar un clúster de 1 nodo y un clúster de 3 nodos. A continuación, se mostrarán los parámetros empleados para configurar el clúster de 3 nodos en un solo rack (Script 2.2, 2.3, 2.4 y 2.5) y la respuesta de nodetool una vez arrancado el sistema (Ilustración 5). En esta imagen se puede observar las IP del nodo con la instancia Cassandra arrancada, el número de tokens establecidos para el algoritmo hash utilizado por el *partitioner* y la carga de datos que reside en cada nodo.

Script 2.2: Instalación y despliegue de Cassandra: cassandra.yaml | Nodo 1

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.27,172.16.31.68"
listen_address: "172.16.31.58"
rcp_address: "172.16.31.58"
endpoint_snitch: GossipingPropertyFileSnitch
```

Script 2.3: Instalación y despliegue de Cassandra: cassandra.yaml | Nodo 2

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.27,172.16.31.68"
listen_address: "172.16.31.68"
rcp_address: "172.16.31.68"
endpoint_snitch: GossipingPropertyFileSnitch
```

Script 2.4: Instalación y despliegue de Cassandra: cassandra.yaml | Nodo 3

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.27,172.16.31.68"
listen_address: "172.16.31.46"
rcp_address: "172.16.31.46"
endpoint_snitch: GossipingPropertyFileSnitch
```

Script 2.5: Instalación y despliegue de Cassandra: cassandra-rackdc.properties

```
dc=datacenter1
rack=rack1
```

```
gestor@CZC731837D:~$ nodetool status
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.31.16.46  364,13 MiB  256          65,1%             7954c500-6d8f-40ff-a8c4-f1b4cb668ffd  rack1
UN  172.31.16.27  426,32 MiB  256          68,7%             8a6cc91f-1c6c-438e-81e6-13622e4d27c6  rack1
UN  172.31.16.68  370,1 MiB  256          66,1%             d0b72749-1bda-41c6-b5de-61f31fde1eab  rack1

gestor@CZC731837D:~$
```

Ilustración 5: Nodetool status: tras configurar el clúster

2.3. Consideraciones de diseño y diferencias con el sistema relacional

El gestor Apache Cassandra no pertence al paradigma relacional y, por tanto, a la hora de diseñar hay que tener en cuenta una serie de características que lo diferencian de estos:

- En Cassandra, como en el resto de sistemas NoSQL, no existe el concepto de integridad referencial o de "joins". Esto lleva a que sea necesario almacenar la información de forma redundante. Además, esto tiene como consecuencia que la aplicación cliente es la encargada de velar por la integridad y la consistencia de los datos.
- El diseño tiene que hacerse basado en las consultas que se van a realizar, creando una tabla específica para cada caso.
- Las consultas solo pueden realizarse por igualdad de la *primary key*. Esto hace de la elección de la clave algo determinante, ya que si no se escoge la clave correcta no se podrá realizar la consulta.
- La *primary key* está compuesta de dos campos: *partition key* y *clustering key*. El primero, define como se distribuyen los datos en particiones y el segundo, el orden de las filas dentro de una misma partición. Por ello, para poder realizar consultas es necesario indicar todos los campos de la *partition key*. Si no se indican todos los campos de la *partition key*, el sistema lanza una advertencia de que se debe consultar varias particiones y no realiza la consulta. Es posible forzar este tipo de consultas pero no es para nada recomendable porque el rendimiento se ve muy mermado.

3 Herramientas de Benchmark

En este capítulo, se introducirá el concepto de benchmark, qué se espera obtener con su ejecución y qué herramientas se han utilizado para este proyecto en concreto. Por cada una de las herramientas se presentarán sus principales características y algunas ventajas y limitaciones.

3.1. Concepto de Benchmark y Objetivos

Desde la década de los 90, la industria y las organizaciones son conscientes de la importancia de desarrollar benchmarks estándar que permitan comparar sistemas y aplicaciones de manera objetiva, persiguiendo también que éstos sean simples, portables y escalables.

La mayor parte de los benchmarks permiten comparar el rendimiento de diferentes sistemas bajo unas condiciones experimentales específicas. Esto ayuda, por un lado, a los proveedores a posicionar su producto respecto a sus competidores y, por otro lado, a los usuarios a tomar decisiones estratégicas basadas en información objetiva.

Deben destacarse los esfuerzos de las organizaciones como Transaction Processing Performance Council (TPC), enfocados en medir el rendimiento de sistemas gestores de bases de datos transaccionales (TPC-C, TPC-E) y analíticas (TPC-H, TPC-DS).

Debido a la importancia que están alcanzando las arquitecturas big data, en 2014, TCP publicó TCPx-BB con el objeto de medir el rendimiento de sistemas Big Data basado en Hadoop. Este sigue el mismo pocedimiento que sus benchmarks anteriores. Replica la ejecución de 30 consultas analíticas en un contexto de venta al por menor aplicando diferentes configuraciones.

Esto no era válido para nuestros propósitos, ya que está pensado para sistemas basados en Hadoop, y Cassandra no sigue este paradigma. Por ello, para la realización de este trabajo fue necesario buscar herramientas que generaran cargas de trabajo configurables y ofrecieran métricas de rendimiento. Se han usado principalmente dos sistemas que se describirán más adelante: YCSB y Cassandra-stress. El primero de ellos, es más genérico y permite medir algunos parámetros en diferentes sistemas gestores pero presenta algunas limitaciones que hacen que no sea del todo adecuado para este proyecto. Por ello, fue necesario utilizar el segundo, Cassandra stress, que es especifico de Cassandra y permite analizar el rendimiento de diferentes consultas sobre esquemas distintos. El objetivo de estos benchmark es obtener información del rendimiento de Cassandra a través de los siguientes parámetros:

- **latencia**: es el tiempo de respuesta de una petición, es decir, desde que se ejecuta la sentencia hasta que el sistema la confirma.

- **throughput**: es el n° de operaciones por segundo que atiende el gestor.

3.2. YCSB

En este apartado se describe YCSB[10] (Yahoo Cloud System Benchmark), un software libre para medir el rendimiento y escalabilidad de diferentes sistemas gestores de bases de datos. De YCSB se expondrán sus características, posibilidades y limitaciones, así como el proceso que se ha seguido para diseñar y ejecutar las pruebas en él.

3.2.1. Características

YCSB es un sistema diseñado para generar una determinada carga de trabajo a un sistema gestor de bases de datos, de manera que se pueda medir su rendimiento (latencia y *throughput*). Este sistema nos permite trabajar con un gran número de sistemas gestores de bases de datos, entre ellos Cassandra. Funciona generando una carga de trabajo a un n° de operaciones por segundo establecido y reportando al final datos sobre la ejecución. Estos datos son: duración de la prueba, n° de operaciones ejecutadas, *throughput*, latencia mínima, máxima, media y percentiles 90, 95, 99 y 99,9 de cada tipo de operación y número de operaciones completadas con éxito. El formato de estos resultados es configurable y se presenta en forma de histograma o de series temporales. Inicialmente, el sistema presenta una conjunto de cargas de trabajo para medir las prestaciones de los sistemas, pero es posible crear nuevas cargas de trabajo mediante la especificación de ficheros en formato *properties* de java.

Estas pruebas de rendimiento las realizan en base a lo que llaman *workloads* (ver Anexo A). Las diferentes *workloads* están conformadas por una serie de parámetros configurables. Con estos parámetros el sistema genera datos y *queries* para simular un trabajo real en la base de datos. Las *workloads* se pueden configurar para realizar las cuatro principales operaciones en bases de datos: lecturas, escrituras, actualizaciones y lectura por rangos de clave (*scan*). Además para la elección de las filas con las que interactuar se necesita definir la distribución de probabilidad que debe emplear el sistema. YCSB dispone de las siguientes:

- **Uniform**: Escoge una fila con una probabilidad uniforme. Esto quiere decir que todas las filas tienen la misma probabilidad de ser elegidos.
- **Zipan**: Escoge una fila de acuerdo a la distribución zipfian. Por ejemplo, cuando se escoge una fila, algunas serán extremadamente populares (la cabeza de la distribución) mientras la mayor parte no serán populares (la cola).
- **Latest**: Funciona como la distribución zipfian pero en este caso las filas más recientes son los que se encuentran en la cabeza de la distribución.
- **Multinomial**: La probabilidad puede ser especificada para cada elemento. Por ejemplo, se podría asigna una probabilidad de 0.95 a las operaciones de lectura, 0.05 a las operaciones de actualización y 0.00 a las operaciones de scan y escritura.

Además de los tipos de operación que se van a realizar y la distribución de probabilidad, se pueden especificar el número de hilos cliente que van a generar trabajo contra la base de datos, el throughput objetivo de la prueba, el número de operaciones y la duración máxima, entre otros. YCSB cuenta con 5 *workloads* genéricas basadas en simular 5 escenarios: alto porcentaje de actualizaciones, alto porcentaje de lecturas, solo lecturas, lecturas más recientes y rangos cortos (pocas filas).

Con todo lo anterior podemos ver que YCSB es un sistema de benchmark con un gran potencial pero, como toda herramienta presenta ventajas y limitaciones.

Ventajas

Una de las principales ventajas de YCSB es que es un sistema de código abierto, con un repositorio en GitHub lo cual permite que el sistema tenga una comunidad que participa activamente en el desarrollo y mantenimiento del sistema, así como solucionando dudas de otros usuarios.

Otra de sus ventajas es que permite analizar el rendimiento de diferentes tecnologías con una interfaz única, lo cual permite comparar gestores documentales, gestores basados en familias de columnas o gestores relacionales de manera sencilla. Además, no te exige conocer el lenguaje de interrogación propio de cada tecnología, ya que indicando una serie de parámetros genera las peticiones de manera automática.

En cuanto a los resultados de los análisis, YCSB genera, por defecto, ficheros de texto plano con una estructura que permite entender fácilmente el contenido e incluso procesarlo mediante algún tipo de *script*. Pero también permite cambiar el *exporter* para, por ejemplo, obtener los resultados en formato JSON.

Por último, si todo esto no fuera suficiente o no se adaptase a las necesidades de algún usuario, se puede extender implementando las diferentes interfaces en lenguaje de programación Java.

Limitaciones

Durante la realización de este trabajo he encontrado una serie de limitaciones que presenta YCSB, y que hacen que no se puedan simular situaciones tan realistas como se esperaba en un comienzo. A pesar de su gran número de parámetros de configuración, el resultado final se ve afectado por el hecho de no poder emplear cualquier tipo dato en las columnas de las tablas, ya que solo funciona para campos de tipo texto. Es cierto que se puede configurar la longitud de los campos para simular el tamaño de un número entero o de un número de coma flotante, pero si se hace esto solo se tendrían campos de ese tamaño en la tabla. Esto hace que no se puedan representar situaciones reales en las que en una misma tabla conviven columnas de tipo entero, de tipo texto, de coma flotante, fechas, etc.

Una de sus principales ventajas es también una limitación, ya que al no escribir las sentencias de manera explícita se pierde control sobre la ejecución del programa y no se pueden modelar de manera precisa los casos de uso. Por tanto, puede ser útil si no se domina el lenguaje o si se quiere comparar con otros sistemas pero resulta limitante para hacer un análisis exhaustivo de una tecnología en concreto.

Otra de sus limitaciones importantes es que las claves primarias no pueden ser compuestas (al menos en el caso de Cassandra). YCSB identifica el campo de la clave primaria por el nombre y_id y, por tanto, si la clave es compuesta falla a la hora de generar consultas. Esto es realmente limitante en Cassandra, ya que el proceso de elección de la clave primaria es el más importante sin duda a la hora de diseñar la base de datos, debido a que determina como se reparten los datos en los diferentes nodos y afecta en gran medida al rendimiento.

Aparte de eso, YCSB necesita un entorno de ejecución con Java[7], Maven[8] y Python 2.7[9]. Es importante utilizar la versión correcta de Python ya que no todas las versiones son compatibles entre sí y, por ejemplo, con la versión 3 no funciona. Todo esto implica un proceso de configuración que no es complicado pero si puede llegar a ser tedioso.

Por último, en mi caso en particular, me he encontrado con problemas a la hora de generar cargas de trabajo. En determinadas situaciones el cliente no es capaz de generar suficiente trabajo para alcanzar el *throughput* indicado o para encontrar el punto en el que se satura el sistema y es necesario emplear varios nodos clientes. Esto no sería un

inconveniente si estuviera automatizado y coordinado por el propio sistema. Sin embargo, el proceso para realizar esta tarea consiste en lanzar el cliente al mismo tiempo desde las diferentes máquinas manualmente y, posteriormente procesar los ficheros de resultados generados por cada nodo para conseguir los datos totales de la ejecución.

3.2.2. Metodología de empleo

Para hacer uso del sistema YCSB hay que seguir una serie de pasos. En primer lugar, es necesario escoger el conector de bases de datos que se va a utilizar (el listado completo se encuentre en su repositorio[2] en GitHub). Después, hay que definir la base de datos que se va a evaluar y generarla. Una vez terminado esto se puede descargar y compilar el código. Cuando se tiene todo el código compilado y los programas para generar el entorno de ejecución necesario para YCBS, es el momento de comenzar a definir los casos de prueba y en base a estos, definir los ficheros de configuración. Se deben definir los parámetros mencionados anteriormente y los parámetros de ejecución asociados a la instancia de la base datos que se desea probar. Estos parámetros son: IP de los nodos, nombre de la base de datos y nombre de la tabla. Estos parámetros pueden variar en función de cada conector.

Cuando ya está todo listo se puede comenzar a ejecutar el cliente. El proceso de ejecución tiene dos fases: *load* y *run*. Para ejecutarlo hay que utilizar los siguientes comandos:

```
ycsb load [binding] [options]
ycsb run [binding] [options]
```

3.3. Cassandra-stress

En este apartado se va a describir Cassandra-stress[4], una herramienta software proporcionada por el equipo de Datastax para medir el rendimiento de consultas sobre diferentes diseños de bases de datos en Cassandra. De Cassandra-stress se expondrán sus características, posibilidades y limitaciones, así como el proceso que se ha seguido para diseñar y ejecutar las pruebas.

3.3.1. Características

Cassandra stress es un software de benchmark desarrollado en lenguaje java y diseñado por el equipo de DataStax para medir el rendimiento de bases de datos Cassandra. Este nos permite generar una carga de trabajo similar a la que se espera que el sistema soporte en el entorno de producción real. Para ello, nos permite definir el *keyspace* y la tabla a probar, especificando para cada columna: el tipo de dato, el dominio de valores y el tamaño que puede tener el campo, que puede ser un valor fijo o un rango. Además de esto, el sistema permite definir diferentes *querys* que se ejecutan cada una en la proporción indicada como parámetro al iniciar el programa. Toda esta información se le proporciona al programa mediante un fichero en formato .yaml (ver Anexo B).

El programa se ejecuta con el comando *cassandra-stress* al que se le deben pasar los siguientes parámetros:

- **user**: permite establecer ciertos parámetros para realizar ejecuciones con diseños propios.
- **duración**: se puede establecer mediante un número de consultas o por tiempo medido en segundos, minutos u horas.

- **profile**: ruta del fichero de configuración en el sistema de archivos.
- **ops**: proporción de cada operación durante la ejecución.
- **log**: para guardar los datos generados por Cassandra-stress como resultado de la ejecución, en un fichero de log.
- **no-warmup**: si se quiere comenzar el test en frío, es decir, sin realizar una serie de iteraciones de calentamiento.
- **threads,throughput**: Para controlar la velocidad con la que se generan las operaciones contra la bases de datos, a través del número de hilos o de las operaciones por segundo.
- **node**: IPs de los nodos a los que se conecta separadas por coma.

Al igual que YCSC, este programa genera un informe que contiene datos de las latencias obtenidas durante las pruebas y que incluye: latencia media, percentiles 95, 99 y 99,9, throughput alcanzado y operaciones totales.

A continuación se señalan las ventajas y las limitaciones que tiene esta herramienta.

Ventajas

Cassandra stress ofrece una serie de ventajas frente a YCSB. Una de las principales ventajas de Cassandra stress, es que se trata de una herramienta altamente especializada y, por tanto, permite mucha más precisión a la hora de diseñar las pruebas y admite muchos más tipos de dato que YCSB. Otra de estas ventajas es que esta herramienta ofrece la posibilidad de realizar o no, un conjunto de operaciones de calentamiento contra la base de datos antes de tomar medidas de la ejecución. Esto tiene dos consecuencias positivas: permite simular un mayor número de escenarios y permite calentar la base de datos y el cliente. Esto último es bastante importante, ya que está desarrollado en Java y, por tanto, corre sobre una máquina virtual y el proceso de inicialización podría alterar los valores medidos.

Limitaciones

En cuanto a la limitaciones de Cassandra-stress, podemos destacar que no permite hacer pruebas sobre bases de datos que tengan tipos de datos definidos por el usuario, mapas o listas.

Y, por otro lado, una de sus ventajas es también una desventaja, ya que al ser una herramienta desarrollada y distribuida por DataStax no es de código libre y no se puede extender o ampliar como se puede hacer con YCSB.

3.3.2. Metodología de empleo

Para llevar a cabo la ejecución de las pruebas, se ha definido un proceso de trabajo similar al previamente expuesto para YCBS. En primer lugar se ha diseñado un conjunto de pruebas a realizar con la herramienta. Para generar este conjunto de pruebas se ha pensado primero en un caso de uso y varias alternativas de diseño de tablas que puedan favorecer los diferentes tipos de consultas que se realizarían en ese supuesto.

Una vez diseñadas la pruebas y las tablas sobre las que se van a realizar estas pruebas, se han generado los ficheros necesarios para la realizar la ejecución de las pruebas con esta herramienta, en el caso de Cassandra-stress los ficheros de profile con extensión *.yaml*.

Tras completar el paso anterior, se han generado unos *scripts* bash con los comandos de ejecución de Cassandra-stress, redirigiendo la salida de la ejecución a unos ficheros para almacenar los resultados, de manera que las pruebas sean fácilmente repetibles en caso de que fuera necesario.

Por último, se han analizado y procesado los ficheros de datos generados por Cassandra Stress como resultado de las diferentes pruebas. Este análisis se describe en el apartado siguiente.

4 Benchmark

En este capítulo se detalla el conjunto de pruebas diseñadas para ejecutarse con las herramientas YCSB y Cassandra-stress, descritas en el apartado anterior. A continuación, se explicará paso a paso el proceso seguido para ejecutar las pruebas en el laboratorio y obtener las métricas de rendimiento solicitadas. Se señalaran además los problemas encontrados y las soluciones que se han aplicado para solventarlos. Por último, se procederá a realizar un análisis y discusión de los resultados obtenidos en estas pruebas.

4.1. Descripción del conjunto de pruebas

En esta sección se describen de las pruebas que se van a realizar para verificar el comportamiento de Cassandra bajo diferentes configuraciones y cargas de trabajo. Se agruparán en dos apartados:

- Pruebas de rendimiento y escalabilidad: estas pruebas están dirigidas a analizar el impacto de añadir nodos al clúster, así como su caída y restauración y evaluar el efecto de la replicación y los niveles de consistencia en el tiempo de respuesta del sistema. Se ejecutarán con el sistema YCSB.

- Pruebas para determinar estrategias de diseño: estas pruebas pretenden dilucidar los costes de rendimiento relacionados con el diseño de las tablas que almacenan la información en Cassandra. Estas se realizarán con el software Cassadra-stress.

4.1.1. Pruebas de rendimiento y escalabilidad

Con ayuda de la herramienta YCSB se ha analizado el rendimiento de Cassandra bajo diferentes configuraciones pero siempre enfocadas en medir la escalabilidad del sistema. Las pruebas se han configurado en base a la combinación de los siguientes parámetros:

- **N°de nodos**: es el número de instancias de Cassandra que hay en un clúster.

- **Tamaño de las tuplas de datos**: es el tamaño de cada fila de datos de una tabla.

- **N°de filas**: es el número de filas o tuplas de valores que contiene una tabla.

- **Factor de replicación**: es el número de copias de los datos que se reparten entre los nodos.

- **Nivel de consistencia**: es el número de nodos que deben confirmar las transacciones. Por ejemplo, si se exige un nivel de consistencia 2 significa que el valor tiene que estar escrito en al menos 2 nodos para que se confirme la operación, en caso contrario, se cuenta como operación fallida.

Para cada uno de esos parámetros se han establecido una serie de valores, que se irán combinando para obtener distintos casos de prueba. Los valores escogidos son los siguientes:

- N°de nodos: 1 y 3
- Número de filas: 1.000.000, 33.000.000
- Tamaño de filas: 24Bytes, 100bytes
- Factor de replicación (Fr): 1 y 3
- Nivel de confirmación (Nc): 1, 2 y 3

A partir de combinar estos valores se especificaron las siguientes configuraciones:

- **conf 0**: 1.000.000 filas, 24 bytes, Fr=1, Nc=1
- **conf 1**: 1.000.000 filas, 100 bytes, Fr=1, Nc=1
- **conf 2**: 33.000.000 filas, 24 bytes, Fr=1, Nc=1
- **conf 3**: 1.000.000 filas, 24 bytes, Fr=3, Nc=2
- **conf 4**: 1.000.000 filas, 24 bytes, Fr=3, Nc=3

La configuración 0 tiene como objetivo obtener curvas de latencia vs *throughput* para ver como escala el sistema al añadir nodos. Las configuraciones 1 y 2 persiguen medir el impacto que tiene el crecimiento de los datos, tanto horizontal como verticalmente, es decir, como afecta a los tiempos de respuesta el número de columnas y su tamaño, y el número de filas respectivamente. Las configuraciones 3 y 4 pretenden medir el impacto que tiene en el rendimiento aumentar los niveles de confirmación y el número de réplicas.

Estás pruebas se centrarán en lecturas y actualizaciones, ya que, por un lado, el diseño de tablas que nos permite usar YCSB no está preparado para hacer consultas por rangos y los datos que otorga no son útiles y, por otro lado, las escrituras podrían desbalancear el tamaño de la base de datos provocando distintas situaciones en las pruebas y, por tanto, falsear los resultados. La parte de escritura queda cubierta por las actualizaciones, ya que, en Cassandra se sigue una estrategia de *upsert*, por tanto, las actualizaciones son escrituras con un valor de la clave primaria ya existente.

Estas cinco configuraciones se probarán por cada tamaño de clúster. En cada caso, se solicitará al cliente que ejecute operaciones con diferentes *throughputs*, que irán aumentando desde 10.000 op/s hasta 100.000 op/s. Cada uno de estos *throughputs* se mantendrá un minuto, con un periodo de *warm-up* de un minuto al comienzo de la prueba. Como resultado, se leerán y visualizarán la latencia media y percentiles 95 y 99, que ofrece la herramienta YCSB.

Con todo esto quedan cubiertos los casos de escalado en frío del sistema. Sin embargo, no se prueba la capacidad del sistema de escalar en caliente o de recuperarse ante fallos. Por ello, tras ejecutar estas pruebas se decidió añadir dos pruebas adicionales: detener el servicio de Cassandra en un nodo y añadir un nuevo nodo al clúster. En el primer caso, la idea es iniciar una carga de trabajo moderada y que se prolongue durante 7 minutos y detener el servicio de Cassandra en uno de los nodos para simular un fallo, posteriormente reiniciar el servicio y ver el efecto que tiene la conexión del nodo en el rendimiento. En el segundo caso, se arranca la prueba de la misma manera pero esta vez con una carga de trabajo alta e incorporando un nuevo nodo al clúster.

Para la prueba de rendimiento ante fallos, se dejarán 50 hilos ejecutando durante 7 minutos, en el minuto 2 se parará el servicio de Cassandra en un nodo del clúster y en el minuto 3 se volverá a iniciar el servicio.

Para la prueba de incorporación de nodos al sistema, se iniciará una carga de trabajo de 10 minutos con 50 hilos y sin especificar el *throughput*, para que se ejecuten a la mayor velocidad posible y se incorporará un nuevo nodo en el minuto 3 de ejecución. Esta prueba es más larga que la anterior debido a que puede variar el tiempo que tarda el nodo en incorporarse al clúster, sincronizarse y estar disponible.

4.1.2. Pruebas para determinar estatégias de diseño

En este caso se ha utilizado la herramienta Cassandra-stress. Se ha realizado el diseño de casos de prueba para analizar el rendimiento de tres consultas aplicando diferentes estrategias de diseño. Además, se han diseñado también algunos casos de prueba que se creen relevantes para poder adaptar Cassandra con éxito a un entorno de Industria 4.0. En primer lugar, se diseñó un esquema conceptual de una base de datos para un caso de IoT (*Internet of Things*) (ver Ilustración 6). En concreto, recoge la descripción de varias máquinas, con diferentes dispositivos o sensores, los cuales toman medidas de ciertos parámetros y las almacenan.

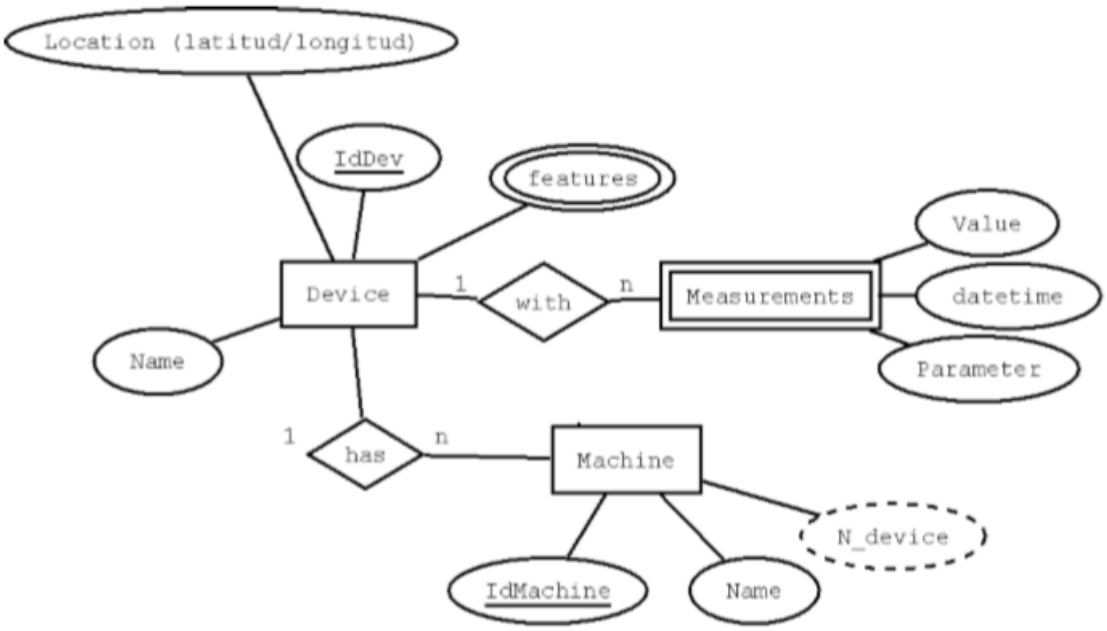

Ilustración 6: Diseño conceptual de base de datos para IOT

Las consultas propuestas son:

- **Q1**: Obtener una máquina y listar todas las máquinas.
- **Q2**: Listar los dispositivos de cada máquina ordenados por localización.
- **Q3**: Recoger todas las medidas tomadas por cada dispositivo por cada parámetro y hora de cada día.

La primera de las consultas se quiere aprovechar para ver el efecto de la partición en el rendimiento de los sistemas. En este caso, el propósito es generar un gran número de tuplas, en torno a 1 millón, y comparar el efecto que tiene establecer una clave primaria simple, lo que llevaría a generar un gran número de particiones, o determinar una clave primaria compuesta, añadiendo un campo bucket como *partition key*. De esta manera, se puede controlar el número de particiones y agrupar los datos en un determinado nodo.

Script 4.1: Estrategias de diseño: Q1 (Alternativa 1)

```
CREATE TABLE machines (
IDmachine UUID,
name text,
n_device int,
PRIMARY KEY (IDmachine));
```

Script 4.2: Estrategias de diseño: Q1 (Alternativa 2)

```
CREATE TABLE machines (
Bucket int,
IDmachine UUID,
name text,
n_device int,
PRIMARY KEY (Bucket, IDmachine));
```

La segunda consulta se plantea para evaluar el impacto que tiene el uso de mapas en el rendimiento de las consultas y actualizaciones. Se pretende valorar esto para almacenar los parámetros que un sensor es capaz de medir. Para ello se proponen dos alternativas: añadir un mapa a la tabla de dispositivos y guardar un parámetro por cada fila redundando las columnas *Name* y *Location*.

Script 4.3: Estrategias de diseño: Q2 (Alternativa 1)

```
CREATE TABLE devices (
MachineName text,
IdDevice uuid,
Location text,
name text,
features map<text><text>,      (-- feature, valor)
PRIMARY KEY (machinesName, IDdevice));
```

Script 4.4: Estrategias de diseño: Q2 (Alternativa 2)

```
CREATE TABLE devices (
MachineName text,
IdDevice uuid,
Location text,
name text,
feature text,
PRIMARY KEY (machinesName, IDdevice, feature));
```

La tercera consulta se quiere aprovechar para analizar los casos en los que el volumen de datos crece con gran rapidez, característica típica de los sistemas *big data*. Una de las limitaciones[1] de Cassandra es el número de valores por partición que soporta, que se encuentra en 2 billones americanos de valores (filas x columnas). En un sistema de

este tipo es razonable que se llegue a esos niveles ya que los sensores toman valores cada segundo o cada pocos milisegundos en el caso de algunos sistemas críticos. Por tanto, se pretende hacer unas estimaciones del tamaño de las tablas y del número de valores que podrían generarse en diferentes supuestos.

En este caso se han planteado 4 alternativas: partición por máquina, partición por dispositivo, partición por dispositivo y parámetro y, partición por dispositivo parámetro y hora.

Script 4.5: Estrategias de diseño: Q3 (Alternativa 1)

```
CREATE TABLE measurement_by_parameter_hour (
IDMachine UUID,
DeviceName text,
parameter text,
value float,
date timestamp --(date solo),
hour int,
PRIMARY KEY ((IDMachine), DeviceName, parameter,hour ,datetime)
   WITH CLUSTERING ORDER BY (datetime DESC))
```

Script 4.6: Estrategias de diseño: Q3 (Alternativa 2)

```
CREATE TABLE measurement_by_parameter_hour (
IDMachine UUID,
DeviceName text,
parameter text,
value float,
datetime timestamp (date solo),
hour int,
PRIMARY KEY ((IDMachine, DeviceName), parameter,hour, datetime)
   WITH CLUSTERING ORDER BY (datetime DESC))
```

Script 4.7: Estrategias de diseño: Q3 (Alternativa 3)

```
CREATE TABLE measurement_by_parameter_hour (
IDMachine UUID,
DeviceName text,
parameter text,
value float,
datetime timestamp (date solo),
hour int,
PRIMARY KEY ((IDMachine DeviceName, parameter), hour ,datetime)
   WITH CLUSTERING ORDER BY (datetime DESC))
```

Script 4.8: Estrategias de diseño: Q3 (Alternativa 4)

```
CREATE TABLE measurement_by_parameter_hour (
IDMachine UUID,
DeviceName text,
parameter text,
value float,
datetime timestamp (date solo),
hour int,
PRIMARY KEY ((IDMachine, DeviceName, parameter,hour), datetime)
    WITH CLUSTERING ORDER BY (datetime DESC))
```

4.2. Proceso de ejecución de las pruebas

Para la realización de las pruebas, en primer lugar se estudió el funcionamiento de las herramientas, comenzando por YCSB. En concreto, se analizaron los parámetros de configuración disponibles, la compatibilidad con Cassandra y entorno de ejecución necesario.

A continuación se realizaron pruebas de rendimiento básicas, empleando el esquema de prueba que se incluye en su repositorio de GitHub.

Con esto se llegó al siguiente listado de parámetros interesantes para configurar nuestras pruebas:

- **readproportion, writeproportion, scanproportion y updateproportion**: proporción de cada tipo de operación.
- **readconsistency y writeconsistency**: nivel de consistencia de lecturas y escrituras, respectivamente.
- **keyspace**: keyspace contra el que se ejecuta el benchmark.
- **recordcount**: número de filas que hay en la tabla.
- **operationcount**: número de operaciones que se van a ejecutar.
- **fieldcount**: número de columnas de la tabla.
- **fieldlength**: tamaño de las columnas.
- **maxexecutiontime**: tiempo máximo de ejecución del cliente.
- **table**: tabla sobre la que se ejecutan las operaciones.
- **host**: nodos de contacto.
- **threads**: número de hilos que van a correr en la aplicación cliente.
- **target**: *throughput* objetivo de la ejecución del cliente.

Tras encontrar una configuración de parámetros adecuada, se generaron varios ficheros de configuración, *scripts* .cql para generar los esquemas y *scripts* para automatizar el lanzamiento de algunas pruebas. Al lanzar estas pruebas, el software YCSB comenzó a fallar debido a que solo admite columnas de tipo texto o blob. Por ello, se modificaron los ficheros .cql para adaptar los esquemas a estas características y los ficheros de configuración para ajustar los tamaños de las columnas.

Tras este proceso de familiarización, se desplegó Cassandra en el laboratorio. Se crearon un clúster de 1 nodo y un clúster de 3 nodos. Las máquinas empleadas tienen las siguientes características:

- **Procesador**: Intel Core i5-7500 @ 3,40GHz x 4
- **Memoria**: 8GB DDR4 2400 MHz
- **Disco**: 33,4 GB HDD 7200rpm

En primer lugar, se usó como cliente uno de los nodos del clúster, pero los resultados no concordaban con las pruebas previas, realizadas durante el periodo de familiarización con el sistema. Por ello, se hizo una prueba traspasando el cliente a un pc ajeno al clúster. Con esto, se consiguió una mejora en el rendimiento, en especial en el clúster de un nodo. A partir de ese momento se empezó a usar como cliente un pc con las siguientes características:

- **Procesador**: Intel Core i5-4590 @ 3,30GHz x 4
- **Memoria**: 8GB DDR3 1600 MHz
- **Disco**: 33,4 GB HDD 7200rpm

Durante la ejecución de las pruebas, se detectó que los resultados eran totalmente diferentes a los esperados, ya que aumentar el tamaño del clúster suponía una pérdida de rendimiento, y en alguna ocasión se detuvieron las pruebas por la caída de algún nodo. Esto se produjo como consecuencia de la falta de espacio de almacenamiento. Esto ha sido un factor limitante durante todo el proceso de ejecución de las pruebas, ya que era necesario hacer una limpieza del disco después de cada caso de prueba para que los resultados no se viesen afectados.

Después de detectar el problema del almacenamiento, se liberó espacio en todos los ordenadores y se reconfiguró el clúster para repetir todas las pruebas.

Bajo esta situación, el aumento del número de nodos, como era de esperar, mejoró el rendimiento del sistema hasta un punto en que el cliente no era capaz de generar suficiente carga de trabajo como para saturarlo. Para conseguir esta saturación, se ejecutó la aplicación cliente en paralelo desde dos nodos externos al clúster. Para ello se utilizó *parallel-ssh*, manteniendo un *throughput* constante en uno de los nodos y aumentando progresivamente el *throughput* en el otro.

Tras terminar las pruebas con YCSB, se investigó el funcionamiento de la herramienta Cassandra-stress. En primer lugar, se comprobó que se adaptaba a nuestras necesidades y, en segundo lugar, se estudió el la configuración de los ficheros de perfil (profile) para las ejecuciones. Estos ficheros constan de 3 secciones:

- **CQL de creación de los esquemas**: en esta sección se indica qué keyspace y tabla se van a probar y, opcionalmente, se añade su definición.
- **Descripción de las columnas**: en esta sección se describen el tipo de dato, el dominio y el tamaño de las columnas. Esta información es utilizada por Cassandra-stress para la generación de datos.
- **Configuración de los *batches* de operaciones**: en esta parte se configura cuántas particiones y cuántas filas de la partición se van a ver afectadas por cada operación que se realiza contra la base de datos.

- **Consultas en lenguaje CQL**: en este apartado se indican las consultas que se quieren medir.

Tras conocer el formato de los ficheros de perfil y el funcionamiento de la herramienta, se ejecutaron las pruebas en el laboratorio con los mismos equipos que se utilizaron para las pruebas ejecutadas con YCSB.

A continuación, se exponen y analizan los resultados obtenidos con las pruebas realizadas.

4.3. Análisis de resultados

Una vez realizadas todas las pruebas, se muestran las métricas recogidas gráficamente y se explica el comportamiento observado.

Del mismo modo que en el apartado *4.1 Descripción de la batería de pruebas*, expondremos los resultados de las pruebas realizadas con Yahoo y Cassandra-stress en dos apartados diferentes.

4.3.1. Pruebas de rendimiento y escalabilidad

Las pruebas diseñadas nos han permitido evaluar el comportamiento Cassandra en una serie de situaciones que se exponen a continuación.

Latencia vs *Throughput*

En primer lugar, haciendo uso de la configuración 0 (caso base), se mide la latencia vs *throughput* en un clúster de un solo nodo y con una base de datos pequeña con objeto de comprobar en que punto comienza a saturarse el servidor. Asimismo, se analiza la diferencia de rendimiento que hay entre las escrituras, que según la documentación de Cassandra son "baratas" y las lecturas, que son "caras".

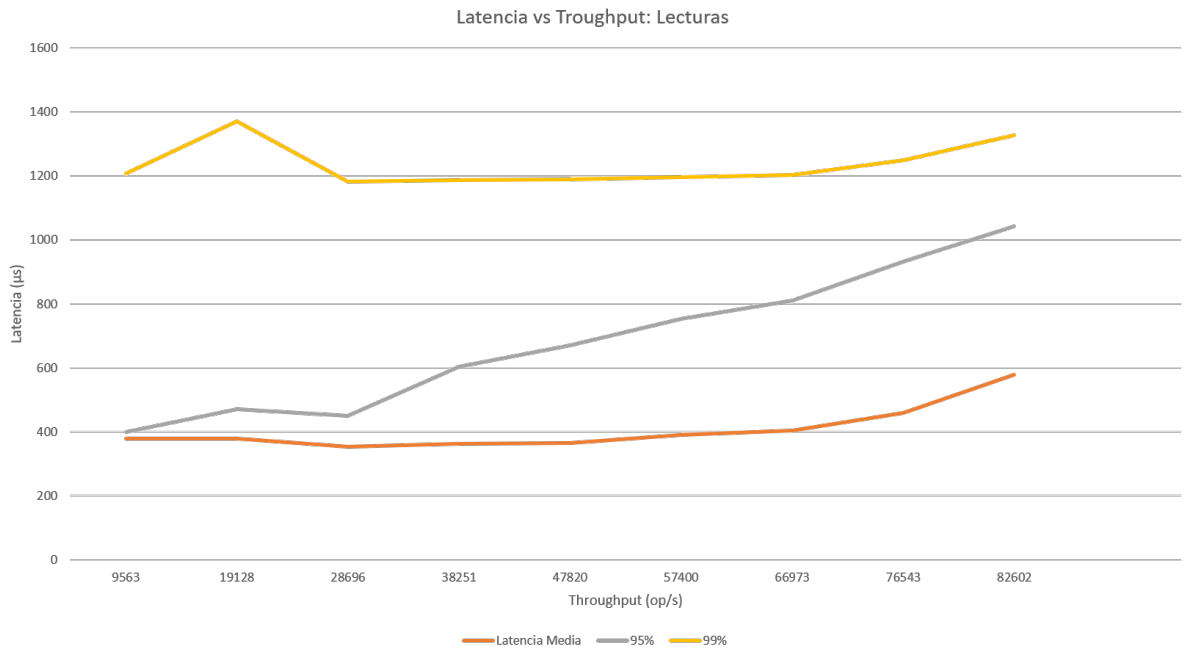

Ilustración 7: Latencia vs Throughput: Lecturas

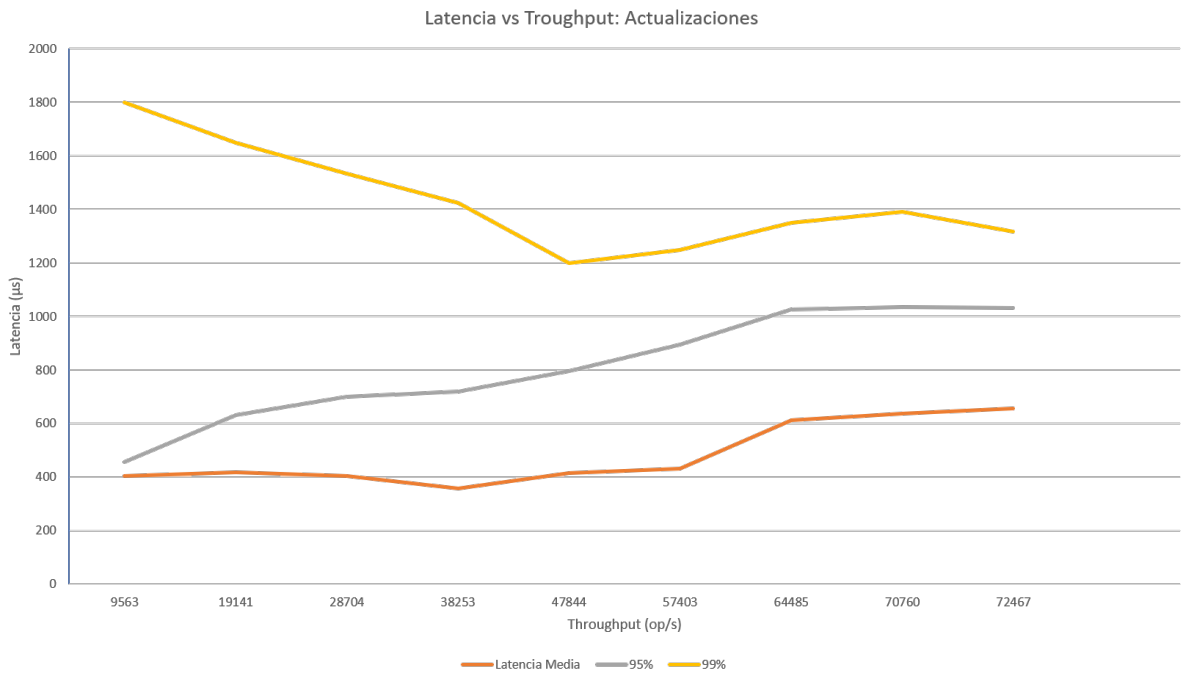

Ilustración 8: Latencia vs Throughput: Actualizaciones

En las ilustraciones 7 y 8, se puede observar como a medida que aumenta el throughput la latencia también crece. En un primer lugar aumenta lentamente y de manera más o menos contantante. Sin embargo, a partir de las 70.000 operaciones por segundo, el servidor empieza a tener problemas para atender las peticiones y la pendiente de la curva comienza a crecer.

En cuanto al comportamiento de Cassandra más favorable a las escrituras frente a lecturas, en esta configuración no se aprecia este hecho. Es más, incluso en ocasiones presenta latencias más altas en las escrituras que en las lecturas.

Tamaño de los datos

Una vez analizado el caso base, se ha hecho una comparativa con tablas de un tamaño mayor. En primer lugar, se repitió la prueba con la configuración 2, donde el tamaño de cada fila es 4 veces mayor. Y posteriormente, se repitió la prueba con la configuración 3, donde se trató de mantener el tamaño de la base de datos de la configuración 2 pero con el tamaño de filas de la configuración 1. Para su análisis se hace uso del percentil 95 de latencia, ya que el percentil 99 se ve muy afectado por las iteraciones de *warmup* y, la latencia media queda muy plana y suaviza demasiado los efectos de los diferentes cambios. Por tanto, para ver la tendencia el dato más útil es el percentil 95.

En las ilustraciones 9 y 10, se puede ver el efecto del crecimiento de la tabla. En ambos casos, tanto cuando crecen las filas como las columnas, crecimiento en tamaño de las filas y crecimiento en el número de filas, se nota una reducción del rendimiento, pero es más notable en caso de un número alto de filas.

En el caso del la tabla con un elevado número de filas, se puede observar como al pasar de 20.000 operaciones de lectura por segundo o 35.000 operaciones de escritura por segundo, la latencia crece con rapidez. Una vez se han alcanzado las 45.000 operaciones de lectura por segundo o las 55.000 operaciones de escritura por segundo, el servidor se satura y no se puede alcanzar un *throughput* mayor.

En el caso del la tabla con filas de mayor tamaño, se puede observar como el crecimiento

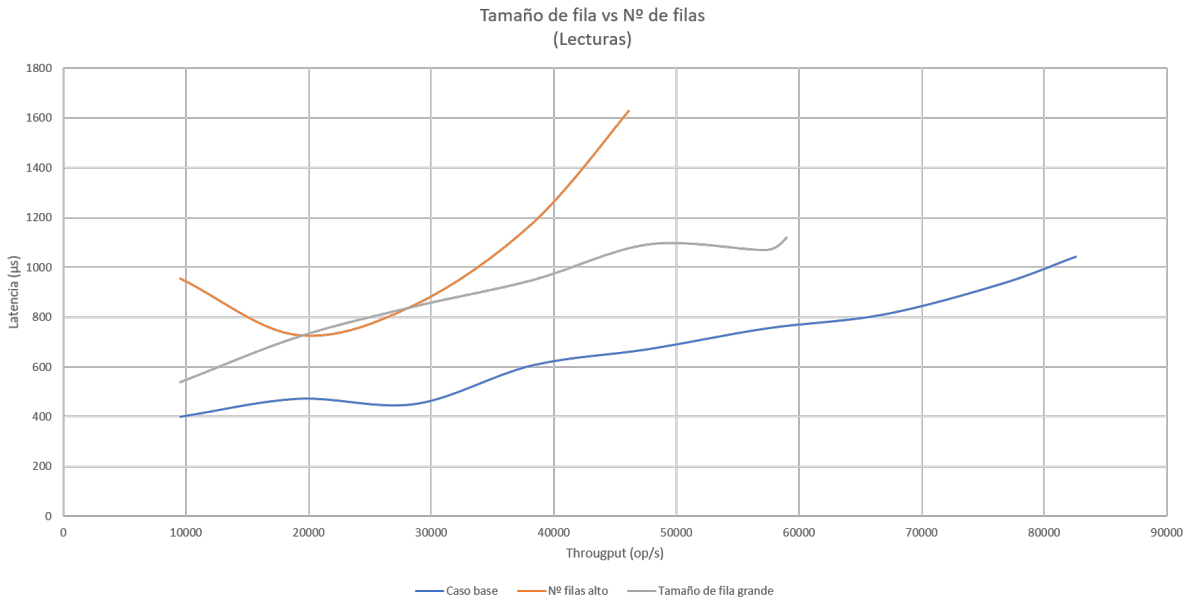

Ilustración 9: Tamaño vs Número de filas (Lecturas)

Ilustración 10: Tamaño vs Número de filas (Actualizaciones)

es algo más lineal y se alcanzan valores de *throughput* mayores, que en la tabla con un mayor número de filas. En este caso, el servidor comienza a saturarse al alcanzar cerca de 60.000 operaciones lectura por segundo y unas 68.000 operaciones de escritura por segundo.

En esta ocasión si se aprecia la diferencia entre lecturas y escrituras. En ambos casos, el *throughput* alcanzado ha sido mayor con las operaciones de escritura que con las de lectura. Además, en el caso de la tabla con un alto número de filas se han encontrado latencias mucho más bajas, entre 400 y 900 microsegundos en el caso de escrituras y entre 700 y 1.700 microsegundos en caso de las lecturas. Esto indica que las diferencias entre escrituras y lecturas se hacen más notables a medida que la base de datos crece.

Escalado en frío

Tras observar como afecta el tamaño de los datos almacenados y el volumen de los mismos, en una base de datos de un solo nodo, se repite el proceso con un clúster de 3 nodos. Según la documentación de Cassandra y como se ha visto en el capítulo *2 Gestor de datos distribuido y escalable Apache Cassandra Cassandra*, una de las características de este gestor es que escala de manera lineal. Por tanto, no debería verse afectado de la misma forma que el clúster de un nodo, alcanzando valores más altos de throughput sin llegar a saturarse.

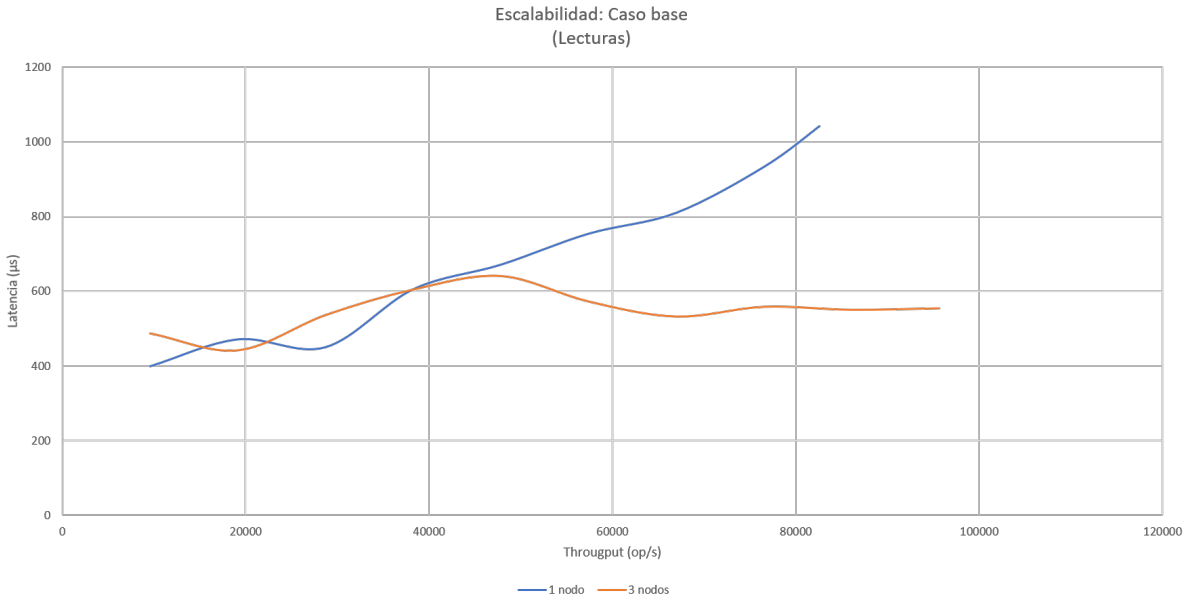

Ilustración 11: Escalabilidad: Caso Base (Lecturas)

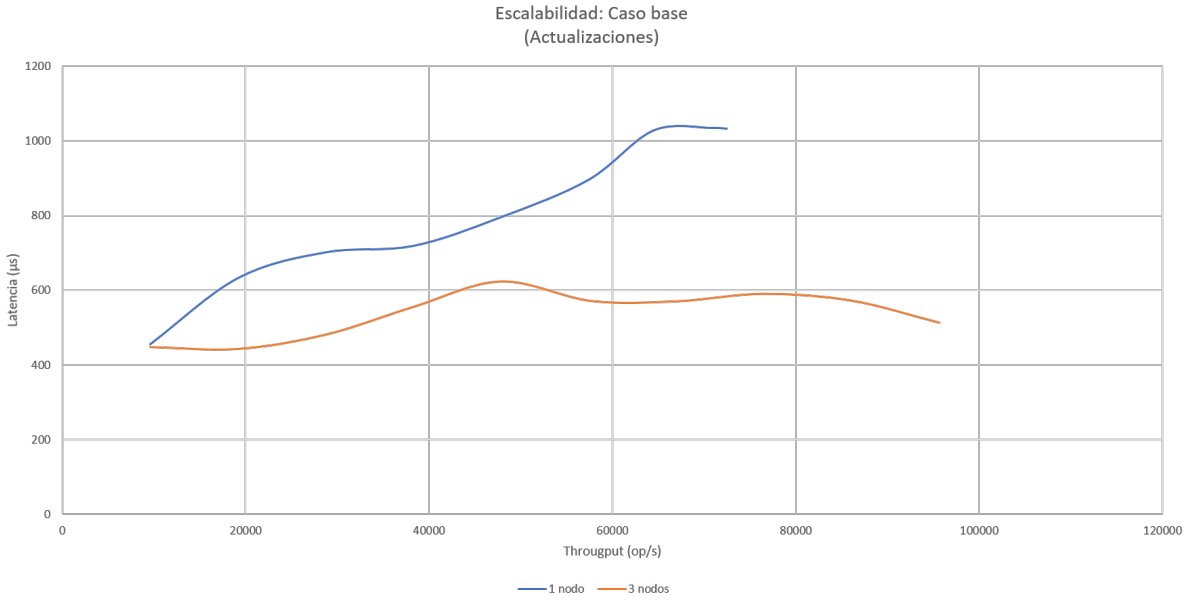

Ilustración 12: Escalabilidad: Caso Base (Actualizaciones)

Al repetir las pruebas con un clúster de 3 nodos, se observa que el rendimiento mejora notablemente en todos los casos, como se puede apreciar al comparar las ilustraciones 11 y 12, llegando antes al limite del cliente que del servidor. Especialmente mejora el caso en

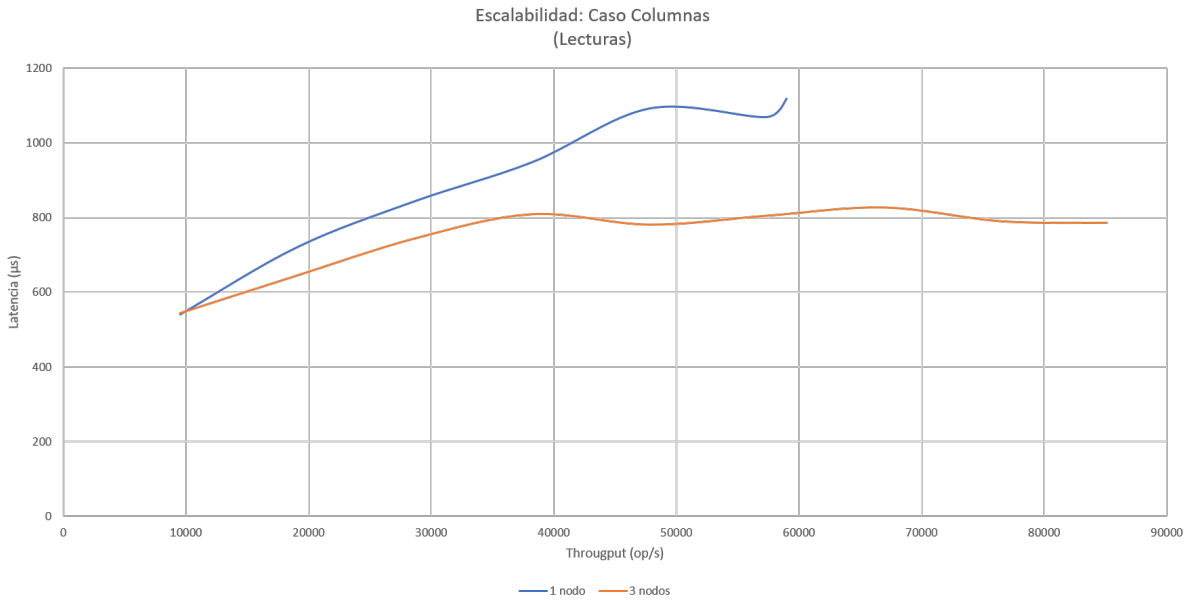

Ilustración 13: Escalabilidad: Caso Columnas (Lecturas)

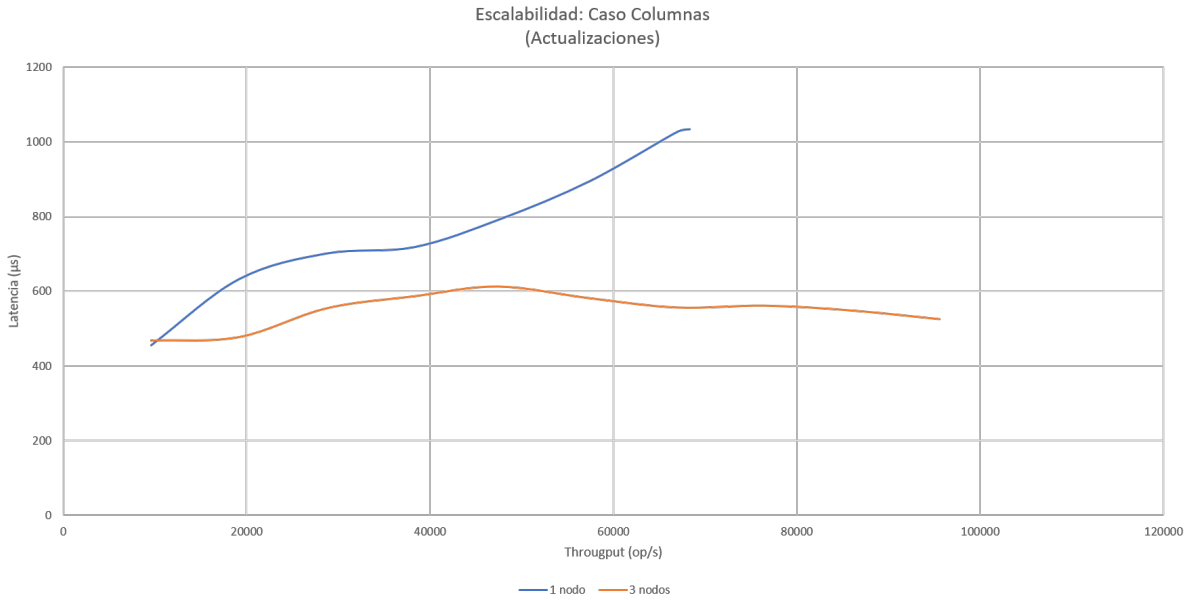

Ilustración 14: Escalabilidad: Caso Columnas (Actualizaciones)

el que el número de filas es mayor (ver ilustraciones 15 y 16). En ese caso, al repartirse las filas y el trabajo entre los 3 nodos, se reduce mucho el impacto que tiene el crecimiento de la tabla, pasando de saturarse con un *throughput* cercano a las 50.000 operaciones por segundo, a poder superar las 100.000 operaciones por segundo sin experimentar aumentos en la latencia de las operaciones

También se puede observar como al ampliar el tamaño de clúster sigue existiendo una diferencia entre lecturas y escrituras, pero se reduce considerablemente, reforzando así la idea de que la mayor diferencia se produce cuando la base de datos crece y se acerca al limite.

Como se aprecia en los casos en los que se ejecutan las pruebas con 3 nodos, el servidor responde de manera estable y sin saturarse, llegando antes al limite del cliente que del servidor. Por tanto, para llevar al sistema al límite, se repitió la prueba en la que el

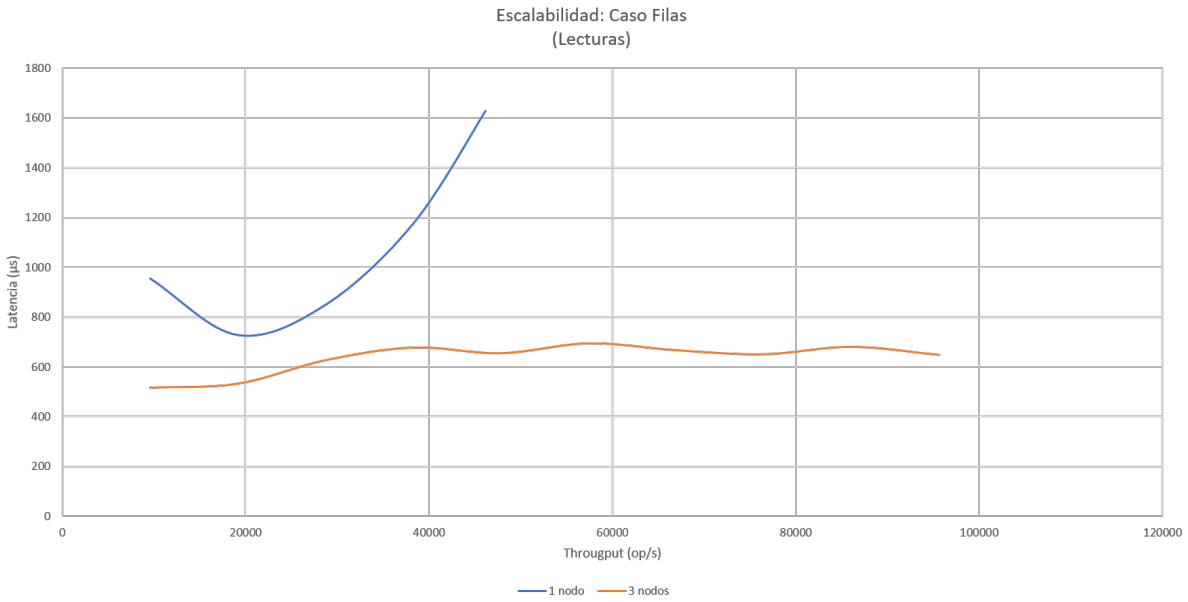

Ilustración 15: Escalabilidad: Caso Filas (Lecturas)

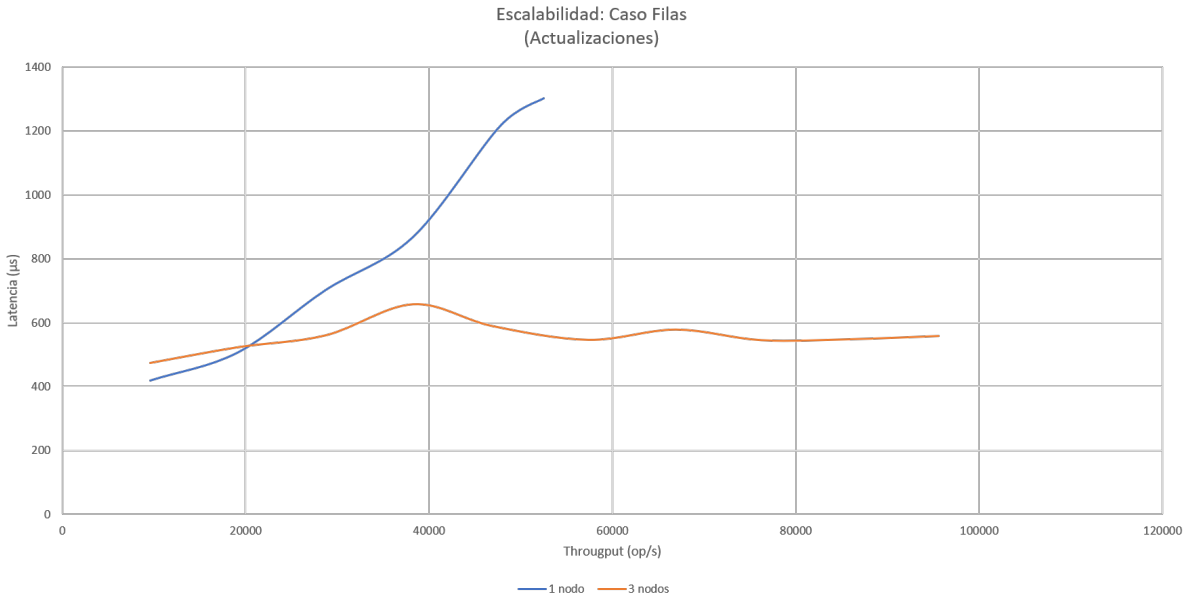

Ilustración 16: Escalabilidad: Caso Filas (Actualizaciones)

rendimiento fue peor, utilizando dos clientes en paralelo para intentar alcanzar el límite del servidor.

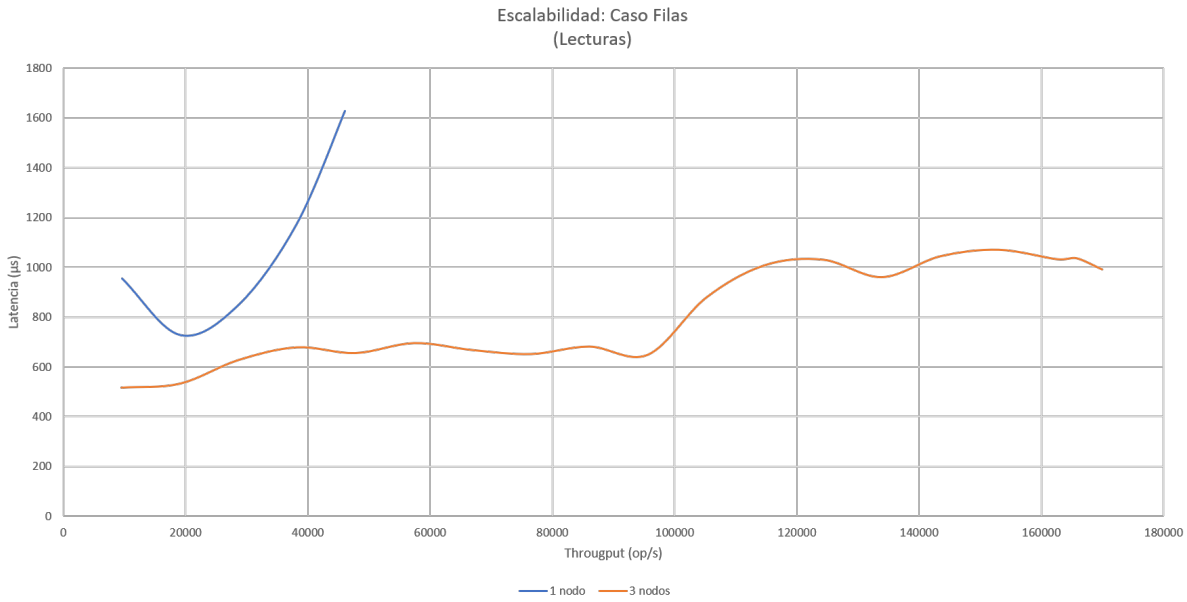

Ilustración 17: Escalabilidad: Caso Filas (Lecturas) 2 clientes en paralelo

En la ilustración 17, se puede observar como el rendimiento permanece constante hasta *throughputs* muy altos, llegando finalmente al límite con un *throughput* cercano a 170.000 operaciones por segundo, más del triple que en el caso del clúster de un nodo, donde las latencias comienzan a ser más irregulares.

Nivel de replicación

Otra de las características de Cassandra, es que se puede escoger un nivel de replicación de los datos, que afecta a toda la información alojada en el mismo datacenter, y un nivel de consistencia mínimo en las consultas, que se establece en cada consulta. La replicación favorece la disponibilidad del sistema frente a la caída de nodos, la consistencia garantiza que la información que se consulta o inserta es correcta y que prevalecerá en el sistema. Sin embargo, esto tiene un coste y es lo que se ha intentado medir en esta sección. A continuación, se muestran los resultados de ejecutar varias cargas de trabajo con diferentes niveles de confirmación sobre una base de datos con un factor de replicación 3 (ilustraciones 18 y 19):

Como se puede observar, tanto en lecturas como en escrituras el factor de replicación tiene un gran impacto tanto en la latencia de las operaciones como en el *throughput* que es capaz de soportar el gestor. En la ilustración 18, se puede ver que con los niveles de confirmación de lectura 2 y 3, la latencia de las operaciones comienza siendo un 54 % y 130 % más alta que la del nivel de confirmación 1, respectivamente. Sin embargo, la pendiente en ambas curvas es diferente, creciendo con mayor rapidez en el caso del nivel de confirmación 3. Finalmente, el servidor se satura con un *throughput* cercano a 48.000 operaciones por segundo con el nivel de confirmación 2 y 45.000 operaciones por segundo con el nivel se confirmación 3.

En la ilustración 19, se observa como, al igual que con las lecturas, el aumento del nivel de confirmación conlleva una gran penalización con una latencia superior a 1.400 milisegundo para un nivel de throughput de 10.000, casi el triple que con nivel de confirmación uno. Sin embargo, la diferencia en el punto de saturación con el caso en el que el nivel de confirmación es 1 es menor que en el caso de las lecturas. Es posible que el factor de replicación tenga algún efecto negativo en el rendimiento de las escrituras, ya que esta

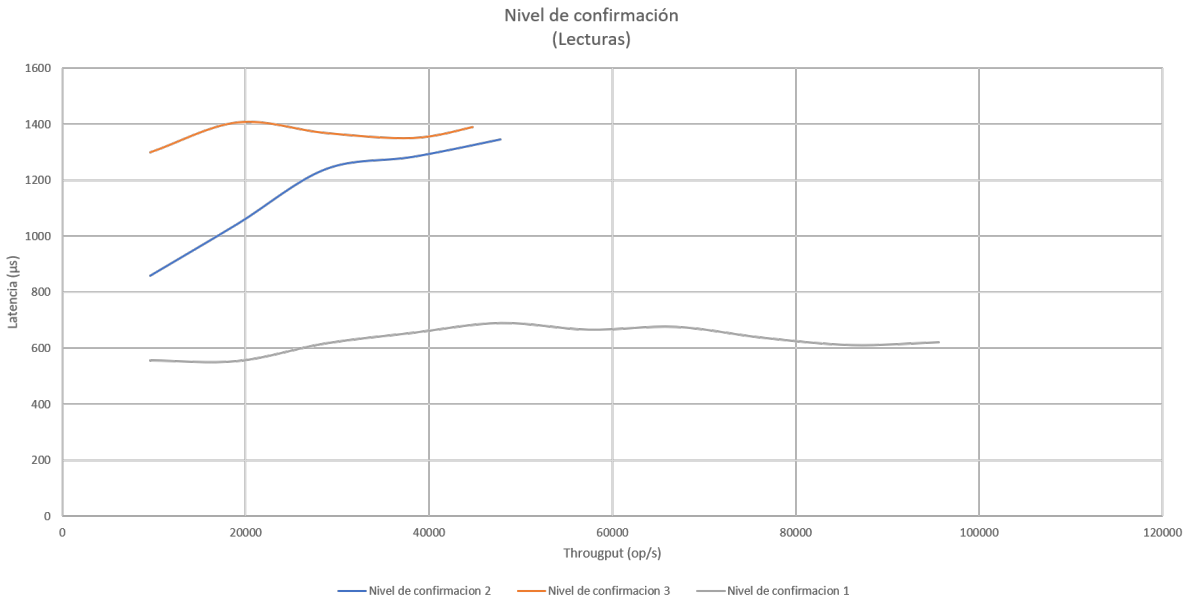

Ilustración 18: Nivel de confirmación (Lecturas)

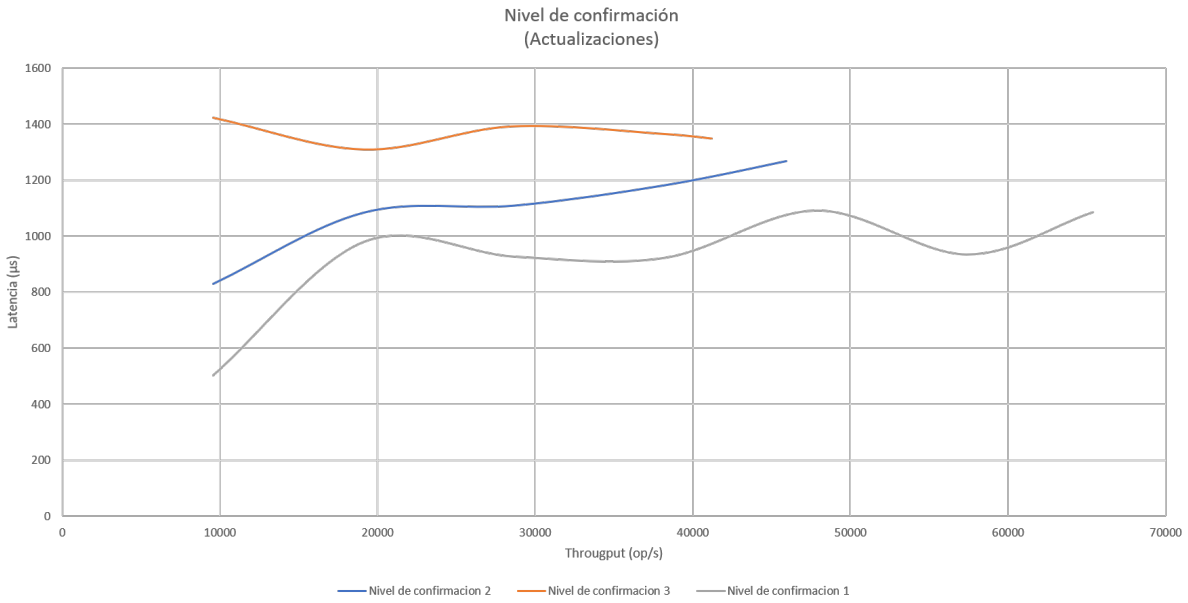

Ilustración 19: Nivel de confirmación (Actualizaciones)

prueba se ha repetido varias ocasiones con resultados similares, y que no se corresponden con los resultados obtenidos en las escrituras realizadas con factor de replicación 1.

Escalado en caliente

Ahora que se ha visto como se comporta Cassandra al aumentar el número de nodos en frío, es decir, parando el sistema y reconfigurando las bases de datos. Se va a analizar el efecto que tiene añadir un nodo mientras el sistema está sometido a una carga de trabajo constante. En este caso, la prueba se ha realizado solamente para lecturas, ya que es lógico que si se escriben datos, parte de estos datos irán destinados a este nodo por el efecto del algoritmo de particionado y afectará al rendimiento, ya que al admitir nuevos datos se encargará de parte de las operaciones. Sin embargo, si la carga es de lecturas y se añade un nodo vacío que nunca ha pertenecido al clúster el comportamiento no está claro, ya

que al no tener datos no podría, a priori, atender a las consultas.

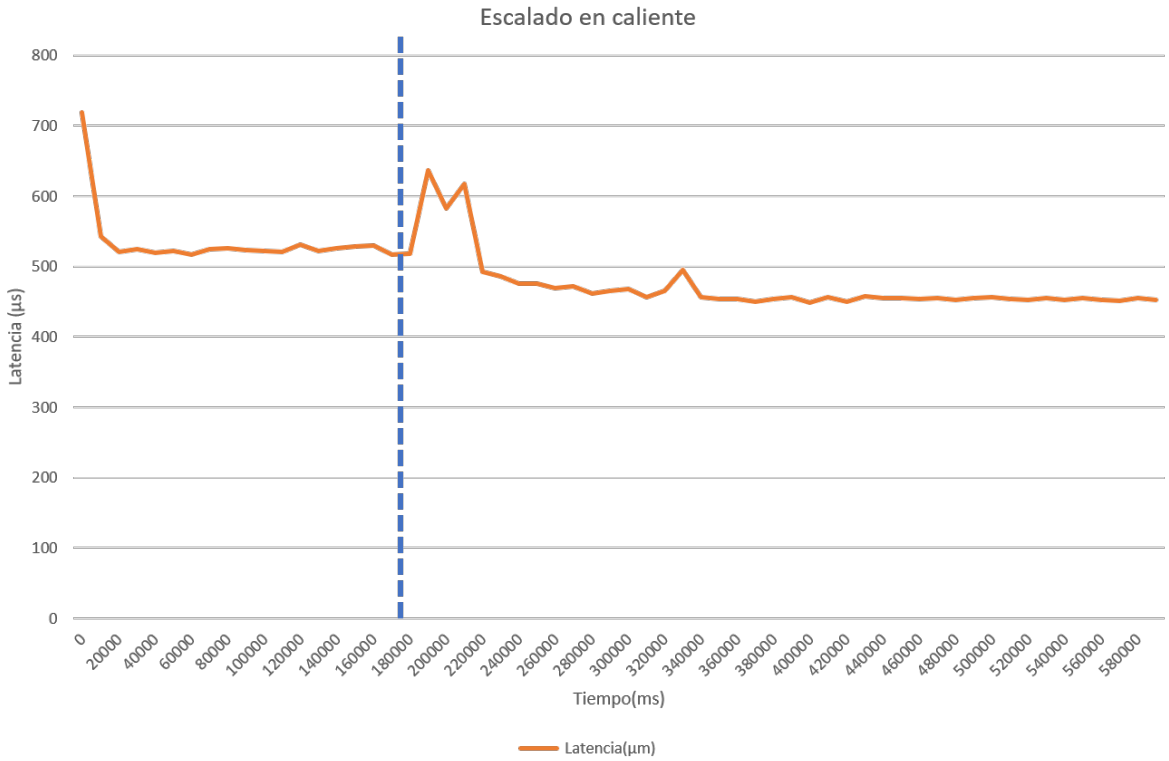

Ilustración 20: Escalado en caliente

```
gestor@CZC731837D:~$ nodetool status
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.31.16.47  128,91 MiB  256          49,5%             c6bc43c3-8907-4700-92ea-2bf9b77d4d55  rack1
UN  172.31.16.46  390,45 MiB  256          49,3%             7954c500-6d8f-40ff-a8c4-f1b4cb668ffd  rack1
UN  172.31.16.27  413,15 MiB  256          52,3%             8a6cc91f-1c6c-438e-81e6-13622e4d27c6  rack1
UN  172.31.16.68  388,65 MiB  256          48,8%             d0b72749-1bda-41c6-b5de-61f31fde1eab  rack1

gestor@CZC731837D:~$
```

Ilustración 21: Nodetool status: tras añadir un nodo con el sistema arrancado

Como se puede ver en la Ilustración 20, cuando se estabilizan los valores en la parte inicial de la prueba la latencia media se sitúa cerca de los 520 microsegundos.

En el minuto 3 de ejecución, se inicia la instancia de Cassandra en el nuevo nodo que provoca un aumento momentáneo de la latencia. Esto es debido a la incorporación y los mensajes del protocolo de *Gossip* empleado por Cassandra para ubicar a los nodos dentro de clúster. Finalmente, se puede observar que, pasados aproximadamente 20 segundos, la latencia media se estabiliza en torno a 450 microsegundos, que supone una reducción de un 13,5 % con respecto a los valores iniciales.

En la ilustración 21 se puede ver el estado del clúster al final de la prueba. Se puede comprobar como el nodo nuevo, con IP 172.31.16.47, tiene una carga total de 128.91 MB, a pesar de ser una carga de trabajo centrada únicamente en lecturas.

Tolerancia al fallo y recuperación

Por último, se va a probar la tolerancia ante fallos de Cassandra y su recuperación ante los mismos. Teniendo un factor de replicación 2, debería ser posible desconectar un

nodo de Cassandra y que pueda seguir atendiendo peticiones aunque a un ritmo más bajo. Y tras reconectarlo, debería volver a aumentar progresivamente el *throughput*.

Ilustración 22: Tolerancia al fallo y recuperación

En la ilustración 22 se puede observar como al comienzo de la ejecución la latencia se sitúa entre 1 y 1,2 milisegundos.

Tras detener una de las instancias de Cassandra en el minuto 2, la latencia media de las peticiones comienza a crecer, pero el sistema continua atendiendo correctamente a todas las peticiones.

En el minuto 3 de ejecución, cuando la latencia media se sitúa por encima de los 1,3 milisegundos y con una pendiente positiva que indica que seguirá creciendo, se vuelve a iniciar la instancia de Cassandra que se había detenido. Esto provoca un pico en la latencia media, debido al proceso de incorporación del nodo y finalmente un descenso rápido para situarse nuevamente cerca de 1 milisegundo.

El clúster en este caso, tarda aproximadamente 1 minuto en recuperarse totalmente, es decir, desde que se inicia de nuevo el nodo hasta que la latencia comienza a estabilizarse.

4.3.2. Pruebas para evaluar de estrategias de diseño

Consulta 1: Selección de PK y efecto de la partición

Con el diseño de la consulta 1 se pretende comprobar el efecto de la partición de los datos. En este caso se ha probado el efecto de tener cada fila en una partición y tener tantas particiones como nodos añadiendo un campo bucket que sea la partition key (ver Script 4.2). Con estas dos configuraciones de PK, se han realizado las siguientes consultas:

```
-- a) listar todos las máquinas
   SELECT * FROM iot.machines ALLOW FILTERING;
-- b) obtener una máquina en concreto.
   SELECT * FROM iot.machines WHERE idmachine = ?;
   SELECT * FROM iot.machines WHERE bucket = ? and idmachine = ?;
```

Así mismo se ha medido el efecto que tiene pedir los datos al nodo en el que están alojados frente a pedir los datos a otro nodo.

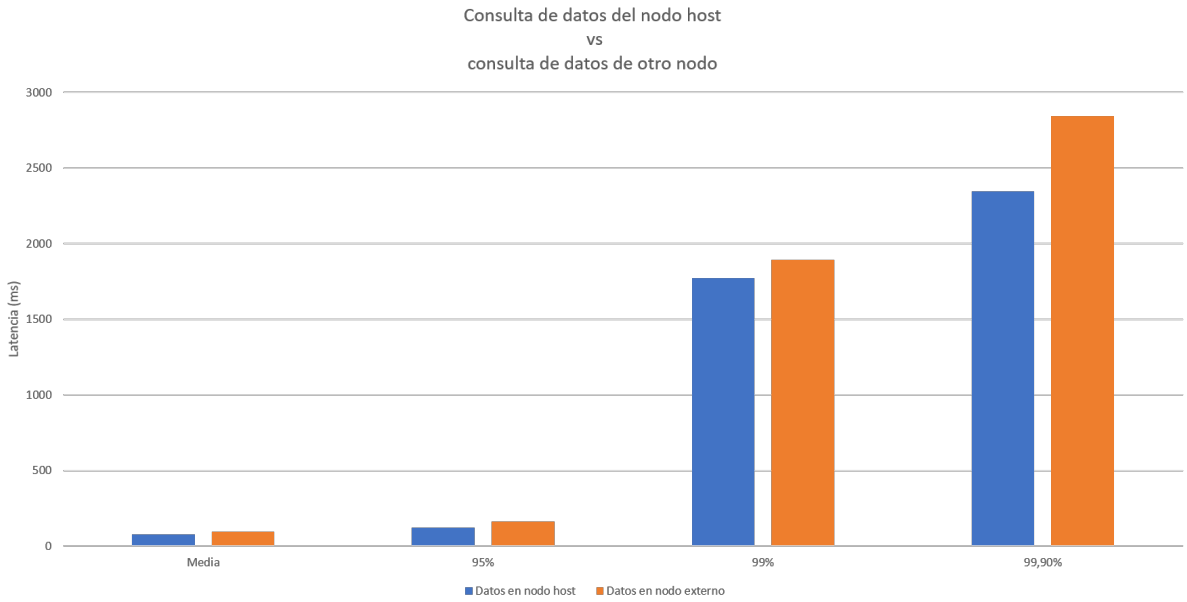

Ilustración 23: Consulta 1: Consultar datos al nodo que los almacena vs consultar dato a otro nodo

En la ilustración 23 se puede observar que realizando las consultas en al nodo en el que se encuentran los datos se reduce la latencia de las peticiones, ya que se ahorra parte del *overhead* introducido por comunicaciones de red. Además, la diferencia es cada vez mayor a medida que se aumenta el nivel de exigencia. Por tanto, parece beneficioso tratar de controlar donde se alojan los datos para reducir los tiempos de respuesta en sistema con restricciones temporales.

En el caso de querer listar todas las máquinas, solo ha respondido en el caso con menor número de particiones. En el caso en el que existe una partición por cada fila, el servidor se queda sin memoria y no es capaz de contestar a la consulta.

Consulta 2: Mapas, listas y tipos definidos por el usuario

Con la consulta 2 se pretendía comprobar el comportamiento de Cassandra cuando se almacenan datos en mapas, listas o tipos definidos por el usuario. Sin embargo, no ha sido posible medirlo porque ninguna de las herramientas de *benchmark* disponibles soporta esta posibilidad. A pesar de esto, se ha llegado a la conclusión de que podría ser una solución beneficiosa, ya que con el uso de estas estructuras se podría almacenar la misma cantidad de información con menor cantidad de filas, y como se ha mostrado en las ilustraciones 15 y 16, Cassandra se comporta mejor, tanto en lecturas como escrituras, cuanto menor sea el n°de filas a almacenar. Esto es, es preferible crecer en columnas que en filas.

Consulta 3: Consultas por rangos y límites del diseño

Con esta consulta se pretende medir el rendimiento de las consultas por rangos de clave para comprobar si se debe optar por este tipo de consultas o, por el contrario es mejor emplear tipos definidos por el usuario.

En la ilustración 24 se puede observar, cómo las consultas por rangos son bastante eficientes, ya que consultar 50 filas o consultar 100 filas tarda 2,25 y 4,37 veces más, en promedio, que consultar una sola fila, respectivamente. Por tanto, el coste de estas consultas es relativamente pequeño. Sin embargo, mantener los datos en filas de pequeño

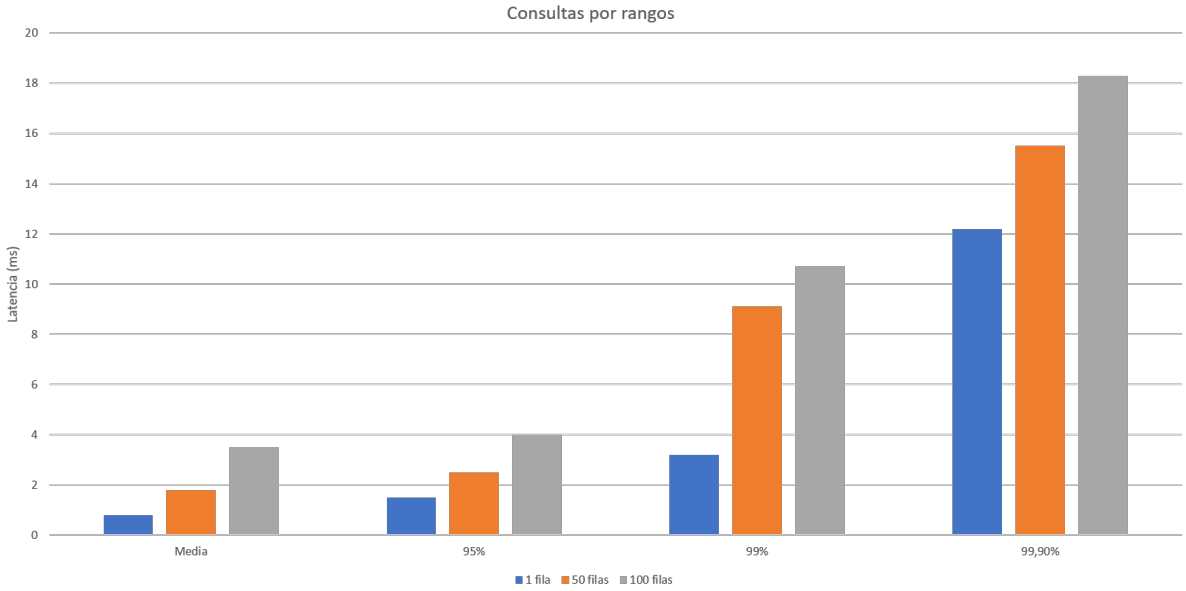

Ilustración 24: Query 3: Consultas por rangos

tamaño puede suponer un aumento excesivo de filas, por lo que si se sabe que la tabla no va a crecer con gran rapidez esta es una buena solución. De lo contrario, es mejor agrupar los datos en tipos definidos por el usuario, que no penalicen tanto el rendimiento del sistema.

Por otra parte, y dado que esta tabla recoge los datos de los sensores, los cuales se generan a gran velocidad, se quiere comprobar cuál es número de filas máximo que se puede alcanzar con las distintas alternativas planteadas (ver sección 4.3), ya que, como se explicó anteriormente Cassandra tiene un límite de 2 billones americanos de valores por partición. Esto significa que, cuanto mayor sea el número de particiones existentes más filas de datos soportará el sistema, pero también será más complicado controlar donde se ubican esas particiones. Como se tiene una limitación bastante fuerte en el almacenamiento disponible en el clúster donde se desarrolla esta experimentación, no se puede llegar a realizar esta prueba de forma práctica. Por ello, se hace una estimación del número de filas que se podría alcanzar y lo que esto supone.

Para esto se supondrá un sistema IoT compuesto de los siguientes elementos:

- 100 máquinas
- 20 sensores por máquina
- Cada sensor mide 5 parámetros diferentes
- Las horas se guardan como números enteros del 0 al 23

Con estos datos, se puede hacer una estimación del numero total de filas que llegaría a soportar el sistema, y por tanto determinar si el diseño es válido, teniendo en cuenta que se quieren mantener las medidas generadas durante un año y que los sensores toman 5 muestras por segundo.

Sabiendo el número de filas máximo que soporta el sistema (Ilustración 25), el ritmo al que se generan los datos y el tiempo que se quieren persistir, se puede calcular si alguno de los diseños es apropiado para el sistema. En primer lugar calculamos el número de datos que se generan cada segundo. Para ello, sabiendo que en total hay 2.000 sensores, que miden 5 parámetros con una frecuencia de 5 muestras por segundo obtenemos que se

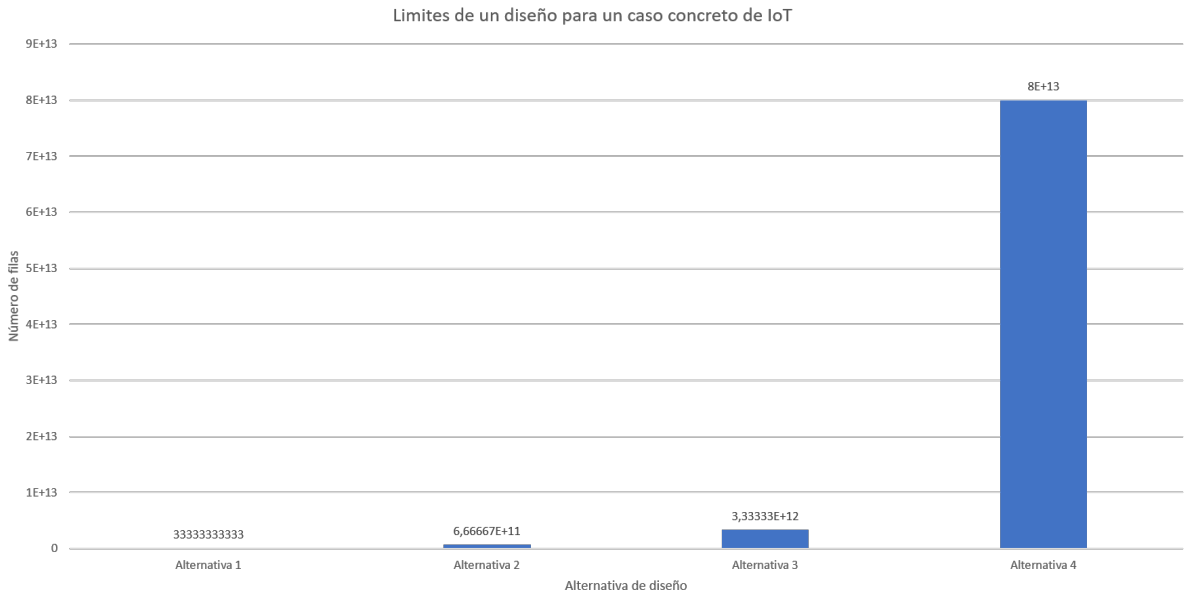

Ilustración 25: Query 3: Limitaciones del diseño para un caso de IoT con diferentes *partition key*

Limitaciones de las diferentes alternativas de diseño

N°de Alternativa	1	2	3	4
N°de Particiones	100	2000	10000	240000
Límite de filas de la tabla	$3,33 \cdot 10^{10}$	$6,66667 \cdot 10^{11}$	$3,33333 \cdot 10^{12}$	$8 \cdot 10^{13}$

Tabla 1: Consulta 3: Limitaciones de las diferentes alternativas de diseño

guardan 50.000 muestras por segundo. En un año, que es el tiempo que deben perdurar los datos, hay 31.536.000 segundos. Por tanto, al final del año el sistema debe ser capaz de gestionar un volumen total de $1,5768 \cdot 10^{12}$ filas. Esto quiere decir que tanto la alternativa 1, como la alternativa 2 se saturarían antes de terminar el año de funcionamiento, siendo válidas para este caso solo las alternativas 3 y 4.

4.4. Resumen y pautas de diseño

A tenor de la experimentación realizada, a continuación se resumen consideraciones de diseño para conseguir el mejor rendimiento del gestor Cassandra:

- Cassandra escala linealmente, por ello, el número de nodos a conectar en un clúster se puede estimar a partir del número de operaciones por segundo que soporta un nodo con la latencia máxima permitida y luego, aumentar el número de nodos hasta el número de operaciones a soportar sobre el caso base.

- Es recomendable utilizar el GossipingPropertyFileSnitch como método para establecer la topología del clúster, ya que permite añadir nodos con facilidad. En caso de usar otro método, sería necesario modificar el fichero *cassadra-topology.properties* de cada nodo manualmente.

- A la hora de diseñar la base de datos, hay que tener en cuenta las consultas que el sistema debe atender, si son preferentemente de consulta o de actualización y el

crecimiento previsto de las tablas con objeto de seleccionar la *Primary Key* adecuadamente. Se ha de tener en cuenta que:

  - Cassandra tiene menores tiempos de respuesta cuando la *Primary Key* solo está compuesta por un campo, y por tanto, las consultas son por key.
  - Conviene diseñar tablas que crezcan más en columnas que en filas. Por lo que la definición de tipos de usuario son una buena alternativa.
  - Cassandra tiene un límite de 2 billones americanos de valores por partición, por lo que convendrá incluir campos en la *partition key* para aquellas tablas que tengan que almacenar volúmenes ingentes de datos.

- El nivel de consistencia también se ha de tener presente. Es importante para garantizar la consistencia de la información pero tiene un coste que se multiplica casi linealmente con el número de nodos que deban confirmar. Se ha de llegar a un *trade-off* entre rendimiento e integridad.

- El nivel de replicación también es importante e incluso imprescindible para garantizar la disponibilidad y recuperabilidad de los datos. Sin embargo, hay que tener en cuenta que esto exige un mayor tráfico de red y de almacenamiento, ya que se generan datos redundantes y comunicación constante entre los nodos.

- Cassandra ofrece la posibilidad de reconfigurar el clúster sin necesidad de parar la actividad, esto permite aumentar el rendimiento de forma sencilla en momentos puntuales de sobrecarga.

5 Cassandra como capa de persistencia de P3forI4

Uno de los objetivos de este proyecto además de analizar y evaluar el comportamiento y rendimiento de Cassandra, es comprobar si Cassandra puede ser configurado de forma que satisfaga las necesidades de persistencia de la plataforma de tercera generación P3forI4 planteada por el grupo de investigación ISTR. A saber:

- Atender el volumen, variedad y velocidad a la que se generan los datos del sector industrial.
- Escalabilidad bajo demanda haciendo uso de recursos contratados en la nube.
- Integración y sincronización de datos con requisitos de tiempo real estrictos.
- Políticas de seguridad y protección de los datos gestionados.

Los entornos industriales están compuestos por una ingente cantidad de sistemas que generan datos con gran rapidez y, actuadores en los que hay requisitos temporales de cuasi tiempo real. Por otro lado, tienen la ventaja de ser entornos poco cambiantes y con un volumen total de dispositivos estable. Por tanto, para implementar una base de datos Cassandra optimizada para funcionar en estos entornos el planteamiento es diferente. A continuación se proponen una serie de pautas para que el diseño de la base de datos se ajuste a estas necesidades.

Debido a las restricciones temporales, es necesario reducir al mínimo la sobrecarga impuesta por la red. Para ello se ha de ubicar los datos en el nodo más cercano a los consumidores del mismo. De esta forma, las consultas se hacen directamente a este nodo y como se ha visto en el análisis realizado previamente, esto supone una menor latencia. Cassandra no está diseñado para funcionar de esta manera, pero se han encontrado dos alternativas para solucionar esto:

- Crear un campo *bucket* ajeno a la problemática, que permita controlar el valor de la *partition key*.
- Crear una nueva implementación de la interfaz IPartitioner para sustituir al *partitioner* por defecto de Cassandra, el MurMur3Partitioner.

La primera de las opciones se puede llevar a cabo gracias a la herramienta nodetool que incorpora Cassandra. Con la opción getendpoints (ver Ilustración 26), a la que se le indica el *keyspace*, la tabla y la *partition key*, se puede conocer la dirección IP del nodo que es responsable de las filas con esa *partition key*. La segunda opción, consiste en implementar la interfaz *IPartitioner* en Java para que el particionado se haga haciendo uso de un algoritmo propio. Cuando se ha finalizado se debe indicar en el fichero cassandra.yaml donde se encuentra la implementación del *partitioner* y, en caso de que ya hubiera datos

```
Last login: Sun Jun 16 11:05:58 2019 from 172.31.16.34
gestor@CZC7318837D:~$ nodetool getendpoints iot machines 1
172.31.16.68
```

Ilustración 26: Cassandra como capa de persistencia de P3forI4: Herramienta nodetool getendpoints

en el sistema, se ha de vaciar por completo, reconfigurar y volver a cargarlos. Otra de las recomendaciones en relación con la restricciones temporales es que se ha de ajustar los niveles de confirmación lo máximo posible. Como se ha visto, los niveles de confirmación altos afectan negativamente al rendimiento del sistema. Así que es importante escogerlos bien. Como en el entorno industrial es importante la consistencia de los datos y Cassandra solo proporciona consistencia eventual, lo más correcto es aumentar el nivel de consistencia requerido, de manera que se cumpla que la suma del nivel de consistencia de lecturas y escrituras sea igual o mayor que el factor de replicación + 1. Esto garantiza lo que se conoce como consistencia fuerte. Como hay que añadir cierto nivel de consistencia, se puede priorizar un tipo de operación frente al otro. En sistemas IoT o de I4.0 se generan datos con gran velocidad, por tanto, es más importante que se conteste rápido a las escrituras y no tanto a las lecturas.

Por último, se debe tener en cuenta el volumen total de los datos que se van a manejar y durante cuanto tiempo se van a persistir. En entornos industriales, que no aumentan la cantidad de dispositivos, ni crecen con frecuencia, es relativamente sencillo. Esto es importante, ya que al ser el volumen de datos elevado existe el riesgo de saturar una partición, llegando al limite de 2 billones americanos de valores. Por tanto, se debería hacer una estimación que permita evitar este problema, sin hacer que el número de particiones sea tan grande que no se pueda controlar y ubicar como se desea.

5.1. Aplicación a un caso de uso de una distribuidora eléctrica

En este apartado, se incluye un diseño de base de datos realizado, aplicando las consideraciones propuestas en este proyecto, para un caso concreto de Industria 4.0 basado en la gestión de estaciones y subestaciones de una compañía distribuidora eléctrica en la ciudad de Santander. Tanto la plataforma computacional como la de gestión eléctrica están establecidas. El caso de estudio es un supuesto escalado para funcionar en el laboratorio ISTR de la Faculta de Ciencias de la Universidad de Cantabria.

La red eléctrica está compuesta por los siguientes elementos:

- 1 centro del operador del sistema de distribución.
- 2 subestaciones de la ciudad
- 6 subestaciones de distrito
- 24 transformadores de calle
- 500 contadores de los usuarios

Además, se realizó un escalado del tiempo para para simular el funcionamiento del sistema en un menor periodo:

- 1s =>20ms
- 1m =>1.200ms
- 1h =>72.000ms = 72s = 1m 12s
- 1d =>1.728.000ms = 1.728s = 28m 48s

Para gestionar esto se establece el despliegue de Cassandra (ver Ilustración 27) en 11 nodos, repartidos en 2 *datacenters*. El primero (fieldDataCenter) es en el que se encuentran los nodos de los diferentes distritos y los de la ciudad, y el segundo (dsoDataCenter) los nodos de la central.

El fieldDataCenter se divide en dos *racks* en uno se agrupan los nodos de los distritos (dsRack) y en otro los de la ciudad (csRack). Esta configuración fuerza a que los nodos de los diferentes distritos repliquen sus datos en los nodos de la ciudad.

El dsoDataCenter está compuesto por 3 racks, dos de ellos con nodos situados en la central (dsoMainRack y dsoAuxRack) y el otro alojado en la nube, para el despliegue en el laboratorio se prescindiría de este último. De este modo, se persigue usar el servidor dsoMainServer como principal y guardar réplicas de los datos en el servidor dsoAuxServer para poder actuar en caso de fallo del sistema.

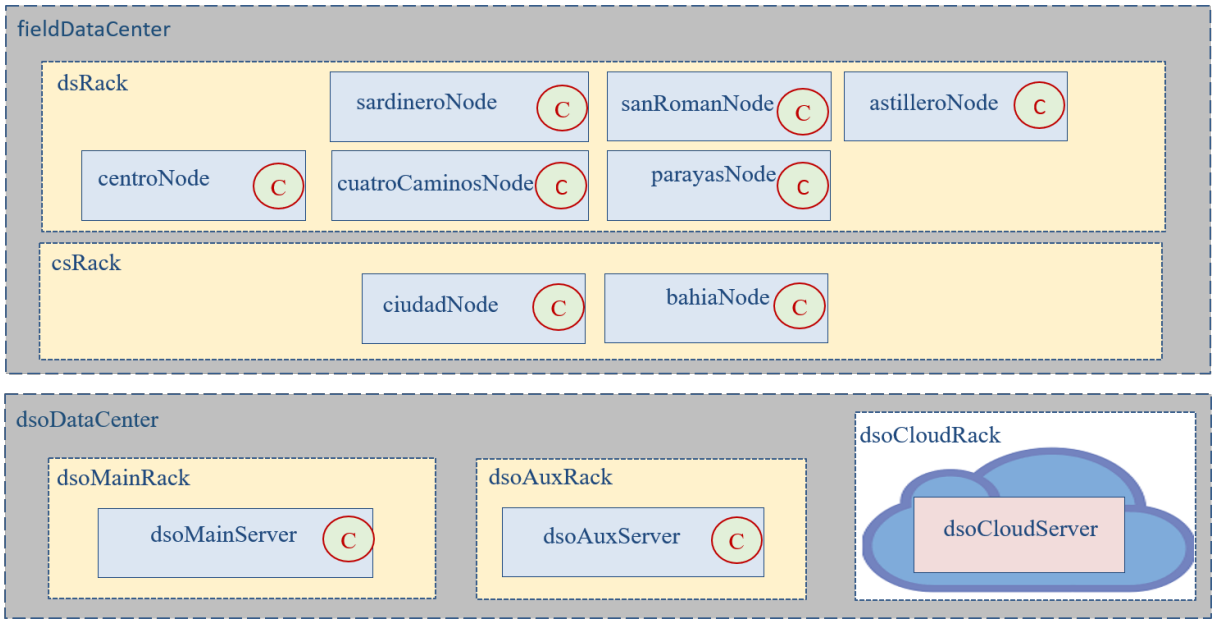

Ilustración 27: Caso de estudio Distribuidora eléctrica: Despliegue de Cassandra

El diseño tiene que ser capaz de dar soporte a las consultas de las diferentes tareas del sistema. Estas consultas se describen en el modelo de la ilustración 28. En el modelo aparecen consultas de diferentes colores, esto se debe a que se configuran en diferentes *keyspaces* por cuestiones semánticas. En este caso se abordarán solamente las consultas relacionadas con la gestion de energía (color azul en el modelo).

Como resultado de ese modelo se ha diseñado un *keyspace* con cinco tablas. El *keyspace* tiene factor de replicación 2 en cada *datacenter* para conseguir el efecto de replicación comentado en la descripción del despliegue. Las tablas diseñadas son las siguientes (diseño completo en el Anexo C):

- UG_by_cup: permite conocer la información del contador de un usuario a través del CUP (identificador único del contrato). Los datos de esta tabla deben de estar alojados en el distrito al que pertenecen. Por tanto, para su diseño se ha incluido un campo *bucket* que permita controlar la partición para enviarlos al nodo correcto.

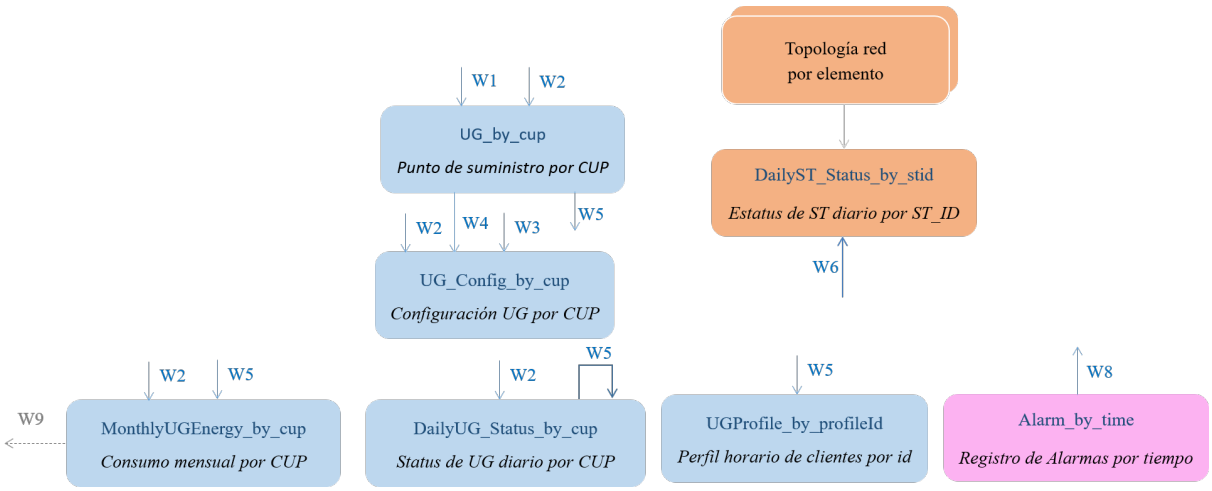

Ilustración 28: Caso de estudio Distribuidora eléctrica: Modelo de consultas

- UG_Config_by_cup: Permite conocer la configuración del contador de un usuario mediante el CUP. Al igual que el anterior, debe situarse en el nodo de su distrito. Por tanto, se aplica la misma estrategia. Además, se debe incluir el mes como *clustering key* para poder consultar por los datos de un mes en concreto, ordenarlos y hacer consultas por rango.

- Monthly_energy_by_CUP: permite conocer el consumo energético del mes de un determinado cliente mediante el cup. Se sigue la misma estrategia que en los dos casos anteriores, pero se incluye también un mapa y un tipo definido por el usuario para evitar un crecimiento excesivo del número de filas, que como se comprobó es perjudicial para el rendimiento del sistema.

- dailyUG_Status_by_CUP: permite conocer el estado del suministro de un cliente de un día concreto. Se sigue los mismos principios que en el caso anterior, añadiendo como *clustering key* el día para poder realizar consultas de un día concreto ordenarlos y hacer consultas por rango.

Posteriormente, se han generado los ficheros de configuración necesario para hacer el despliegue de Cassandra en el laboratorio simulando el esquema de la Ilustración 27. Los detalles de la configuración se incluyen en el anexo D.

6 Conclusiones y líneas futuras

La industria 4.0 tiene por objeto revolucionar la industria de la fabricación y producción gracias a la digitalización de los equipos del entorno industrial, la computación en la nube, la integración de los datos y los avances tecnológicos de los sistemas de producción y fabricación.

Este trabajo, desarrollado en el contexto de un proyecto de investigación que tiene entre sus objetivos proponer una arquitectura tecnológica que de soporte a la Industria 4.0, ha estudiado y determinado estrategias de configuración y diseño del gestor de bases de datos distribuido y escalable Apache Cassandra para ser utilizado como servicio de persistencia en esta nueva arquitectura.

Para ello se ha diseñado una serie de pruebas y se han utilizado herramientas de *benchmark* para comprobar su rendimiento y posibilidades de adaptación a los requisitos de baja latencia que los entornos de tiempo real tienen. Se ha comprobado que, Cassandra es un gestor que ofrece un gran rendimiento y escala linealmente pero no está pensado para funcionar en un entorno industrial en el que hay restricciones temporales y de consistencia tan altas. Sin embargo, gracias a la estabilidad de estos entornos, en los que a pesar de manejar un gran volumen datos la infraestructura no crece con frecuencia, es posible ajustar la base de datos y hacer un diseño específico que se ajuste a las exigencias de estos entornos.

Con las consideraciones de diseño extraídas, se ha diseñado la base de datos para el caso de uso de una distribuidora eléctrica. El siguiente paso será desplegarlo en la plataforma P3forI4 y realizar pruebas adaptadas a este ejemplo. Otro aspecto importante a abordar es el apartado de seguridad que es uno de los aspectos más complejos en los entornos *big data*.

En el plano personal, el desarrollo de este trabajo me ha permitido aprender bastante sobre el funcionamiento y configuración de Cassandra y, sobre buenas prácticas para la realización de *benchmarks*. Además, gracias a la beca de colaboración, he visto desde dentro como trabaja un grupo de investigación de la universidad.

7 Referencias

[1] Cassandralimitations - cassandra wiki, último acceso el 10/06/2019.
https://wiki.apache.org/cassandra/CassandraLimitations

[2] Github - brianfrankcooper/ycsb: Yahoo! cloud serving benchmark, último acceso el 02/06/2019.
https://github.com/brianfrankcooper/YCSB/

[3] About apache cassandra | apache cassandra 3.0, último acceso el 06/06/2019.
https://docs.datastax.com/en/cassandra/3.0/cassandra/cassandraAbout.html

[4] The cassandra-stress tool | apache cassandra 3.0, último acceso el 04/06/2019.
https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCStress.html

[5] The cassandra.yaml configuration file | apache cassandra 3.0, último acceso el 03/06/2019.
https://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/configCassandra_yaml.html

[6] Evolución de la industria, útimo acceso 18/06/2019.
https://www.industriaconectada40.gob.es/SiteCollectionImages/evolucion.jpg

[7] java.com: Java y tú, último acceso el 02/06/2019.
https://www.java.com/es/

[8] Maven - welcome to apache maven, último acceso el 02/06/2019.
https://maven.apache.org/

[9] Welcome to python.org, último acceso el 02/06/2019.
https://www.python.org/

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, y R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.
http://doi.acm.org/10.1145/1807128.1807152

[11] K. Kwang. Third platform shift triggers enterprise software evolution. http://www.zdnet.com/article/third-platform-shift-triggers-enterprise-software-evolution/, 5 2013. último acceso el 18/06/2019.

[12] K.-D. Thoben, S. Wiesner, , y T. Wuest. industrie 4.0 and smart manufacturing a review of research issues and application examples. *International Journal of Automation Technology*, 11(1):4–16, 2017.

[13] N. Velásquez, E. Estevez y P. Pesado. Cloud computing, big data and the industry 4.0 reference architectures. *Journal of Computer Science and Technology*, 18(03):e29, Dec. 2018.
`http://journal.info.unlp.edu.ar/JCST/article/view/1151`

A Fichero de ejemplo de workload para YCSB

```
# Copyright (c) 2012-2016 YCSB contributors. All rights reserved
    .

#

# Licensed under the Apache License, Version 2.0 (the "License")
    ; you

# may not use this file except in compliance with the License.
    You

# may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing,
    software

# distributed under the License is distributed on an "AS IS"
    BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
    or

# implied. See the License for the specific language governing

# permissions and limitations under the License. See
    accompanying

# LICENSE file.



# Yahoo! Cloud System Benchmark
```

```
# Workload Template: Default Values
#
# File contains all properties that can be set to define a
# YCSB session. All properties are set to their default
# value if one exists. If not, the property is commented
# out. When a property has a finite number of settings,
# the default is enabled and the alternates are shown in
# comments below it.
#
# Use of most explained through comments in Client.java or
# CoreWorkload.java or on the YCSB wiki page:
# https://github.com/brianfrankcooper/YCSB/wiki/Core-Properties

# The name of the workload class to use
workload=com.yahoo.ycsb.workloads.CoreWorkload

# There is no default setting for recordcount but it is
# required to be set.
# The number of records in the table to be inserted in
# the load phase or the number of records already in the
# table before the run phase.
recordcount=1000000

# There is no default setting for operationcount but it is
# required to be set.
```

```
# The number of operations to use during the run phase.
operationcount=3000000

# The number of insertions to do, if different from recordcount.
# Used with insertstart to grow an existing table.
#insertcount=

# The offset of the first insertion
insertstart=0

# The number of fields in a record
fieldcount=10

# The size of each field (in bytes)
fieldlength=100

# Should read all fields
readallfields=true

# Should write all fields on update
writeallfields=false

# The distribution used to choose the length of a field
fieldlengthdistribution=constant
#fieldlengthdistribution=uniform
```

```
#fieldlengthdistribution=zipfian

# What proportion of operations are reads
readproportion=0.95

# What proportion of operations are updates
updateproportion=0.05

# What proportion of operations are inserts
insertproportion=0

# What proportion of operations read then modify a record
readmodifywriteproportion=0

# What proportion of operations are scans
scanproportion=0

# On a single scan, the maximum number of records to access
maxscanlength=1000

# The distribution used to choose the number of records to
    access on a scan
scanlengthdistribution=uniform
#scanlengthdistribution=zipfian
```

```
# Should records be inserted in order or pseudo-randomly
insertorder=hashed
#insertorder=ordered

# The distribution of requests across the keyspace
requestdistribution=zipfian
#requestdistribution=uniform
#requestdistribution=latest

# Percentage of data items that constitute the hot set
hotspotdatafraction=0.2

# Percentage of operations that access the hot set
hotspotopnfraction=0.8

# Maximum execution time in seconds
#maxexecutiontime=

# The name of the database table to run queries against
table=usertable

# The column family of fields (required by some databases)
#columnfamily=

# How the latency measurements are presented
```

```
measurementtype=histogram

#measurementtype=timeseries

#measurementtype=raw

# When measurementtype is set to raw, measurements will be
    output

# as RAW datapoints in the following csv format:

# "operation, timestamp of the measurement, latency in us"

#

# Raw datapoints are collected in-memory while the test is
    running. Each

# data point consumes about 50 bytes (including java object
    overhead).

# For a typical run of 1 million to 10 million operations, this
    should

# fit into memory most of the time. If you plan to do 100s of
    millions of

# operations per run, consider provisioning a machine with
    larger RAM when using

# the RAW measurement type, or split the run into multiple runs.

#

# Optionally, you can specify an output file to save raw
    datapoints.

# Otherwise, raw datapoints will be written to stdout.

# The output file will be appended to if it already exists,
    otherwise

# a new output file will be created.

#measurement.raw.output_file = /tmp/
    your_output_file_for_this_run



# Whether or not to emit individual histogram buckets when
```

```
    measuring

# using histograms.

# measurement.histogram.verbose = false



# JVM Reporting.

#

# Measure JVM information over time including GC counts, max and
    min memory

# used, max and min thread counts, max and min system load and
    others. This

# setting must be enabled in conjunction with the "-s" flag to
    run the status

# thread. Every "status.interval", the status thread will
    capture JVM

# statistics and record the results. At the end of the run, max
    and mins will

# be recorded.

# measurement.trackjvm = false



# The range of latencies to track in the histogram (milliseconds
    )

histogram.buckets=1000



# Granularity for time series (in milliseconds)

timeseries.granularity=1000



# Latency reporting.

#
```

```
# YCSB records latency of failed operations separately from
    successful ones.

# Latency of all OK operations will be reported under their
    operation name,

# such as [READ], [UPDATE], etc.

#

# For failed operations:

# By default we don't track latency numbers of specific error
    status.

# We just report latency of all failed operation under one
    measurement name

# such as [READ-FAILED]. But optionally, user can configure to
    have either:

# 1. Record and report latency for each and every error status
    code by

#    setting reportLatencyForEachError to true, or

# 2. Record and report latency for a select set of error status
    codes by

#    providing a CSV list of Status codes via the "
    latencytrackederrors"

#    property.

# reportlatencyforeacherror=false

# latencytrackederrors="<comma separated strings of error codes
    >"



# Insertion error retry for the core workload.

#

# By default, the YCSB core workload does not retry any
    operations.

# However, during the load process, if any insertion fails, the
    entire
```

```
# load process is terminated.

# If a user desires to have more robust behavior during this
    phase, they can

# enable retry for insertion by setting the following property
    to a positive

# number.

# core_workload_insertion_retry_limit = 0

#

# the following number controls the interval between retries (in
     seconds):

# core_workload_insertion_retry_interval = 3



# Distributed Tracing via Apache HTrace (http://htrace.incubator
    .apache.org/)

#

# Defaults to blank / no tracing

# Below sends to a local file, sampling at 0.1%

#

# htrace.sampler.classes=ProbabilitySampler

# htrace.sampler.fraction=0.001

# htrace.span.receiver.classes=org.apache.htrace.core.
    LocalFileSpanReceiver

# htrace.local.file.span.receiver.path=/some/path/to/local/file

#

# To capture all spans, use the AlwaysSampler

#

# htrace.sampler.classes=AlwaysSampler
```

```
#

# To send spans to an HTraced receiver, use the below and ensure

# your classpath contains the htrace-htraced jar (i.e. when
    invoking the ycsb

# command add -cp /path/to/htrace-htraced.jar)

#

# htrace.span.receiver.classes=org.apache.htrace.impl.
    HTracedSpanReceiver

# htrace.htraced.receiver.address=example.com:9075

# htrace.htraced.error.log.period.ms=10000
```

B Fichero de ejemplo de profile para Cassandra-stress

```
#
# Keyspace name and create CQL
#
keyspace: iot
keyspace_definition: |
  CREATE KEYSPACE iot WITH replication = {'class': '
     SimpleStrategy', 'replication_factor' : 1};
#
# Table name and create CQL
#
table: measurements_by_parameter_hour
table_definition: |
  CREATE TABLE measurements_by_parameter_hour (
        idmachine UUID,
        devicename text,
        parameter text,
        value float,
        datetime timestamp,
        hour int,
        PRIMARY KEY ((idmachine,devicename,parameter,hour),
           datetime)) with clustering order by
        (datetime desc)


#
# Meta information for generating data
#
columnspec:
  - name: idmachine
    size: fixed(7)
    population: uniform(1..100)
  - name: devicename
    size: fixed(7) #In chars, no. of chars of UUID
    population: uniform(1..20)
  - name: parameter
    size: uniform(10..20)
    population: uniform(1..5)
  - name: value
```

```
    size: fixed(8)
    population: uniform(1000..2000) # 1000 - 2000 metrics per
       host
  - name: datetime
    population: uniform(1..365)
    cluster: fixed(365)
  - name: hour
    size: fixed(4)
    population: uniform(0..23)

#
# Specs for insert queries
#
insert:
  partitions: fixed(1)       # 1 partition per batch
  batchtype: UNLOGGED        # use unlogged batches
  select: fixed(1)/365       # no chance of skipping a row when
     generating inserts

#
# Read queries to run against the schema
#
queries:
   getmeasurement:
      cql: select * from measurements_by_parameter_hour where
         parameter = ? and hour= ? and idmachine= ? and
         devicename= ? and datetime = ?
      fields: samerow             # pick selection values from
         same row in partition
   last100:
      cql: select * from measurements_by_parameter_hour where
         parameter = ? and hour= ? and idmachine= ? and
         devicename= ? limit 100
      fields: samerow             # pick selection values from
         same row in partition
   last50:
      cql: select * from measurements_by_parameter_hour where
         parameter = ? and hour= ? and idmachine= ? and
         devicename= ? limit 50
      fields: samerow             # pick selection values from
         same row in partition
```

C Caso de estudio de una distribuidora eléctrica: script CQL

```
CREATE KEYSPACE POWERKEYSPACE
  WITH REPLICATION = {
   'class' : 'NetworkTopologyStrategy',
   'dsoDataCenter'  : 2 , // Datacenter 1
   'fileDataCenter' : 2 , // Datacenter 2
   }
   AND DURABLE_WRITES = true ;
```

```
 CREATE TABLE POWERKEYSPACE.UG_by_cup (
    bucket tinyint,
        cup      bigint,
        name     text   ,
        nif      text,
        street   text,
        streetId         text,
        number   text,
        portal   text,
        door     text,
        district         text,
        districtId       bigint,
        city     text,
        region   text,
        postalCode       text,
        country text,
        profileTypeId    bigint,
        power    float , ,
        optionalPower    boolean,
        isCacelled       boolean,
    PRIMARY KEY(bucket,cup)
);
```

```
CREATE TABLE POWERKEYSPACE.UG_Config_by_cup(
        bucket tinyint,
        districtId       bigInt,
```

```
        cup      bigint,
        meterId  bigint,
        streetID         bigint,
        meterModel       text,
        connectInetv     inet,
        userPassword     text,
        powerMax         float,
        optionalPower    boolean,
    PRIMARY KEY(bucket, cup)
);

CREATE TYPE POWERKEYSPACE.DailyEnergyData(
        day      date,
        energy   float,
        extraEnergy      float,
        reactiveEnergy   float
)
CREATE TABLE POWERKEYSPACE.MonthlyEnergy_by_cup(
        bucket tinyint,
        districtID       bigint,
        cup      bigint,
        readMounthYear   date,
        meterID  bigint,
        meterModel       text,
        streetID         bigint,
        initialDay       date,
        powerMax         float,
        dairyEnergy map<int,DailyEnergyData>,
    PRIMARY KEY (bucket, cup, readMounthYear)
)
WITH CLUSTERING ORDER BY (cup ASC, readMounthYear DESC);
--todas las medidas de los cup de un distrito ordenados por cup
   y fecha desc


CREATE TYPE POWERKEYSPACE.HourlyUGStatusData(
        hour     time,
        isValid  boolean,
        meanPower        float,
        flickerPercent   float,
        servicePercent   float,
        numVoltDip       int
)

CREATE TABLE POWERKEYSPACE.dailyUG_Status_by_CUP(
        bucket tinyint,
        districtID       bigint,
        cup      bigint,
        readDay  date,
        meterId  bigint,
```

```
        meterModel      text , ,
        streetId        bigint ,
        initialHour     time ,
        hourlyUGStatus  map<int , HourlyUGStatusData >,

    PRIMARY KEY (bucket , cup , readDay)
)
WITH CLUSTERING ORDER BY (cup ASC, readDay DESC);
```

D Caso de estudio de la distribuidora: Despliegue Cassandra

centroNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.34"
rcp_address: "172.16.31.34"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

sardineroNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.56"
rcp_address: "172.16.31.56"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

cuartroCaminosNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.38"
rcp_address: "172.16.31.38"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

sanRomanNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.54"
rcp_address: "172.16.31.54"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

parayasNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.44"
rcp_address: "172.16.31.44"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

astilleroNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.63"
rcp_address: "172.16.31.63"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=dsRack
```

ciudadNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.49"
rcp_address: "172.16.31.49"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=csRack
```

bahiaNode

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.53"
rcp_address: "172.16.31.53"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=fieldDataCenter
rack=csRack
```

dsoMainServer

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.52"
rcp_address: "172.16.31.52"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=dsoDataCenter
rack=dsoMainRack
```

dsoAuxServer

```
cluster_name: "CassandraCluster"
seeds: "172.16.31.34,172.16.31.49,172.31.16.52"
listen_address: "172.16.31.47"
rcp_address: "172.16.31.47"
endpoint_snitch: GossipingPropertyFileSnitch
```

```
dc=dsoDataCenter
rack=dsoAxuRack
```