UCLV
Universidad Central
"Marta Abreu" de Las Villas

FIE
Facultad de
Ingeniería Eléctrica

Departamento de Electrónica y Telecomunicaciones

TRABAJO DE DIPLOMA

Título: Enfoque Big Data utilizando Kafka y Spark Streaming para el análisis en tiempo real.

Autor: Alián Bernal Alba

Tutores: Dr. Héctor Cruz Enríquez

Ing. Ernesto Pérez Pelaez

Santa Clara, junio del 2019

Copyright©UCLV

UCLV
Universidad Central
"Marta Abreu" de Las Villas

FIE
Facultad de
Ingeniería Eléctrica

Department of Electronics and Telecommunications

TRABAJO DE DIPLOMA

Title: Approaching Big Data using Kafka and Spark Streaming for real-time analytics

Author: Alián Bernal Alba

Thesis Director: Dr. Héctor Cruz Enríquez

Ing. Ernesto Pérez Pelaez

Santa Clara, june 2019

Copyright©UCLV

Este documento es Propiedad Patrimonial de la Universidad Central “Marta Abreu” de Las Villas, y se encuentra depositado en los fondos de la Biblioteca Universitaria “Chiqui Gómez Lubian” subordinada a la Dirección de Información Científico Técnica de la mencionada casa de altos estudios.

Se autoriza su utilización bajo la licencia siguiente:

Atribución- No Comercial- Compartir Igual

Para cualquier información contacte con:

Dirección de Información Científico Técnica. Universidad Central “Marta Abreu” de Las Villas. Carretera a Camajuaní. Km 5½. Santa Clara. Villa Clara. Cuba. CP. 54 830
Teléfonos.: +53 01 42281503-1419

Hago constar que el presente trabajo de diploma fue realizado en la Universidad Central "Marta Abreu" de Las Villas como parte de la culminación de estudios de la especialidad de Ingeniería en Automática, autorizando a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos, ni publicados sin autorización de la Universidad.

Firma del Autor

Los abajo firmantes certificamos que el presente trabajo ha sido realizado según acuerdo de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

Firma del Tutor

Firma del Jefe de Departamento donde se defiende el trabajo

Firma del Responsable de Información Científico-Técnica

PENSAMIENTO

“Lo sabio es la meta del alma humana y a medida que se avanza en sus conocimientos va alejando a su vez el horizonte de lo desconocido.”

HERACLITO

DEDICATORIA

A mis padres que lo son todo.

A mi abuela materna que la quiero tanto.

A mis abuelos paternos.

AGRADECIMIENTOS

A mi familia por todo el apoyo.

A mis tutores por su atención y el conocimiento brindado.

A mis amigos, todos aquellos que de una forma u otra estuvieron siempre presentes.

A todos los profesores que aportaron a mi formación, los cuales recuerdo y agradezco.

TAREA TÉCNICA

- Realización de un estudio profundo de la bibliografía técnico-especializada sobre este tema.
- Caracterización de las tecnologías de Big Data.
- Selección de las tecnologías más adecuadas para la plataforma Big Data.
- Instalación, configuración y puesta en funcionamiento de Kafka, Apache Spark y Cassandra.
- Conexión entre la herramienta Kafka y la fuente de datos externos ThingsBoard.
- Realización de la integración entre todas las tecnologías que conforman la plataforma Big Data.
- Comprobación del desempeño de la plataforma mediante la realización de una aplicación programada en lenguaje Scala.

Firma del Autor

Firma del Tutor

RESUMEN

Big Data se ha convertido en todo un nuevo ámbito para el desarrollo de aplicaciones para el tratamiento de grandes volúmenes de datos.

Con el desarrollo de este proyecto de investigación se propone una herramienta Big Data para el tratamiento específico de datos de sensores de una plataforma ThingsBoard de Internet de las Cosas, la arquitectura se creó basada en la profundización del estudio de las tecnologías para un sistema Big Data en tiempo real, el estudio permitió la elección de las herramientas más adecuadas para la creación de la arquitectura.

En el desarrollo de la plataforma se instalaron y configuraron todas las tecnologías seleccionadas, y la evaluación del sistema se realizó mediante el desarrollo de una aplicación capaz de tomar los datos de la plataforma de Internet de las Cosas mediante la herramienta Kafka, procesarlos en Spark Streaming y almacenar el resultado del procesamiento en la base de datos Cassandra.

Palabras claves: Big Data, Kafka, Spark Streaming, Cassandra, tiempo real.

TABLA DE CONTENIDOS

PENSAMIENTO v

AGRADECIMIENTOS vii

TAREA TÉCNICA viii

RESUMEN ix

INTRODUCCIÓN 1

CAPÍTULO 1. INTRODUCCIÓN A LAS TECNOLOGÍAS DE BIG DATA 6

1.1 Descripción de las tecnologías 6

1.1.1 Apache Flume 6

1.1.2 Apache Kafka 7

1.1.2.1 Arquitectura y componentes 7

1.1.3 Apache Storm 10

1.1.4 Apache Spark 10

1.1.4.1 RDDs 11

1.1.4.2 Componentes de Spark 12

1.1.5 Cassandra 15

1.1.5.1 Arquitectura 16

1.1.5.2 Modelo de datos 17

1.1.6 MongoDB 17

1.2 Comparación de las tecnologías según la función en el flujo de la información 18

1.2.1 Adquisición de los datos 18

1.2.2 Procesamiento y análisis de los datos 19

1.2.3 Almacenamiento 21

1.3 Conclusiones del capítulo 23

CAPÍTULO 2. ARQUITECTURA E IMPLEMENTACIÓN DE LA PLATAFORMA BIG DATA 24

2.1 Arquitectura propuesta....24

2.1.1 Entorno de trabajo....25

2.1.2 Requerimientos....26

2.2 Instalación, configuración y funcionamiento de Apache Kafka....28

2.3 Instalación e inicialización de Apache Spark....35

2.4 Instalación y funcionamiento de Cassandra....36

2.5 Integración de las Tecnologías....39

2.5.1 Integración de ThingsBoard y Kafka....39

2.5.2 Integración de Kafka y Spark Streaming....40

2.5.3 Integración de Spark y Cassandra....41

2.6 Desarrollo de la aplicación....42

CAPÍTULO 3. COMPROBACIÓN DEL DESEMPEÑO DE LA PLATAFORMA BIG DATA 46

3.1 Despliegue de las Tecnologías....46

3.2 Funcionamiento de la plataforma Big Data....50

CONCLUSIONES Y RECOMENDACIONES....55

Conclusiones....55

Recomendaciones....56

REFERENCIAS BIBLIOGRÁFICAS....57

ANEXOS....60

Anexo I Configuración del complemento Kafka en Thingsboard.....60

Anexo II Código del archivo built.sbt.....62

Anexo III Código de la aplicación.....63

INTRODUCCIÓN

Con la reciente aparición de tecnologías emergentes como Internet de la Cosas (IoT, del inglés *Internet of Things*) y el crecimiento vertiginoso de Internet han provocado un crecimiento masivo de los datos y se espera que crezcan aún más en los próximos años. Esta información recolectada que antes no estaba accesible, ahora se almacena y nos permite hacer uso de ella y convertir el valor existente en estos datos en verdadero conocimiento. Es aquí donde surge el término de Big Data pues permite analizar toda aquella información que no puede ser procesada utilizando las herramientas tradicionales. Entonces se puede describir Big Data como el almacenamiento, procesamiento y análisis de grandes volúmenes de datos, que pueden estar estructurados o no estructurados, para encontrar información y aprender de ellos.

Para caracterizar el sistema y explicar las ventajas de Big Data se utilizan cinco reglas o directrices conocidas como las 5 V's[1]:

- **Volumen**: Es la primera característica que se piensa cuando se habla de Big Data debido a que se trabaja con un volumen de datos enorme. Masivas cantidades de datos son generados cada día y este volumen continúa en aumento.
- **Velocidad**: Los datos que se generan fluyen a velocidades muy altas y es necesario realizar su análisis a alta velocidad. Es aquí donde entra el análisis en tiempo real o *streaming* con el objetivo de tomar decisiones más oportunas donde este tipo de análisis marca la diferencia.
- **Variedad**: Los datos poseen muchos tipos de estructuras y formatos. Es esta la razón por la que Big Data debe estar preparado para poder analizar una amplia variedad de datos: desde datos estructurados, no estructurados, datos numéricos, audio, video, documentos de texto, etc.
- **Veracidad**: Existen ocasiones donde es inevitable dudar sobre qué tan veraz son los datos que se tienen, entonces es necesario filtrar estos datos y asegurarse de la veracidad de la información contenida para su posterior aprovechamiento. Dependiendo de la aplicación, la importancia de la veracidad puede ser imprescindible o simplemente un acto secundario que no sea completamente necesario.

- **Valor**: Es el aspecto más relevante de Big Data, con el aumento del volumen y la complejidad de los datos, su valor se hace mucho más costoso de obtener, debido a las dificultades para su explotación.

Todo lo explicado anteriormente permite enfocar el estudio bibliográfico en los principales trabajos realizados sobre Big Data y las herramientas que son utilizadas para crear una arquitectura de este tipo, y más específicamente las que están enfocadas en el análisis en tiempo real de los datos.

En el proyecto [2] se implementó en un clúster Big Data una plataforma capaz de recoger y procesar datos IoT en tiempo real. Los datos son recibidos por el bróker MQTT Mosquitto en el *topic* MQTT "mqtt". Los mensajes son trasladados a un *topic* Kafka mediante un módulo programado en Scala. Una vez procesados los datos en Kafka por una red neuronal *feed forward* son volcados en otro *topic* Kafka para ser consumidos por un servidor *web Flask*. Los datos son enviados en tiempo real mediante *WebSocket* a la página *dashboard* para la visualización del cliente.

En el trabajo [3] se implementa un sistema de *streaming* en Big Data para hacer un análisis de sentimiento de mensajes de la red social Twitter. Se seleccionó una lista de palabras que expresan emociones con la que se han comparado dichos mensajes. Twitter proporciona una interfaz de programación de aplicaciones (API, del inglés *Application Programming Interface*) para la extracción de los *tweets*. Se programa en Java un *KafkaTweetProducer* que accede a los tuits y los deja en Kafka, y un *SparkTweetConsumer* que deja los datos en Spark para ser procesados y depositados en la base de datos MySQL, donde son accedidos y presentados gráficamente por la aplicación *VisualizacionGrafica*. En otro trabajo [4] se implementa una arquitectura de Big Data en IoT para campos de Cadenas de Suministro Inteligentes, los datos se ingieren en el sistema de archivos distribuidos de Hadoop (HDFS, del inglés *Hadoop Distributed File System*), donde se analizan utilizando Apache Hive, Impala y Spark, y para la visualización de los datos se utiliza el cliente de Impala llamado Hue e Impala Shell que pueden interactuar con Impala, Tableau y R para las predicciones.

En el proyecto [5] se implementa una arquitectura Big Data para el análisis de datos en tiempo real. Se utiliza Docker para conformar el clúster, compuesto por tres nodos de Apache Zookeeper, al igual que de Apache Kafka y tres nodos de Apache Storm, uno para el

procesamiento de los datos, un único nodo encargado de controlar el clúster, y un nodo adicional para acceder a la interfaz gráfica de Apache Storm. Se implementan dos aplicaciones en Java, una para actuar de productor de datos para Apache Kafka y la otra es la de Apache Storm encargada de leer los datos desde Apache Kafka.

En el trabajo [6] se implementa un sistema para la detección de fraude bancario en tiempo real. Se desarrolla un proceso en Scala que se encarga de generar cargos en tarjetas de créditos de forma pseudoaleatoria (aproximación aleatoria de datos) y enviar este mensaje a Kafka. Los datos son leídos de las colas Kafka para ser procesados mediante otro proceso implementado en Scala que crea un Spark *Streaming consumer* donde le es aplicado el algoritmo de Luhn y un algoritmo para la detección de anomalías, y se almacenan en MongoDB.

En [7] se realiza un análisis de las arquitecturas existentes para el procesado de *streaming* con enfoque Big Data. Luego se implementan casos prácticos haciendo uso de Apache Spark programados en diferentes lenguajes como Scala y Python. En otro proyecto [8] se desarrolla una herramienta de análisis, procesado y visualización de datos donde se implementan dos APIs de Kafka: *Producer* y *Consumer*. Spark Streaming realiza el procesado de los datos y los almacena en Cassandra. Luego se puede visualizar mediante la herramienta Tableau.

La realización del presente trabajo se enmarca en el tema de Big Data y el empleo de herramientas para el procesamiento de datos en tiempo real. La novedad del tema puede resultar en el potencial desarrollo de un trabajo que no ha sido desarrollado con anterioridad en la Universidad Central “Marta Abreu” de Las Villas y por tanto no se cuenta con ninguna plataforma de este tipo para el procesamiento de datos en tiempo real.

La conformación de una plataforma Big Data requiere de un estudio sobre las principales componentes y cuáles son las más adecuadas según la necesidad de estudio. Además, existen diferentes plataformas comerciales Big Data disponibles. Estas plataformas vienen implementadas, y brindan varias herramientas como el almacenamiento, análisis y visualización de datos. Son plataformas bastante costosas y la creación de una propia que sea capaz de analizar grandes cantidades de datos en tiempo real ofrece grandes ventajas en su uso. Para lograr una arquitectura en tiempo real como se desea, existen muchas variantes y arquitecturas diferentes y no todas brindan las mismas facilidades, se necesita una

arquitectura que permita el análisis de flujos de datos en tiempo real, de manera escalable y con un alto rendimiento. Es por ello que se plantea el siguiente problema científico: ¿Cómo implementar una plataforma de Big Data para el procesamiento de datos en tiempo real?

Objetivos

Luego de plantear la situación problémica y el problema científico de este proyecto se proponen los siguientes objetivos:

Objetivo general

- Implementar una plataforma Big Data con Kafka y Spark Streaming para el análisis de datos en tiempo real.

Partiendo del objetivo general surgen los siguientes **objetivos específicos**:

- Describir las plataformas de Big Data de *streaming* de datos e identificar las herramientas que se utilizan para su implementación.
- Caracterizar las herramientas de Big Data que se utilizan en el análisis de *streaming* datos.
- Implementar las herramientas de Big Data con un enfoque de *streaming* para el procesamiento de datos en tiempo real.
- Realizar pruebas en la plataforma de Big Data implementada para comprobar el desempeño.

Partiendo de los objetivos específicos planteados anteriormente surgen las siguientes tareas de investigación:

Tareas de investigación:

- Análisis crítico del estado del arte de la literatura relacionada con las técnicas de recolección, análisis y procesamiento de datos en tiempo real.
- Selección de las principales herramientas a utilizar para la recolección de los datos.
- Realizar la interconexión del sistema de datos primarios con la herramienta de recolección de datos seleccionada.

- Selección de las técnicas a emplear para el análisis y procesamiento de datos en tiempo real.
- Realización de pruebas para comprobar el funcionamiento del sistema.

El trabajo queda estructurado en introducción, capitulario, conclusiones, recomendaciones, referencias bibliográficas y anexos.

En el Capítulo 1, dedicado a la "Introducción a las tecnologías de Big Data**",** se presentan algunos antecedentes de plataformas Big Data de procesamiento de datos en tiempo real y se describen las tecnologías usadas para la implementación de la plataforma. Se hace un análisis crítico de la literatura que fundamenta la selección de las técnicas y tecnologías empleadas.

El Capítulo 2 se enmarca en la "Arquitectura e implementación de la plataforma Big Data", donde se describe la instalación, configuración y manejo de las diferentes etapas de la arquitectura de la plataforma Big Data como parte de los materiales y métodos de la presente investigación. Además, se ofrecen los pasos a seguir para la implementación real del sistema y la puesta en funcionamiento del mismo.

Se concluye con el Capítulo 3 dedicado a la presentación y discusión de los resultados mediante la "Comprobación del desempeño de la plataforma Big Data" para lo cual serán realizadas pruebas de funcionamiento, se discute sobre la evaluación hecha al sistema mediante un trabajo de prueba para comprobar el funcionamiento de la implementación propuesta.

CAPÍTULO 1. INTRODUCCIÓN A LAS TECNOLOGÍAS DE BIG DATA

Existen diversas plataformas Big Data, así como muchas tecnologías en el mercado en estos momentos según la función que desempeñan en una plataforma de este tipo. Como se explicó en la introducción de este proyecto Big Data es capaz de procesar o tratar, volúmenes considerados de información con una relativa facilidad y permite obtener resultados de la información analizada. Si se sigue el flujo de la información en un ambiente Big Data, este está compuesto por bloques funcionales que corresponden a la adquisición de datos, el análisis, el almacenamiento y los resultados. El flujo de la información permite enmarcar el estudio y la búsqueda de las herramientas que se necesitan para conformar una plataforma en cada bloque funcional. Este capítulo trata sobre la realización de un estudio de las principales tecnologías y la selección de una de ellas para cada etapa del sistema Big Data.

1.1 Descripción de las tecnologías

En la actualidad existen diversas tecnologías para formar parte de una plataforma Big Data y cada una de ellas realiza una función específica según la etapa de la que forma parte en la plataforma. Puede ser una herramienta para adquirir datos, para procesar y analizar, almacenar, y hasta visualizar los resultados. Hay herramientas que no son muy buena solución en un escenario en tiempo real, pero el estudio que se realiza está enfocado precisamente en las tecnologías que brindan mejores funcionalidades en tiempo real.

1.1.1 Apache Flume

Apache Flume es un producto que forma parte del ecosistema Hadoop, es un servicio distribuido, confiable y disponible para recopilar, agregar y mover de manera eficiente grandes cantidades de datos de registro a un almacenamiento de datos centralizado. Tiene una arquitectura simple y flexible basada en flujos de datos de transmisión. Es robusto y tolerante a fallas con mecanismos de confiabilidad sintonizables y mecanismos recuperación de fallas[9]. La arquitectura de Flume tiene los siguientes componentes: (1) Agente; (2) Fuente; (3) Canal y (4) Sumidero.

Agente

Un agente es un proceso demonio independiente en el canal de flujo que recibe los datos de los clientes u otros agentes y lo reenvía a su próximo destino (sumidero o agente). El canal

de flujo puede tener más de un agente, un agente contiene tres componentes principales[10]: fuente, canal, y el sumidero.

Fuente

Una fuente es un componente propio de Flume que se encarga de recoger los datos desde una fuente externa en un formato reconocible por Flume y pasárselos transaccionalmente al canal[11].

Canal

Un canal es otro componente de Flume que actúa de almacén intermedio entre la fuente y el sumidero. La fuente será la encargada de escribir los datos en el canal y permanecerán en él hasta que el sumidero u otro canal los consuman, esto es muy importante ya que hace que el flujo de datos sea confiable y tolerante a fallos[11].

Sumidero

Este componente de Flume será encargado de recoger los datos desde el canal intermedio dentro de una transacción y de moverlos a un repositorio externo, otra fuente u otro canal intermedio. Es el punto final donde llegarán los datos[11].

1.1.2 Apache Kafka

Apache Kafka es una plataforma de *streaming* distribuida utilizada para el procesamiento de datos en tiempo real. Kafka se considera como un sistema de mensajería distribuido basado en publicación-suscripción. Posee una serie de características que la hacen diferente de los tradicionales sistemas de mensajería, tales como su desarrollo en Scala, su diseño escalable por naturaleza, alto rendimiento tanto en publicación como en suscripción, permite más de una suscripción y presenta persistencia en disco de los mensajes[12].

1.1.2.1 Arquitectura y componentes

Existen cuatro términos que identifican la arquitectura lógica en Kafka[13]: (1) Tema; (2) Partición; (3) Productor y (4) Consumidor. En la figura 1-1 se muestra la interrelación entre estos componentes.

Un tema es una categoría o nombre de *feed* en el que se publican los registros. Los temas en Kafka son siempre múltiples suscriptores; es decir, un tema puede tener cero, uno o muchos consumidores que se suscriban a los datos escritos en él. Los temas se dividen en particiones,

y una partición es similar a un registro en sí mismo, por lo que sigue las mismas reglas, cada mensaje entrante es agregado por Kafka al final del registro. Aquí los mensajes están ordenados estrictamente por el tiempo y a cada mensaje le es asignado un número de desplazamiento que lo identifica. Hay que tener en cuenta que un tema tendrá generalmente varias particiones por lo que no existe garantía de ordenar los mensajes en las particiones de cada tema, pero el orden de los mensajes dentro de cada partición está garantizado.

Los productores son clientes de Kafka que se encargan de la publicación de mensajes en un tema específico, por lo general a los productores no les interesa en que partición se publicarán los mensajes, pero este puede dirigir los mensajes en algunos casos a una partición especifica usando la clave del mensaje.

Los consumidores son otros clientes que se suscriben y leen mensajes que se publicaron en uno o más temas, estos mantienen un seguimiento del desplazamiento de los mensajes publicados con la agregación de otro bit de metadatos (un valor entero que aumenta continuamente y es agregado por Kafka a cada mensaje que se produce). El desplazamiento del último mensaje consumido en una partición es almacenado ya sea en Zookeeper o en Kafka, y esto permite al consumidor detenerse y volver a consumir sin perder su posición.

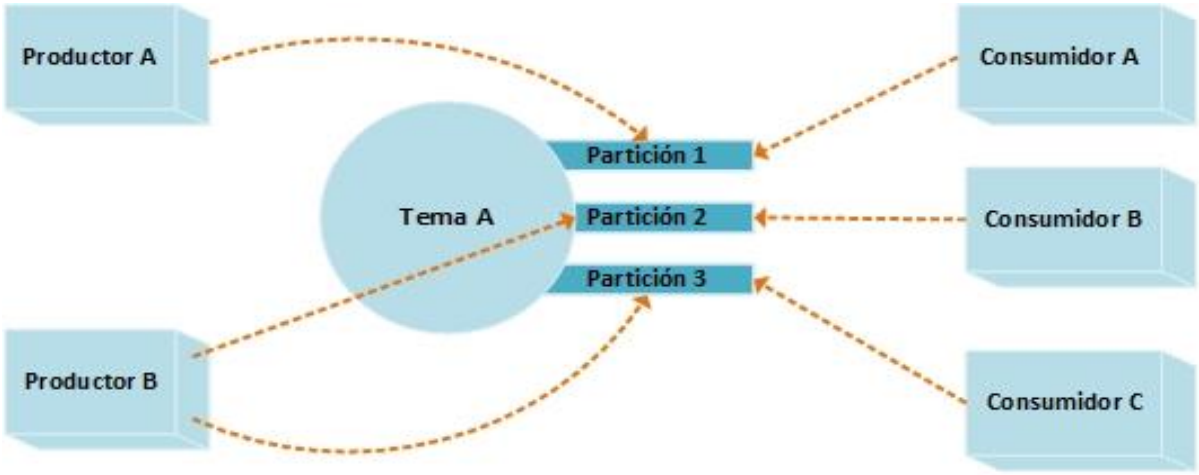


Figura 1-1. Arquitectura lógica de Kafka.

En cuanto a la arquitectura física en un clúster Kafka existen dos componentes que se deben conocer: (1) Bróker y (2) Zookeeper[13].

Un clúster Kafka se puede interpretar como un conjunto de datos recolectados que abarca muchas máquinas diferentes donde se almacenan estos archivos y en correspondencia con esto, un bróker Kafka es el encargado en cada máquina de recibir los mensajes que publican los productores, les asigna desplazamientos y confirmación para almacenar los mensajes en cada tema. De la misma manera sirve a los consumidores atendiendo a sus solicitudes para consumir los mensajes publicados en las particiones de un tema. Es muy común un grupo Kafka compuesto por varios brókeres para equilibrar la carga de trabajo en el clúster [14].

ZooKeeper es un servicio centralizado para mantener la información de configuración, asignar nombres, proporcionar sincronización distribuida y proporcionar servicios de grupo. Todos estos tipos de servicios son utilizados de una forma u otra por las aplicaciones distribuidas [15].

Dado que Kafka es una aplicación distribuida debe quedar claro cómo ZooKeeper está involucrado en la arquitectura de Kafka. Zookeeper es un componente muy importante de un clúster Kafka pues es el encargado de la gestión y coordinación de todos los brókeres activos del clúster y de sus consumidores de tema. Realiza un seguimiento del estado de los nodos del clúster Kafka, cualquier nueva adición de un bróker o fallo que ocurra en el clúster, lo que le permite notificar a los productores y consumidores sobre el estado del clúster.

En la Figura 1-2 se puede observar un caso de un clúster Kafka compuesto por un nodo y dos brókeres a modo de ejemplo.

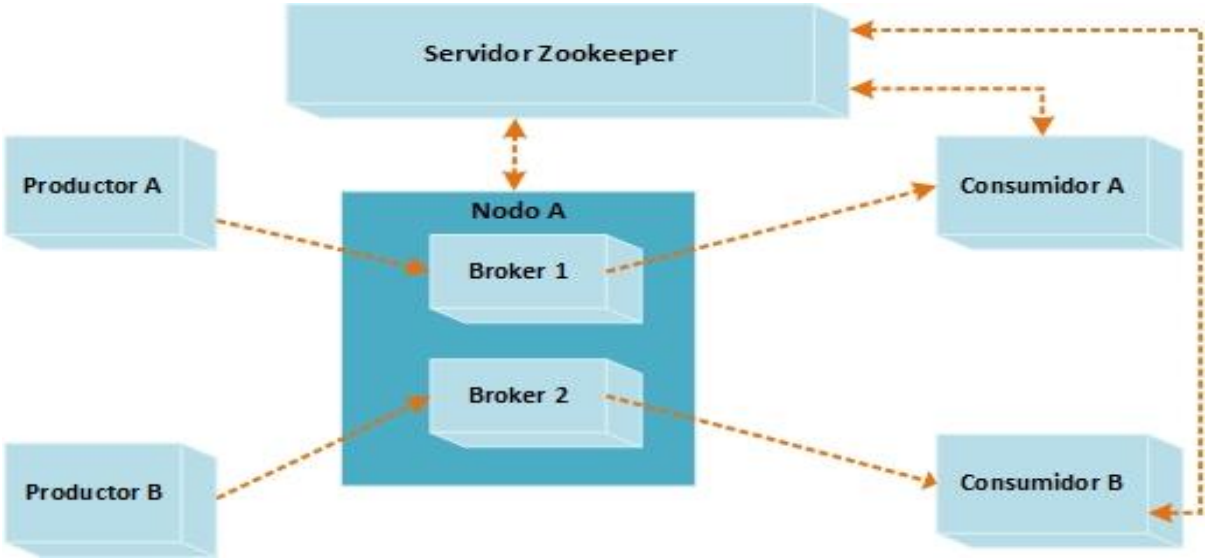


Figura 1-2. Arquitectura física de Kafka.

1.1.3 Apache Storm

Apache Storm es un sistema de computación en tiempo real distribuido y de código abierto que permite el procesamiento de flujos de datos de manera confiable. Storm es simple y se puede utilizar con varios lenguajes de programación como Java, Python, Scala, C#, entre otros. Se puede integrar con bases de datos y con las tecnologías de colas. Procesa los datos en tiempo real, los transforma y los analiza dentro de un proceso de entrada constante de información. Entre sus fortalezas se encuentran que es escalable, tiene alta tolerancia a fallos y garantiza que todos los datos van a ser procesados[16].

Sigue una arquitectura maestro-esclavo donde ninguna instancia mantiene estado y es gestionado por Zookeeper. El maestro es denominado Nimbus y los esclavos se llaman Supervisor. El maestro o Nimbus es el responsable de realizar asignación y monitorización de las distintas tareas en los nodos del clúster, y el esclavo o Supervisor recoge y realiza las tareas que le son asignadas, si uno de ellos falla, Zookeeper es el encargado de avisar al maestro para redirigir las tareas a otro esclavo[3].

Apache Storm está formado por *Topologies*, que son árboles complejos donde se realiza el procesamiento y que a su vez están formadas por *spouts* y *bolts*. Un *spout* es el encargado de recoger el flujo de datos de entrada y el *bolt* contiene las operaciones para realizar los cálculos[3].

1.1.4 Apache Spark

Apache Spark es un *framework* que permite trabajar con Big Data, diseñado para ser rápido, fácilmente accesible y de propósito general que proporciona APIs de alto nivel en Java, Scala, Python y R. Apache Spark es capaz de ejecutar operaciones en memoria hasta 100 veces más rápido que otros sistemas de cómputo. Esto se debe principalmente a la reducción del número de operaciones de lectura y escritura en el disco, y en su lugar aparece el almacenamiento de datos de procesamiento intermedio en memoria que ofrece mayor velocidad, interacción y productividad en la ejecución de las aplicaciones [17], [18].

Spark sigue la arquitectura maestro-esclavo, el demonio maestro llamado *Spark drivers* y los múltiples demonios esclavos llamados *executors*, tanto el *driver* como los diferentes *executors*, tienen su propio proceso Java independiente. Spark está diseñado para ejecutarse

en un clúster y utiliza administradores de recursos de clúster como YARN, Mesos o Spark Standalone que es el manejador de clúster propio de Spark.[19].

La Figura 1-3 muestra la arquitectura de Apache Spark.

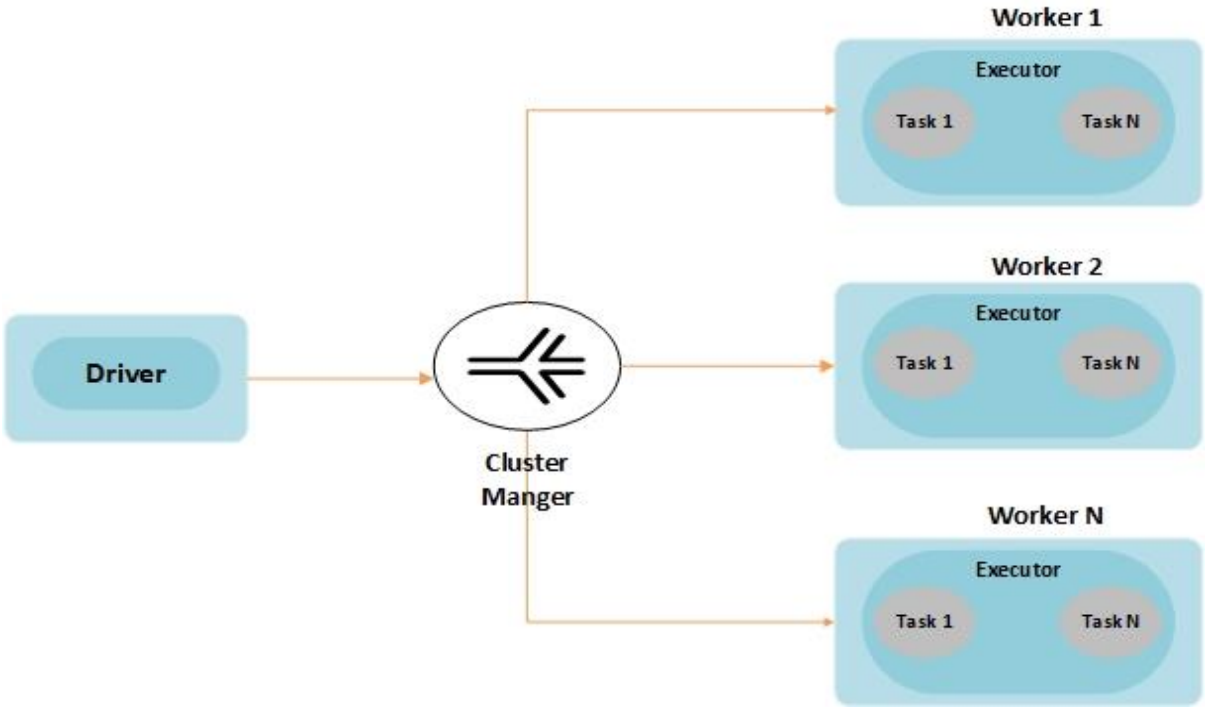


Figura 1-3. Arquitectura de Apache Spark.

1.1.4.1 RDDs

Spark funciona alrededor del término conocido como conjunto de datos distribuidos resistentes (RDD, del inglés *Resilient Distribuited Datasets*), estos son la columna vertebral de Spark y se dividen en particiones lógicas que se distribuyen en diferentes máquinas de trabajo. Los RDDs son un conjunto de datos inmutable, distribuidos y tolerante a fallas cargados en memoria, donde cualquier operación de filtrado en este RDD o modificación producirá un nuevo RDD. También permiten trabajar con grandes cantidades de datos en clúster de una forma rápida y se logra la eficiencia mediante la paralelización del procesamiento en múltiples nodos en el clúster y la minimización de la replicación de datos entre esos nodos [18].

La Figura 1-4 nos brinda una mejor idea de las particiones en los RDDs.

Existen dos tipos básicos de operaciones que se pueden realizar luego que los datos son cargados en un RDD: (1) Transformaciones y (2) acciones.

En las transformaciones se crea un nuevo RDD al modificar el original a través de procesos como mapeo y filtrado. El RDD original permanece inmutable y la cadena de transformaciones de RDD1 a RDDn se registra para que pueda ser repetida en caso de pérdida de datos o falla de un nodo del clúster.

Las transformaciones son realizadas de forma dejada, lo que significa que no se ejecutan y se irán guardando hasta que una acción posterior necesita el resultado y se ejecute. Esto normalmente mejorará el rendimiento, ya que puede evitar la necesidad de procesar datos innecesariamente.

En el caso de las acciones se obtiene un resultado luego de ejecutar una computación sobre el conjunto de datos. Un ejemplo puede ser contar (*count()*), que cuenta, pero no cambian los datos originales [18].

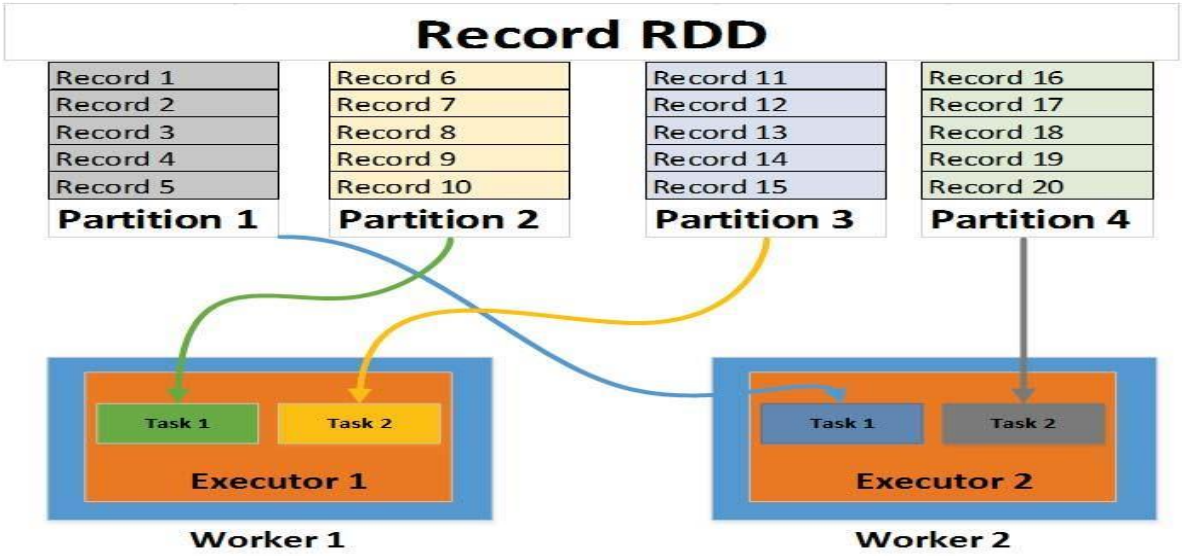


Figura 1-4. Particiones en los RDDs[19].

1.1.4.2 Componentes de Spark

Spark se compone por Spark SQL, Spark Streaming, MLib y procesamiento de grafos con GraphX, donde todos están apoyados en el núcleo de Spark Core y por tanto ofrece la ventaja que cualquier mejora en el *core* beneficiará las API superiores.

La Figura 1-5 muestra los componentes de Spark[20].

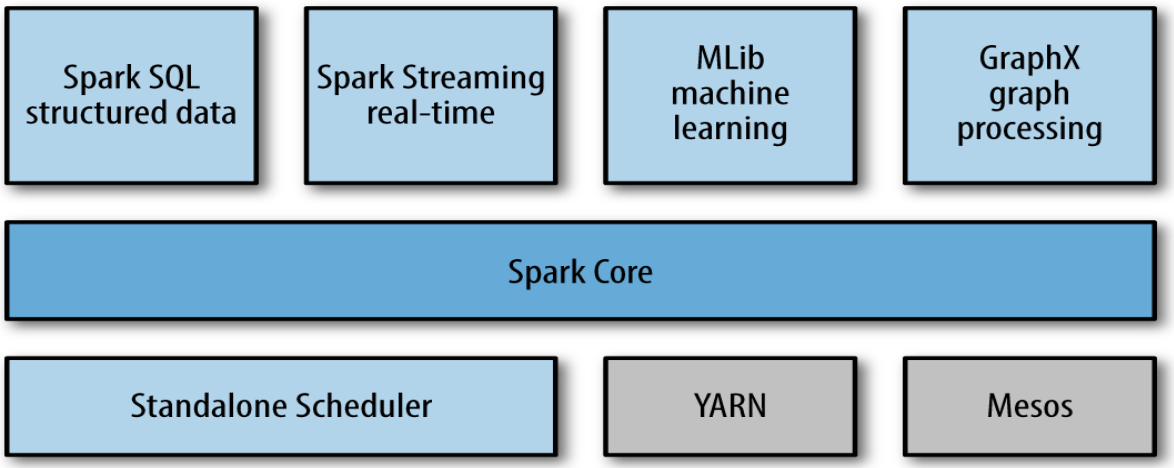


Figura 1-5. Componentes de Spark [20].

Spark Core

Es la capa base del ecosistema Spark que contiene sus funcionalidades, sus componentes para la programación y administración de tareas, la recuperación ante fallos, la interacción con los sistemas de almacenamiento, etc. Spark Core proporciona muchas API para construir y manipular RDD, que es la abstracción principal de la programación de Spark [20].

Spark SQL

Es un módulo que permite el procesamiento de datos estructurados y semi-estructurados mediante sentencias SQL o una API DataFrame. A diferencia de la API básica de Spark RDD, las interfaces proporcionadas por Spark SQL proporcionan a Spark más información sobre la estructura de los datos y el cálculo que se realiza, lo que permite utilizar esta información internamente para realizar optimizaciones adicionales [21].

MLlib

Es una biblioteca básica de Spark, un *framework* que tiene algoritmos de alta calidad que hacen que el aprendizaje automático sea fácil de usar e implementar, escalable y práctico. Posee algoritmos comunes de aprendizaje como clasificación, regresión, agrupación y filtrado colaborativo [22].

GraphX

Es un nuevo componente en Spark para gráficos y cómputo gráfico-paralelo, flexible y veloz. Para admitir el cálculo de gráficos GraphX expone un conjunto de operadores fundamentales (por ejemplo, *subgraph*, *joinVertices* y *aggregateMessages*. Además, GraphX incluye una creciente colección de algoritmos de gráficos y constructores para simplificar las tareas de análisis gráfico [23].

Modo Clúster

Spark está diseñado para escalar eficientemente de uno a muchos miles de nodos de cómputo. Para lograr esto, al tiempo que maximiza la flexibilidad, Spark puede ejecutarse en una variedad de administradores de clústeres, incluidos Hadoop YARN, Apache Mesos, y un administrador de clústeres simple incluido en el mismo Spark llamado Standalone Scheduler (clúster autónomo). Si solo está instalando Spark y no se cuenta con otro administrador de clúster, el Standalone Scheduler proporciona una manera fácil de comenzar [20].

Spark Streaming

Spark Streaming es una extensión del Core Spark API que permite procesar flujos combinando *streaming* con consultas por lotes e interactivas, esta ventaja proporciona que los datos sean analizados conforme entran, sin tiempo de latencia y a través de un proceso de gestión en continuo tránsito, tolerante a fallos, escalable y de alto rendimiento. Estas ventajas se deben a que su núcleo tiene una alta velocidad de programación. Los datos se pueden introducir de muchas fuentes como sockets, Kafka, Flume, Kinesis o TCP, y se pueden procesar gracias a la API que trabaja con Java, Scala y Python mediante algoritmos complejos expresados con funciones de alto nivel como *map*, *reduce*, *join* y *window* [24].

Internamente, Spark Streaming primero recibe flujos de datos de entrada en tiempo real y se dividen estos datos en lotes, luego son procesados por el motor de Spark para generar el flujo final en lotes como resultado [24]. En la Figura 1-6 se observa este proceso.

Figura 1-6. Flujo de datos de Spark Streaming [24].

Los DStreams o flujo discretizado son una abstracción de alto nivel proporcionada por Spark Streaming, que representa un flujo continuo de datos. Internamente un DStream se representa como una secuencia de RDD.[24] Cada RDD en un DStream contiene datos para un cierto intervalo de tiempo, esto se observa en la Figura 1-7.

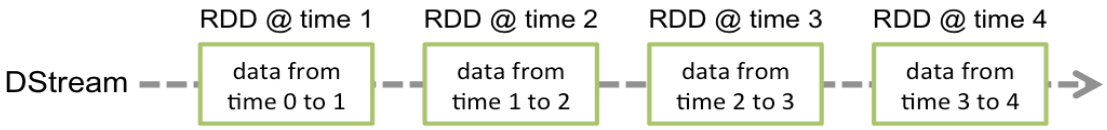


Figura 1-7. DStreams [24].

Como se mencionó al comienzo de esta sección, Spark streaming proporciona una API para recibir datos y aplicar algoritmos complejos sobre ellos. Los datos de transmisión pueden provenir de cualquier fuente mencionada anteriormente. Básicamente, existen dos enfoques[19]: (1) Enfoque basado en el receptor y (2) Enfoque directo. El enfoque directo proporciona ventajas sobre el enfoque basado en el receptor como: no presenta registro de escritura anticipada lo cual evita la pérdida de datos, no utiliza Zookeeper y en su lugar Spark utiliza un mecanismo de punto de control (*checkpoint*) para lidiar con la pérdida de datos y para iniciar la ejecución desde el último punto de ejecución en caso de falla, además, brinda exactamente un procesamiento lo que significa que no se procesan datos dos veces y no se pierden datos [19].

1.1.5 Cassandra

Apache Cassandra es un sistema de gestión de bases de datos distribuido y código abierto diseñado para manejar grandes cantidades de datos en muchos servidores. Proporciona alta disponibilidad sin un solo punto de falla, es un sistema robusto y replicado lo que permite operaciones de baja latencia para todos los clientes.

1.1.5.1 Arquitectura

La arquitectura de Cassandra es totalmente *peer-to-peer* en sus nodos, y los datos se distribuyen entre todos los nodos en un clúster, donde todos los nodos desempeñan el mismo papel. Cada nodo es independiente y al mismo tiempo está interconectado a los demás. En el clúster de Cassandra cada nodo puede aceptar solicitudes de lectura y de escritura, sin importar donde se encuentre realmente los datos en el clúster y cuando un nodo cae, las solicitudes de lectura y escritura se pueden atender desde otros nodos de la red. Los usuarios pueden acceder a Cassandra a través de sus nodos utilizando el lenguaje de consulta de Cassandra (CQL, del inglés *Cassandra Query Language*) [25].

Replicación de los datos

Cassandra tiene una replicación bastante poderosa, y como ya se comentó antes, todos los nodos son tratados de la misma manera, y como no existe una arquitectura maestro- esclavo la replicación ocurre de forma asíncrona y los datos no tienen que estar escritos en un servidor especifico ni hay que esperar a que los datos se repliquen en todos los nodos. Existe la capacidad de elegir cuantos nodos se utilizan para leer y escribir, esto da la posibilidad de modificarlo para lograr una latencia muy baja. Es importante destacar que Cassandra usa el protocolo Gossip en segundo plano para permitir que los nodos se comuniquen entre sí y detectar cualquier nodo defectuoso en el clúster [26].

Componentes

Los componentes claves en Cassandra son: (1) Nodo; (2) Centro de datos; (3) Clúster; (4) Registro de confirmación; (5) *MemTable*; (6) *SSTable* y (7) *Bloom filter* [27]. El nodo es el lugar donde se almacenan los datos. El centro de datos es una colección de nodos relacionados y el clúster es el componente que contiene uno o más centros de datos. El registro de confirmación es un mecanismo de recuperación de fallos en Cassandra donde se escribe cada operación de escritura. La *MemTable* es una estructura de datos residente en la memoria donde se escriben los datos después de pasar por el registro de confirmación. La *SSTable* es un archivo de disco en el que se descargan los datos de la *MemTable* cuando su contenido alcanza un valor umbral. Y el *Bloom filter* es una estructura de datos probabilística que permite a Cassandra determinar uno de dos estados posibles: (1) Los datos definitivamente no existen en el archivo dado y (2) Los datos probablemente existen en el archivo dado [25].

1.1.5.2 Modelo de datos

El modelo de datos de Cassandra es significativamente diferente de lo que normalmente se observa en un sistema de gestión de bases de datos relacionales (RDBMS, del inglés *Relational Database Manager System*). En esta sección se pretende proporciona una visión general de cómo Cassandra almacena sus datos.

La base de datos de Cassandra funciona como un clúster, donde se distribuye en varias máquinas que operan juntas. Para el manejo de fallas, cada nodo contiene una réplica, y en caso de una falla, la réplica se hace cargo. Los nodos en Cassandra son organizados en el clúster en un formato de anillo, y aquí les son asignados los datos [25].

Cassandra tiene tres contenedores, uno dentro del otro. El contenedor más externo es el Keyspace. Se puede pensar en Keyspace como una base de datos en el terreno RDBMS, normalmente hay un Keyspace por aplicación. Luego se ve la familia de columnas, que es como una tabla, este es un contenedor de una colección de filas. Cada fila contiene columnas ordenadas las cuales son la unidad básica de almacenamiento consistente en una estructura de tres valores: *name*, *value* y *timestamp*. Cada fila se identifica mediante una clave de fila única que es como la clave primaria en RDBMS. Las familias de columnas representan la estructura de sus datos y cada Keyspace tiene al menos una familia de columnas.

Un Keyspace en Cassandra tiene los siguientes atributos básicos: (1) Factor de Replicación, que es el número de máquinas en el clúster que reciben los datos y (2) *Replica placement strategy*, que indica cómo se colocan las réplicas en el anillo: (2-1) *SimpleStrategy* (solo un centro de datos) y (2-2) *NetworkTopologyStrategy* (varios centros de datos).

1.1.6 MongoDB

MongoDB es una base de datos orientada a documentos, este tipo de bases de datos guarda los datos en documentos en lugar de en registros y estos documentos son almacenados en BSON, que es una representación binaria de JSON (del inglés *JavaScript Object Notation*). Es una de las bases de datos No-Sql más fácil de aprender a manejar, por su lenguaje simple.

Entre sus propiedades se encuentra la replicación que permite tener dentro de la misma base de datos varias copias de un dato. Las réplicas pueden tener distintos roles, primario y secundario, donde todas las acciones de lectura y escritura se realizan sobre la primaria[6].

La consistencia es otra de sus propiedades. Cuando se despliega esta base de datos sobre un ambiente distribuido se debe tener en cuenta que varios nodos participan en la validación de las operaciones de escritura. Si se necesita una base de datos de alta consistencia donde se realizan varias operaciones de escritura, el número de nodos validadores debe ser elevado, y en caso de que prime la rapidez en lectura, el número de nodos validadores debe ser menor[6].

El *Sharding* es otra propiedad de MongoDB, esta es una técnica que se encarga de dividir un conjunto de datos y distribuirlos sobre múltiples servidores o "shards". Cada servidor es una base de datos independiente, y de forma conjunta estos servidores actúan como una única base de datos lógica[6].

En las bases de datos MongoDB los datos son almacenados en unas estructuras llamadas colecciones, y sobre estas se pueden realizar operaciones como[6]: (1) Insertar documentos con el comando *insert*; (2) Actualizar documentos mediante el comando *update*; (3) Realizar consultas con el comando *find*, el cual permite la búsqueda de documentos en la base de datos y (4) Crear índices que permite mejorar los tiempos de consulta.

1.2 Comparación de las tecnologías según la función en el flujo de la información

1.2.1 Adquisición de los datos

La primera etapa de la plataforma es la recolección de los datos. Existen diversas herramientas hoy en día que pueden utilizarse para la recolecta de datos hacia un entorno Big Data como Apache Kafka o Apache Flume. Según la arquitectura de los tradicionales sistemas de mensajería se pueden mencionar ActiveMQ o RabitMQ. En la **Tabla 1** se hace una comparación entre Apache Kafka y Apache Flume teniendo en cuenta sus principales características. Se escogieron estas dos plataformas para esta comparación por ser tecnologías punteras y de gran uso hoy en día en múltiples plataformas de Big Data.

Apache Kafka está diseñado para ser un sistema de propósito general mientras que Flume es una herramienta de propósito especial para Hadoop, y no es muy escalable en comparación con Kafka, ya que implica la pérdida de datos cuando los usuarios no pueden mantenerse actualizados. Agregar más usuarios a Flume significa cambiar la topología del diseño del canal de Flume, replicando el canal para enviar los mensajes a un nuevo receptor, por lo que no es realmente una solución escalable cuando tiene una gran cantidad de consumidores [28].

TABLA 1. Características de las herramientas de recolección de datos [9], [12].

Características	**Apache Kafka**	**Apache Flume**
Pre-procesamiento	Si	Si
Garantía de entrega	Al menos una vez	Al menos una vez
Tolerante a fallos	Si	Si
Lenguajes de programación	Java	Java
Documentación	Extensa	Extensa
Escalable	Si	No tanto
Replicación de datos	Si	No
Tiempo Real	Si	Si
Latencia	Muy baja	Muy baja

Finalizada esta comparación se puede concluir que el uso de Apache Kafka para la recolección de los datos es la mejor opción para este sistema por las características explicadas y las mostradas en la **Tabla 1** que convierten a Kafka en una fuerte herramienta para la adquisición de datos en tiempo real para un ambiente de Big Data.

1.2.2 Procesamiento y análisis de los datos

Para el procesamiento de la información se tuvo en cuenta dos de las tecnologías más usadas para Big Data en la actualidad según se pudo constatar mediante la investigación sobre el tema. Estas herramientas son Apache Spark y Apache Storm. En la **Tabla 2** se hace una comparación entre ambas tecnologías para encontrar razones que nos permitan seleccionar una de ellas para este proyecto.

Tabla 2. Características de las herramientas de procesamiento y análisis[16], [29].

Características	**Apache Spark Streaming**	**Apache Storm**
Lenguajes de programación	Java, Scala, Python y R	Java, C#, Python, Scala, Perl
Confiabilidad	Admite el modo de procesamiento: exactamente una vez	Admite el modo de procesamiento: exactamente una vez y al menos una vez
Modelo de procesamiento	*Micro-Batch*	*Stream, micro-batch*
Tolerancia a fallos	Si	Si
Facilidad de operatividad	Bastante fácil	No tanto
Monitoreo	Si	Si
Latencia	Baja	Muy buena sin muchas restricciones
Escalabilidad	Muy Alta	Alta
Tiempo real	Si	Si
Documentación	Muy extensa	Extensa

La diferencia en la latencia entre Spark Streaming y Storm es un punto a favor de Storm que lo convierte en una plataforma muy adecuada para el procesamiento de datos en tiempo real, pero en general cuentan con funcionalidades muy parecidas.

Apache Spark cobra más interés debido a que es un motor de cómputo de propósito general siendo más apropiado para esta investigación. Esta es la razón por la que la demanda de Apache Spark es mayor que otras herramientas por parte de los profesionales de la Tecnología de la Información. Apache Spark maneja diferentes tipos de problemas, es fácil de utilizar por los desarrolladores y brinda la flexibilidad de trabajar en diferentes idiomas y entornos. Spark no solo cuenta con una amplia comunidad, además, incorpora una serie de

herramientas muy útiles como la librería MLib para implementar soluciones de aprendizaje automático y muy importante es su módulo Spark Streaming pues permite el análisis en tiempo real de flujos de datos mediante *micro-batch*, todo esto convierte a Spark en una solución mucho más eficiente que Storm. La selección de Spark Streaming de Apache Spark es la mejor opción para este proyecto por todas las características dadas anteriormente.

1.2.3 Almacenamiento

Se necesita de un sistema fiable, escalable y con un alto rendimiento. En la actualidad las bases de datos las podemos dividir en Sql y No-Sql. Si nos centramos en un entorno Big Data, la elección de la base de datos será sin dudas de tipo No-Sql, pues resulta menos estricta, fácil de escalar y debido a la naturaleza de los datos en Big Data, que no siempre tienen una estructura, un orden y estos datos almacenados no requieren una estructura fija. Se tuvo en cuenta dos bases de datos No-sql para este proyecto y se realizó una comparación entre ellas a modo de seleccionar la más adecuada. Las bases de datos son: Apache Cassandra y MongoDB. La **Tabla 3** muestra la comparación.

Tabla 3. Características de las bases de datos[27], [29].

Características	**Apache Cassandra**	**MongoDB**
Escalabilidad de escritura	Alta	Baja
Alta disponibilidad	Si	No tanto
Soporte de lenguaje de consulta	Si. CQL	No tiene
Facilidad de uso	Si	Si
Método de replicación	Factor de replicación seleccionable	Replicación *master-slave*
Lenguaje de implementación	Java	C++
Casos de uso	Análisis en tiempo real, IoT, datos en streaming	Análisis en tiempo real, IoT

En el caso de MongoDB con su modelo de " *single master* " puede realizar escrituras solo en el servidor primario. Los servidores secundarios solo pueden usarse para lecturas. Por ejemplo, si tiene un conjunto de réplicas de dos nodos, solo el maestro está tomando escrituras y el otro nodo solo se usa para las lecturas. Esto limita enormemente la escalabilidad de escritura. Cassandra al no seguir el modelo *master-slave* puede realizar escrituras en cualquier servidor. Su escalabilidad de escritura está limitada por la cantidad de servidores que tiene en el clúster, es decir, cuantos más servidores tenga en el clúster mejor se escalará.

Como el modelo de MongoDB es "*single master*", en caso de que el *master* caiga, uno de los *slaves* es elegido como maestro. Este proceso ocurre automáticamente, pero toma tiempo, generalmente de 10 a 40 segundos. Durante este tiempo de elección de nuevo líder, su conjunto de réplicas está inactivo y no puede recibir escrituras. Cassandra soporta un modelo de "*multiple master*" y la pérdida de un solo nodo no afecta la capacidad del clúster para tomar escrituras, por lo que puede lograr un tiempo de actividad del cien por ciento para las escrituras.

Cassandra soporta el lenguaje de consulta CQL que es muy similar a SQL (del inglés *Structured Query Languaje*). MongoDB en este punto no tiene soporte para un lenguaje de consulta. Las consultas están estructuradas como fragmentos JSON.

En cuanto a la facilidad de uso de estas tecnologías MongoDB sería una opción indiscutible muy fácil para que esté en funcionamiento. Sin embargo, Cassandra ha hecho grandes avances en este aspecto del producto con la adopción de CQL como la interfaz principal para Cassandra, esto ha llevado un paso más allá y ha hecho que sea muy sencillo para los programadores SQL usar Cassandra.

En términos generales, MongoDB es el mejor para cargas de trabajo con gran cantidad de datos altamente desestructurados. Si no sabe o tiene una capacidad mínima para anticipar la escala y los tipos de datos con los que se va a trabajar, entonces las estructuras de datos flexibles de MongoDB se adaptan mejor que en Cassandra.

Cassandra es la mejor opción para casos de uso que trabajan con tipos de datos de estilo SQL. También funciona bien si se necesita velocidades de escritura muy altas y ofrece una muy buena flexibilidad y escalabilidad.

Luego de todo lo expuesto sobre estas bases de datos se seleccionó Apache Cassandra como el sistema para el almacenamiento, esto se debe a todas las facilidades y beneficios que posee sobre MongoDB.

1.3 Conclusiones del capítulo

En este capítulo se analizaron varias de las tecnologías punteras en el entorno de Big Data. Se realizó una comparación entre dos de las principales tecnologías en cada etapa correspondiente al flujo de la información en un sistema Big Data, para ello se han expuesto las principales características de las tecnologías con el objetivo de encontrar las más adecuada en la conformación de la propuesta de sistema para el tratamiento de datos en tiempo real. Las tecnologías que más se ajustaron a los parámetros para formar parte de la plataforma son: Apache Kafka, para la recolección de los datos, Apache Spark para el análisis y procesamiento de la información recolectada con su módulo Spark Streaming y Apache Cassandra como la base de datos para el almacenamiento de la información procesada.

CAPÍTULO 2. ARQUITECTURA E IMPLEMENTACIÓN DE LA PLATAFORMA BIG DATA

Este capítulo comienza con la definición de la estructura del sistema para continuar con la instalación y configuración de las tecnologías. El despliegue de la plataforma Big Data se realiza en una Servidor Virtual Privado (VPS, del inglés Virtual Private Server), dicha VPS se crea con todas las características técnicas para soportar cada una de las tecnologías que forman parte del sistema. Con todas las tecnologías en funcionamiento se realiza una aplicación programada en lenguaje Scala sobre IntelliJ IDEA para el tratamiento de datos de una plataforma ThingsBoard de IoT, esto permite poner en funcionamiento la plataforma Big Data.

2.1 Arquitectura propuesta

Este trabajo consiste en la implementación de las tecnologías necesarias para conformar una plataforma Big Data en tiempo real, y requiere de conocimiento y uso de Big Data. Se necesita una fuente de datos que mantenga un flujo en tiempo real hacia el sistema, para ello se utiliza ThingsBoard, una plataforma de IoT con la que cuenta la Universidad Central "Marta Abreu" de Las Villas, dicha plataforma recibe datos de una serie de sensores.

El flujo de la información continúa con una herramienta para la recolecta de los datos que garantice la óptima integridad de los mismos, y esta plataforma va a contar con Apache Kafka, como quedó establecido en el capítulo 1 de este proyecto. Le sigue una herramienta que permita el procesamiento y análisis de la información ingerida por Kafka en *streaming*, donde entra Apache Spark con su módulo Spark Streaming, debido a todo lo expuesto en el capítulo 1 sobre esta tecnología que resulta en la mejor opción para la plataforma Big Data que se propone.

La base de datos Cassandra es la última herramienta seleccionada para el bloque relacionado con el almacenamiento en el flujo de la información, aquí se almacenan los resultados del procesamiento realizado por Spark Streaming.

Un esquema de la arquitectura de la plataforma integrada por cada una de las tecnologías se muestra en la Figura 2-1.

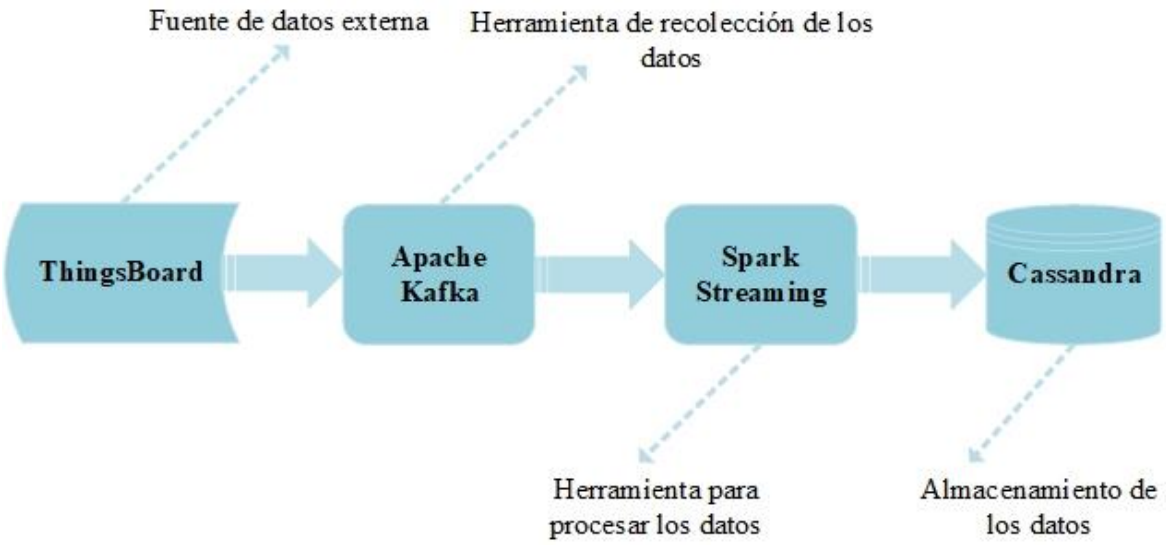


Figura 2-1. Arquitectura de la plataforma Big Data propuesta.

2.1.1 Entorno de trabajo

Para la instalación de las tecnologías y la creación del ambiente de trabajo se utilizó un entorno virtual, existen diferentes tecnologías de virtualización, entre ellas se puede mencionar VMWARE o VirtualBox que son algunas de las más populares, pero en este proyecto se utilizó PROXMOX VE. PROXMOX es una tecnología *open sourse* de virtualización que permite operar con virtualización completa KVM (del inglés *Kernel Virtual Machine*) y contenedores de tipo OpenVZ en un mismo entorno. Dicha tecnología utiliza un entorno basado en Debian con un *kernel* RHEL (del inglés *Red Hat Enterprise Linux*) modificado, de esta manera sobre estos dos pilares se puede manejar los servidores virtuales y los contenedores. Una de las grandes ventajas de éste producto es que las operaciones se pueden realizar desde la línea de comandos o *shell* y vía interfaz web[30].

Luego, desde el PROXMOX se comienza con la virtualización KVM y preparar el entorno de trabajo, se crea una VPS con las propiedades necesarias para soportar el funcionamiento de todas las herramientas que conforman la plataforma Big Data. Se instala sobre esta VPS una imagen del sistema operativo. Como sistema operativo se puede emplear cualquier distribución GNU/Linux[30]. Actualmente existe soporte para Debian, Ubuntu, CentOS, entre otros. De los anteriores se escogió Ubuntu en su versión 18.04, esta es una distribución estable, gratis, posee soporte y una amplia documentación. En cuanto a las especificaciones

técnicas la VPS cuenta con 8 GB de memoria RAM, 4 *sockets*, 2 *cores* por *sockets* y 32 GB de disco duro.

2.1.2 Requerimientos

El primer requisito necesario es tener la VPS con acceso a Internet para las instalaciones que se ejecutan y la comprobación del funcionamiento de la plataforma con el desarrollo de una aplicación que también requiere de Internet.

Las tecnologías que se instalan para conformar el sistema Big Data necesitan una serie de requisitos como Java, Scala, además, las aplicaciones que se ejecutan en un clúster de Apache Spark deben ser construidas y para ello existen dos herramientas disponibles, SBT (del inglés *Simple Build Tool*) y Maven. Se decidió emplear SBT pues está enfocado hacia Scala y es más sencillo de usar[17], SBT es una herramienta utilizada en proyectos de Scala para la automatización de compilaciones, gestión de dependencias, etc.

Esta sección se dedicó a la instalación y configuración de todos estos requisitos necesarios para el despliegue de la plataforma y cada una de las tecnologías que la conforman. La VPS creada cuenta con el sistema operativo Ubuntu 18.04 instalado, se accede a la VPS para comenzar con las instalaciones mediante *ssh* desde la máquina local con la dirección IP de la misma:

```
$ ssh root@10.12.53.15
$ root@10.12.53.15's password: Introducir contraseña.
```

Después de acceder a la VPS se crea un directorio para la instalación y se pasa a la instalación de Java en el sistema.

```
# mkdir opt
```

Instalación de *Java*:

Se descarga la última versión de Java:

```
# jdk-8u202-linux-x64.tar.gz
```

Se mueve hacia la carpeta de instalación:

```
# mv jdk-8u202-linux-x64.tar.gz /opt/
```

Se entra en esta dirección y se descomprime el archivo de instalación:

```
# cd /opt/

# tar xzf jdk-8u202-linux-x64.tar.gz
```

Se entra en la dirección siguiente y se actualiza las alternativas de Java:

```
# cd /opt/jdk1.8.0_202/

# update-alternatives --install /usr/bin/java java /opt/jdk1.8.0_202/bin/java 2

# update-alternatives --config java

# update-alternatives --install /usr/bin/jar jar /opt/jdk1.8.0_202/bin/jar 2

# update-alternatives --install /usr/bin/javac javac /opt/jdk1.8.0_202/bin/javac 2

# update-alternatives --set jar /opt/jdk1.8.0_202/bin/jar

# update-alternatives --set javac /opt/jdk1.8.0_202/bin/javac
```

Se agrega la variable de entorno JAVA_HOME al archivo que contiene una configuración de entorno de todas las variables del sistema mediante el siguiente comando:

```
# nano ~/.bashrc
```

Se agrega lo siguiente:

```
# export JAVA_HOME=/opt/jdk1.8.0_202

# export JRE_HOME=/opt/jdk1.8.0_202/jre

# export PATH=$PATH:/opt/jdk1.8.0_202/bin:/opt/jdk1.8.0_202/jre/bin
```

Actualizar el archivo:

```
# source ~/.bashrc
```

Comprobar la instalación:

```
# java –version
```

Instalación de *Scala*.

Se ingresa al directorio de instalación y mediante el comando *wget* se descarga el archivo de instalación:

```
# cd /opt/

# wget www.scala-lang.org/files/archive/scala-2.12.8.deb
```

Usar el siguiente comando para instalar *Scala*:

```
# dpkg -i scala-2.12.8.deb
```

Comprobar la instalación:

```
# scala -version
```

Instalación de SBT.

Ejecutar el siguiente comando:

```
# echo "deb https://dl.bintray.com/sbt/debian /" | tee -a /etc/apt/sources.list.d/sbt.list # apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 –recv

2EE0EA64E40A89B84B2DF73499E82A75642AC823
```

Actualizar el sistema:

```
# apt update
```

Instalar SBT:

```
# apt install sbt

# apt remove scala-library scala
```

2.2 Instalación, configuración y funcionamiento de Apache Kafka

Apache Kafka es el sistema de adquisición de datos en tiempo real. El protocolo de mensajería Kafka es un protocolo basado en TCP que proporciona un método rápido, escalable y duradero para intercambiar datos entre aplicaciones, este proveedor de mensajería permite separar el proceso de mensajería de la fuente de datos que los genera. Kafka se implementa en Scala y utiliza herramientas de compilación Gradle para construir binarios Kafka. Gradle es una herramienta de automatización de compilación para proyectos Scala, Groovy y Java que requieren instalar Java, este requisito fue resuelto en la sección 2.1.2 por lo que se puede pasar a la instalación de Kafka.

Los pasos a seguir para la correcta instalación de Kafka son los siguientes[31]:

Crear el directorio para descargar las herramientas:

```
# mkdir KafkaSpark
```

Descargar los binarios de Apache Kafka mediante el siguiente comando:

```
# curl "https://www.apache.org/dist/kafka/2.1.1/kafka_2.11-2.1.1.tgz" –o
/KafkaSpark/kafka.tgz
```

Crear un directorio para la instalación con el siguiente comando:

```
# mkdir /KafkaSpark/kafka
```

Ingresar en el directorio de instalación y descomprimir el archivo mediante los siguientes comandos:

```
# cd /KafkaSpark/kafka

# tar -xvzf /KafkaSpark/kafka.tgz --strip 1
```

Se especifica la bandera –strip 1 para garantizar que el contenido del archivo comprimido se extraiga en el directorio ~ /kafka en sí mismo y no cualquier otro.

Se agregan algunos aspectos de la configuración para el correcto funcionamiento.

Es necesario la creación de un archivo de unidad del sistema para el servicio Kafka, esto permite realizar acciones de servicio comunes, como iniciar, detener y reiniciar Kafka de manera coherente con otros servicios de Linux. Se crea un archivo para Zookeeper, que como se explicó en el capítulo 1 de este proyecto, es un servicio que Kafka usa para administrar su estado de clúster y configuraciones.

Para crear los servicios para Zookeeper ejecutar el siguiente comando:

```
# nano /etc/systemd/system/zookeeper.service
```

Luego agregar la configuración siguiente:

```
[Unit]

Requires=network.target remote-fs.target After=network.target remote-fs.target

[Service]

Type=simple

User=kafka

ExecStart=/KafkaSpark/kafka/bin/zookeeper-server-start.sh

/KafkaSpark/kafka/config/zookeeper.properties
```

```
ExecStop=/KafkaSpark/kafka/bin/zookeeper-server-stop.sh
Restart=on-abnormal
[Install]
WantedBy=multi-user.target
```

La sección [Unit] especifica que Zookeeper requiere redes y el sistema de archivos para estar listo antes de que pueda comenzar.

La sección [Service] especifica que *systemd* debe usar los archivos de *shell* zookeeper-server-start.sh y zookeeper-server-stop.sh para iniciar y detener el servicio respectivamente. También especifica que Zookeeper debe reiniciarse automáticamente si sale de forma anormal.

Se crean los servicios para Kafka:

```
# nano /etc/systemd/system/kafka.service
```

Agregar la configuración siguiente:

```
[Unit]
Requires=zookeeper.service
After=zookeeper.service
[Service]
Type=simple
User=root
ExecStart=/bin/sh -c '/KafkaSpark/kafka/bin/kafka-server-start.sh
/KafkaSpark/kafka/config/server.properties > /KafkaSpark/kafka/kafka.log 2>&1'
ExecStop=/KafkaSpark/kafka/bin/kafka-server-stop.sh
Restart=on-abnormal
[Install]
WantedBy=multi-user.target
```

La sección [Unit] especifica que este archivo de unidad depende de zookeeper.service. Esto asegurará que Zookeeper se inicie automáticamente cuando se inicie el servicio Kafka.
La sección [Service] especifica que *systemd* debe usar los archivos de *shell* kafka-server-start.sh y kafka-server-stop.sh para iniciar y detener el servicio respectivamente. También especifica que Kafka debe reiniciarse automáticamente si ocurre una salida anormal del sistema.

Ahora que los archivos de unidad fueron definidos se ejecutan los siguientes comandos:

```
# systemctl daemon-reload

# systemctl start zookeeper

# systemctl enable zookeeper

# systemctl start kafka

# systemctl enable Kafka
```

Como otro paso adicional se instaló KafkaT que es una herramienta que facilita ver los detalles del grupo Kafka y realizar algunas tareas para la administración y manejo de los componentes de Kafka desde la línea de comandos. Los siguientes comandos permiten descargar e instalar KafkaT. Como es un *gem* de Ruby, se necesita que Ruby la use y es necesario el paquete *build-essential* para poder construir las otras *gem* de las que depende. Seguir los pasos siguientes para realizar la instalación:

```
# apt install ruby ruby-dev build-essential
```

Usar el comando *gem* para instalar KafkaT:

```
# gem install kafkat
```

KafkaT usa *.kafkatcfg* como el archivo de configuración para determinar los directorios de instalación y registro de su servidor Kafka. También debe tener una entrada que apunte KafkaT a su instancia de ZooKeeper.

Crear un nuevo archivo de configuración:

```
# nano ~/.kafkatcfg
```

Agregar las siguientes líneas al archivo de configuración para especificar la información requerida sobre el servidor Kafka y la instancia de Zookeeper:

```
{
  "kafka_path": "~/kafka",
  "log_path": "/tmp/kafka-logs",
  "zk_path": "localhost:2181"
}
```

Comprobar el funcionamiento con el siguiente comando que brinda detalles sobre las particiones en Kafka:

```
# kafkat partitions
```

Las propiedades más importantes definidas en *zookeeper.properties* se muestran a continuación:

```
# Directorio de datos donde se almacena la instantánea del zookeeper.
dataDir=/tmp/zookeeper
# El puerto de escucha por solicitud de un cliente.
clientPort=2181
# Deshabilite el límite por ip en el número de conexiones ya que esta es una configuración que no es de producción.
maxClientCnxns=0
```

De manera predeterminada el servidor Zookeeper escuchara por el puerto 2181/TCP [12].

Las propiedades más importantes del archivo *server.properties* que se definen para el bróker de Kafka son las siguientes:

```
# El id del bróker. Esto se establece mediante un entero único para cada bróker.
Broker.id=0

# El puerto en el que el servidor socket escucha.

port=9092
```

```
# El directorio en el cual se van almacenar los archivos de registro.
log.dir=/tmp/kafka8-logs

# El número predeterminado de particiones de registro por tema.
num.partitions=2

# La dirección URL de conexión de Zookeeper.

zookeeper.connect=localhost:2181

# Para poder eliminar un topic:

delete.topic.enable = true
```

Las propiedades más importantes predeterminadas para el productor se definen en *producer.properties*.

```
# Metadata broker list. Lista de brókeres utilizados para el conocimiento de arranque sobre el resto del clúster. Formato: host1:port1,host2:port2...
metadata.broker.list=localhost:9092
# Especifique el códec de compresión para todos los datos generados: none, gzip, snappy.
compression.codec=none
```

Las propiedades predeterminadas para el consumidor se definen en *consumer.properties*. Estas son las más importantes:

```
# ID de grupo de consumidores, no es más que un valor que identifica de manera única a un grupo de consumidores dentro del mismo grupo de consumidores.
group.id=test-consumer-group
```

Kafka proporciona un archivo de configuración de Zookeeper simple y de manera predeterminada. Por tanto, se inicia una instancia local de Zookeeper usando el siguiente comando[12]:

```
# bin/zookeeper-server-start.sh config/zookeeper.properties
```

Para iniciar el bróker de Kafka, es suficiente con abrir una nueva ventana de consola y ejecutar el siguiente comando[12]:

```
# bin/kafka-server-start.sh config/server.properties
```

Se crea un tema de llamado “my-topic” con una sola partición. Kafka brinda una utilidad de líneas de comandos para crear temas. En una nueva venta de consola se ejecuta el siguiente comando[12]:

```
# bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 1 –topic my-topic
```

Se debe obtener la salida siguiente:

```
Created topic "my-topic"
```

La línea de comando ejecutada kafka-topics.sh crea un tema, el valor predeterminado de particiones de dos se anula a una y se observa un mensaje de creación satisfactorio, toma información sobre el servidor Zookeeper mediante: localhost: 2181.

Para obtener una lista de temas en cualquier servidor Kafka se puede utilizar el siguiente comando[12]:

```
# bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Se puede iniciar un cliente productor mediante líneas de comandos en Kafka y permite publicar entradas como mensajes en el clúster Kafka. Luego de iniciado el productor por consola de manera predeterminada cada línea de comando es considerada como un nuevo mensaje. La siguiente línea de comandos se debe correr en una nueva ventana de consola para iniciar un productor y enviar mensajes[12]:

```
# bin/kafka-console-producer.sh --broker-list 10.12.53.15:9092 --topic my-topic
```

El productor de líneas de comandos requiere el parámetro *broker-list* que especifica los brókeres que se conectarán como (dirección del nodo: puerto), 10.12.53.15: 9092. El tema “*my-topic*” que fue creado anteriormente es requerido para enviar los mensajes a un grupo de consumidores que se suscriban a dicho tema.

De igual manera Kafka proporciona un cliente consumidor de líneas de comandos para el consumo de los mensajes publicados. El comando a continuación se usa para iniciar un consumidor, que recibe los mensajes tan pronto se suscriba al tema donde se publicaron por el productor. Se utiliza el comando *from-beginning* para leer desde el primer mensaje publicado [12]:

```
# /bin/kafka-console-consumer.sh --bootstrap-server 10.12.53.15:9092 --topic my-topic --from-beginning
```

2.3 Instalación e inicialización de Apache Spark

La etapa de la plataforma para el análisis y procesamiento de datos proviene de Kafka cuenta con Spark Streaming de Apache Spark. Spark es un sistema distribuido a través de un clúster, donde es coordinado a través del objeto *SparkContext*, pero en el caso de aplicaciones basadas en *streams*, como es el caso de su módulo Spark Streaming se utiliza *StreamingContext* en lugar de *SparkContext*, debido a que *StreamingContext* soporta *DStreams*. Los métodos *start()* y *awaittermination()* del contexto de *streaming* permiten arrancar y terminar el proceso cuando la fuente de datos se cierre. El proceso de instalación de Apache Spark resulta muy fácil, no obstante, es necesario tener instalado Java, Scala, y SBT o Maven para la construcción de la aplicación de consumidor en Spark Streaming [32]. En este proyecto se instala SBT, como también fue instalado Java y Scala en la sección 2.1.2.

Los pasos a seguir para la instalación de Spark son los siguientes:

Descargar los binarios de Spark en el directorio de descargas usando el comando wget[17]:

```
# cd KafkaSpark/

# wget https://www-us.apache.org/dist/spark/spark-2.4.3/spark-2.4.3-bin-hadoop2.7.tgz
```

Descomprimir el binario de Spark:

```
# tar -xzf spark-2.4.3-bin-hadoop2.7.tgz
```

Mover los archivos descomprimidos al directorio spark:

```
# mv spark-2.4.3-bin-hadoop2.7 spark
```

Agregar la variable de entorno SPARK_HOME al archivo que contiene una configuración de entorno de todas las variables del sistema:

```
# cd

# nano ~/.bashrc
```

Agregar las siguientes líneas:

```
# export SPARK_HOME=/root/spark

# export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Actualizar el archivo con las variables del sistema:

```
# source ~/.bashrc
```

Se puede iniciar Spark manualmente en el modo Spark Standalone como el administrador simple para el clúster de Spark, pues solo se tiene instalado Spark y su clúster autónomo proporciona una manera fácil de comenzar. Para ello se entra al directorio de Spark e iniciar un servidor *master* independiente ejecutando los comandos siguientes[17]**:**

```
# cd KafkaSpark/spark

# ./sbin/start-master.sh
```

Similarmente, se puede comenzar uno o más *workers* y se conectan al *master* proporcionándoles su URL como en el comando siguiente[17]:

```
# ./sbin/start-slave.sh spark://streaming:7077
```

Se puede encontrar la *web UI* del *master* en la dirección http://10.12.53.15:8080. Una vez iniciado el *worker*, en la *web UI* del *master* aparece un nuevo nodo adicionado con un número de CPUs y memoria, por defecto los puertos para la *web UI* son 8080 y 8081 para el *master* y el *worker* respectivamente. El *master* por defecto se inicializa por el puerto 7077[17].

Se puede ejecutar un *Spark shell* interactivo sobre el clúster ejecutando el siguiente comando:

```
# ./bin/spark-shell --master spark://streaming:7077
```

2.4 Instalación y funcionamiento de Cassandra

La forma más fácil de instalar Apache Cassandra en Ubuntu es instalando el paquete *deb* desde el repositorio oficial de Apache Cassandra. Cassandra requiere la instalación de Java[27], pero este es un punto realizado en la sección 2.1.2.

Los pasos que se desarrollaron para la instalación de Cassandra son los siguientes[27]:

Con el siguiente comando se agrega el repositorio Apache de Cassandra a /etc/apt/sources.list.d/cassandra.sources.list:

```
# echo "deb http://www.apache.org/dist/cassandra/debian 39x main" | tee -a /etc/apt/sources.list.d/cassandra.sources.list
```

Añadir las claves del repositorio de apache Cassandra:

```
# curl https://www.apache.org/dist/cassandra/KEYS | sudo apt-key add -
```

Actualizar el repositorio:

```
# apt update
```

Instalar Cassandra:

```
# apt install cassandra
```

Habilitar el servicio de Cassandra:

```
# systemctl enable cassandra
```

Iniciar Cassandra:

```
# systemctl start cassandra
# systemctl status cassandra
```

Verificar que Cassandra está ejecutándose con el siguiente comando:

```
# nodetool status
```

Para interactuar con Cassandra a través CQL, se puede usar una utilidad de línea de comandos llamada *cqlsh* que se incluye en el paquete de Cassandra. Ejecutar el comando:

```
# cqlsh
```

Se obtiene la siguiente salida donde se puede empezar a interactuar con Cassandra:

```
Connected to Test Cluster at localhost:9042.
[cqlsh 5.0.1 | Cassandra 3.8 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Un KeySpace en una base de datos NoSQL como Cassandra no tiene ninguna estructura concreta y existe un KeySpace por aplicación. Para comenzar a interactuar con Cassandra se crea un KeySpace de nombre "*data*" mediante el comando siguiente[33]:

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

Aquí se crea un KeySpace con Estrategia Simple que se utiliza cuando se cuenta con un solo centro de datos. En esta estrategia la primera replica se coloca en el nodo seleccionado por el particionador. Los nodos restantes se colocan en el sentido de las agujas del reloj en el anillo sin tener en cuenta la ubicación del nodo. El factor de replicación en 1 es la cantidad de réplicas de los datos en los diferentes nodos.

Se sigue con la creación de una tabla llamada "*value*" con las variables de los datos que se reciben de la plataforma de IoT. Se ejecuta el siguiente comando[33]:

```
# CREATE TABLE value (temperatura decimal PRIMARY KEY, Volt decimal, humedad decimal, nivel_agua decimal, catidad_personas decimal);
```

Para insertar unos valores de prueba manualmente se ejecuta el comando siguiente [27]:

```
# INSERT INTO value(temperatura, Volt, catidad_personas, humedad, nivel_agua) VALUES (26.5,  90.4, 79.9, 50.7, 111);
```

Para comprobar lo realizado se ejecutan los siguientes comandos para entrar en el KeySpace "data" y ver la tabla "value":

```
# USE data;

# SELECT * from value;
```

Se obtiene la siguiente salida:

```
temperatura | catidad_personas | humedad | nivel_agua | volt

-------------+----------------------+-----------+--------------+------

       26.5 |             90.4 |              79.9 |          50.7 |       111

(1 rows)
```

Para eliminar los datos de la tabla es suficiente con utilizar el comando TRUNCATE. Se entra al KeySpace y se ejecuta sobre la tabla "*value*":

```
# USE data;

# TRUNCATE value;
```

Se obtiene la siguiente salida:

```
temperatura | catidad_personas | humedad | nivel_agua | volt

-------------+----------------------+-----------+--------------+------

 (0 rows)
```

2.5 Integración de las Tecnologías

Para lograr que los datos fluyan por todos los bloques funcionales de la plataforma Big Data es necesario la configuración de algunos parámetros que permitan la comunicación entre todas las tecnologías involucradas. La configuración de estos parámetros permite recibir los datos de la plataforma IoT en la plataforma Big Data, además, deben ser configurados en la misma arquitectura Big Data algunos detalles de configuración para permitir el flujo de los datos desde el sistema de entrada Apache Kafka hasta el sistema final de almacenamiento que es la base de datos Cassandra.

2.5.1 Integración de ThingsBoard y Kafka

Al inicio de este capítulo se comentó sobre la utilización de ThingsBoard como la fuente de datos. ThingsBoard es una plataforma IoT de código abierto para la recopilación de datos. En este proyecto se usa como un recurso con que cuenta la universidad, solo hay que configurar algunos aspectos para poder enviar los datos de telemetría desde ThingsBoard hasta el bróker de Kafka.

ThingsBoard cuenta con un complemento para Kafka que, mediante la correcta configuración, este es responsable de enviar mensajes al o a los brókeres de Kafka activados por regla específicas. En los parámetros de la configuración se puede especificar: (1) el parámetro *bootstrap servers* que indica la lista de brókeres de Kafka; (2) el número de intentos para reconectarse a Kafka si falla la conexión; (3) el número de mensajes por unidad en lote en cliente; (4) el tiempo para almacenar en búfer localmente antes de enviarlo al bróker de Kafka (en ms); (5) el máximo tamaño del *buffer* en el cliente; (6) el número mínimo de réplicas que debe reconocer una escritura; (7) el tema clave serializador por defecto - *org.apache.kafka.common.serialization.StringSerializer*; (8) el valor predeterminado del tema serializador - *org.apache.kafka.common.serialization.StringSerializer* y (9) cualquier otra propiedad adicional podría ser proporcionada para la conexión del bróker de Kafka [34].

La configuración de la extensión de Kafka necesita que el bróker de Kafka esté en funcionamiento. Tener un apropiado tema Kafka creado y ThingsBoard tiene que estar en marcha. En este capítulo fue creado un tema llamado “my-topic”, fue iniciado el servicio de Kafka y ThingsBoard se encuentra en funcionamiento por lo que solo queda configurar el complemento. En el **Anexo I** se encuentran los pasos que se siguieron para configurar esta

extensión de Kafka y poder enviar un mensaje al tema de Kafka cada vez que llegue un nuevo mensaje de telemetría desde un dispositivo a ThingsBoard.

El complemento de ThingsBoard para enviar los datos a Kafka está listo, solo queda una última configuración para que el bróker de Kafka externo reciba estos datos de telemetría de los sensores.

Es necesario configurar otro parámetro que se encuentra dentro del archivo de las configuraciones de Apache Kafka.

```
nano /KafkaSpark/kafka/config/server.properties
```

Dentro de la configuración se añade la siguiente línea, que permite que Kafka escuche por esta dirección y puerto, con el protocolo no SSL (del inglés *Secure Socket Layer*) predeterminado, *PLAINTEXT*. Esta dirección y puerto coincide con la configuración en el complemento de Kafka en ThingsBoard.

```
listeners=PLAINTEXT://10.12.53.15:9092
```

2.5.2 Integración de Kafka y Spark Streaming

Existen algunos aspectos a tener en cuenta para lograr la correcta integración entre ambas tecnologías. El proyecto Kafka introduce una nueva API de consumidor entre sus versiones 0.8 y 0.10, por tanto, existen dos paquetes de Spark Streaming disponibles. Es necesario seleccionar correctamente el paquete según la versión del bróker de Kafka que se esté utilizando pues la versión 0.8 es compatible con la 0.9 y la 0.10, pero la 0.10 no es compatible con las versiones anteriores de Kafka [35]. La figura 2-2 muestra las principales características y cambios de los paquetes spark-streaming-kafka en sus dos versiones.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Maturity	Deprecated	Stable
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes

Figura 2-2. Paquetes disponibles para la integración entre Kafka y Spark Streaming[32].

Se decidió utilizar el paquete en su versión 0.10.0 o mayor pues se cuenta con las versiones más recientes de ambas tecnologías y esta nueva versión presenta características superiores al anterior, además que ya no se desarrolla soporte para la versión más antigua. Esta versión cuenta con soporte en Java y Scala [32].

Para configurar Spark Streaming para recibir datos desde Kafka existen dos enfoques como se explicó en el capítulo 1 de este proyecto. Un primer enfoque utilizando receptores para recibir datos, y que dicho receptor es implementado utilizando la API de consumidor de alto nivel de Kafka, y un nuevo segundo enfoque introducido en Spark 1.3 que no utiliza receptores y se conoce como enfoque directo. Ambos enfoques tienen modelos de programación y características de rendimiento diferentes, y son considerados API estables a partir de la última versión de Spark. El paquete más reciente para la integración utiliza el enfoque directo y es similar al enfoque directo para el *stream* de datos diseñado en la versión 0.8. Este nuevo enfoque brinda garantías más sólidas de principio a fin, y provee un simple paralelismo entre las particiones de Kafka y las particiones de Spark [24].

Lo que hace es consultar periódicamente a Kafka para obtener las últimas compensaciones de cada tema y partición definiendo los rangos de desplazamiento para procesar cada lote. Cuando se inician los trabajos utiliza una nueva API de consumidor de Kafka en lugar de la simple API de consumidor para leer los rangos definidos de compensaciones de Kafka, y estas tienen notables diferencias en su uso, esta versión de la integración está marcada como experimental por lo que está sujeta a cambios[35]. Es importante aclarar que cuando se obtiene un *stream* en Spark proveniente de Kafka se va a tener un registro de un consumidor Kafka en un par clave-valor. La parte correspondiente al valor contiene el dato de tiempo en formato JSON.

2.5.3 Integración de Spark y Cassandra

Para utilizar Cassandra en una aplicación de Apache Spark es necesario añadir las dependencias al archivo build.sbt del proyecto. En la aplicación, es necesario importar el conector de Apache Spark con Cassandra. Además, hay que decirle a Spark como conectarse al clúster de Cassandra, mediante la dirección IP de cualquiera de los nodos del clúster de Cassandra.

Los datos en Cassandra se pueden guardar con el siguiente método:

```
datos.saveToCassandra(“KEY_SPACE”, “NOMBRE TABLA”)
```

Después de las especificaciones anteriores que están encaminadas al desarrollo de la aplicación en la sección 2.8 en lo correspondiente a la conexión entre Spark y Cassandra, queda una configuración adicional en el archivo de configuración de Cassandra que permite escuchar desde Spark para recibir los datos y guardarlos en la tabla creada. Para ello se especifica la propiedad *rcp_address* y *broadcast_rcp_address*, asignándoles 0.0.0.0 y 10.12.53.255 respectivamente, la dirección 0.0.0.0 permite escuchar desde cualquiera. Se siguen los pasos siguientes para la configuración:

```
nano /etc/cassandra/cassandra.yaml
```

Agregar lo siguiente a la configuración para escuchar desde cualquier dirección:

```
rpc_address: 0.0.0.0

broadcast_rpc_address: 10.12.53.255
```

2.6 Desarrollo de la aplicación

Para la programación de la aplicación existen tres lenguajes que se pueden emplear para crear aplicaciones que se ejecuten en Apache Spark, estos son: (1) Java; (2) Scala y (3) Python. Scala es el lenguaje en que está programado Apache Spark, lo cual brida ventajas si se programa en este lenguaje. En el caso de Python habría que realizar trabajo extra para convertir tanto el código como los datos a otro entorno, y algunas cosas pueden no funcionar como se espera. Scala aporta diversas ventajas para la creación de la aplicación por lo que es el lenguaje ideal para este proyecto.

Para el desarrollo de la aplicación se utilizó IntelliJ IDEA, este es un Entorno de Desarrollo Integrado (IDE del inglés *Integrated Development Environment*) para el desarrollo de programas informáticos. Está disponible en dos ediciones: edición para la comunidad y edición comercial. En el caso de uso se utiliza la edición para la comunidad. Además, permite la programación en el lenguaje Scala mediante un complemento de Scala que se le añade sin problemas[36].

Para comenzar a desarrollar la aplicación se entra en IntelliJ IDEA y se crear un nuevo proyecto Scala de tipo SBT.

Se configuran los parámetros del proyecto con la versión 2.11.8 de Scala. Después se continúa con la configuración de SBT con las correctas versiones de las tecnologías y los paquetes necesarios para el proyecto. El **Anexo II** contiene la configuración del archivo *built.sbt*.

Se creó un archivo llamado *kafka.properties* con las propiedades de Kafka que es utilizado en la aplicación. Este archivo se colocó en la carpeta del proyecto SBT.

Con la modificación realizada solo queda guardar los cambios y aceptar las auto-importaciones para que SBT descargue todas las dependencias agregadas al fichero *built.sbt***.**

Se continúa con la definición de la clase principal (*Main*) de la aplicación. Para ello dentro de la carpeta *src/main/scala* se crea el fichero que contendrá la clase principal (clic en botón derecho y crear *scala class* > *object*). Aquí está contenido el código desarrollado para el recibir los datos desde Kafka, procesarlo en Spark Streaming y almacenarlo en la tabla creada en Cassandra. Como se comentó en secciones anteriores los datos recibidos por Kafka desde la plataforma ThingsBoard están en formato JSON, por tanto, la programación de la aplicación comienza con las importaciones de todas las clases necesarias, entre ellas la clase JSON estándar de Scala y las librerías necesarias para el parseo de datos en formato JSON. Las importaciones básicas de Spark para utilizar el módulo Spark Streaming y la creación de un *SparkSession*. Además, el conector de Cassandra y Spark, y las clases para la integración entre Kafka y Spark Streaming.

Después de las importaciones de las clases se comienza con la creación del objeto *Main* que contiene todo el desarrollo de la aplicación. Aquí se crea un *SparkSession* utilizando el método *SparkSession.builder* como punto de entrada a Spark SQL donde se le pasa la dirección del *host* (10.12.53.15) y puerto (9042) por donde escucha Cassandra.

Se crea un objeto local *StreamingContext* con *5* segundos de intervalo de *batch* y un temporizador que se usa para detener el procesamiento después de 60 segundos para poder imprimir los resultados.

Se lee del archivo *kafka.properties* creado que se encuentra en la carpeta del proyecto SBT para obtener las propiedades de Kafka y pasárselas a la variable *kafkaParams*. Se crea la variable *topicsSet* en la que se especifica el tema “my-topic” que contiene los datos recibidos desde ThingsBoard. Después se inicializa el *stream* Kafka en Spark en la variable *messages*

pasándole el valor del *StreamingContext*, el tema de Kafka y los valores de la configuración de Kafka a la función *createDirectStream*. Además, se especifica la *LocationStrategy* y las *Consumer Strategies*.

En cuanto a la *LocationStrategy* la nueva API para el consumidor de Kafka precargará los mensajes en *buffers*. Por lo tanto, es importante por razones de rendimiento que la integración de Spark mantenga a los consumidores almacenados en caché en los ejecutores en lugar de recrearlos para cada lote. En la aplicación se usa LocationStrategies.PreferConsistent y esto distribuirá particiones uniformemente entre los ejecutores disponibles.

La nueva API para el consumidor de Kafka tiene varias formas diferentes de especificar temas. *ConsumerStrategies* proporciona una abstracción que permite a Spark obtener consumidores configurados correctamente incluso después de reiniciar desde el punto de control. En la aplicación se utilizó *ConsumerStrategies.Subscribe*, lo que permite suscribirse a una colección fija de temas.

Se continúa con la creación de la lógica de procesamiento que no se inicia hasta que el *streaming context* comienza, y se ejecuta una vez para cada intervalo de lote. En la variable *json_string* se obtiene el valor solamente del mensaje en un *string* JSON, luego se parsea para obtener el dato limpio y se muestran los resultados. Siguiendo la lógica del flujo el próximo paso es guardar cada RDD en la tabla de Cassandra, para ello se utilizó el método *saveToCassandra("data","value")* donde se le pasa el nombre del KeySpace y de la tabla respectivamente. Para finalizar la aplicación se comienza el *streaming context* mediante *ssc.start()* y se utiliza *ssc.awaitTermination()* para bloquea el terminal mientras se ejecuta el contexto hasta que el temporizador lo detiene. El código completo de la aplicación se puede encontrar en el **Anexo III**.

Para ejecutar la aplicación en IntelliJ IDEA es suficiente con dar clic en el botón de arranque (*run*).

Para crear el .JAR se debe ir a la estructura del proyecto *File > Project Structure* y una vez allí crear un *Artefacto JAR* en el que se incluyan las dependencias. Para construir el JAR basta con construir el artefacto (*Built > Built Artifacts*).

El .JAR generado se va a encontrar en la carpeta del proyecto en la dirección:

```
/Cassandra/out/artifacts/Cassandra_jar/Cassandra.jar
```

Hasta los pasos seguidos anteriormente la aplicación se ejecuta en el IntelliJ IDEA desde la máquina local. Cuando se ejecuta la aplicación los datos son leídos y procesados por Spark Streaming desde el tema "my-topic" en el bróker de Kafka y enviados al KeySpace "data" y la tabla "value" en Cassandra para ser almacenados, y todo esto ocurre en la VPS donde se encuentran instaladas las tecnologías para la plataforma Big Data.

Para ejecutar la aplicación desde la misma VPS se tuvo en cuenta los siguientes pasos.

El proyecto creado con IntelliJ IDEA en la máquina local se encuentra en la dirección:

```
/home/alian/IdeaProyects/cassandra/
```

Todos los archivos del proyecto están contenidos en la carpeta "cassandra" y para ejecutar la aplicación es necesario mover este directorio a la VPS, para ello se utilizó FileZilla que es un cliente FTP multiplataforma de código abierto y software libre, este permitió mover el directorio desde la máquina local hasta la virtual.

Con el directorio en la VPS y acceso a Internet, se entra a este directorio "cassandra" y se ejecuta el comando *sbt run*. Como se recordará SBT fue instalado en el sistema al inicio de este capítulo. De esta manera se inicia la aplicación desde la VPS:

```
cd KafkaSpark/cassandra

sbt run
```

CAPÍTULO 3. COMPROBACIÓN DEL DESEMPEÑO DE LA PLATAFORMA BIG DATA

En este capítulo se proporciona la arquitectura Big Data desarrollada y en funcionamiento. El enfoque del capítulo es cubrir y mostrar mediante el método ilustrativo los resultados de todos los pasos seguidos para llegar al funcionamiento de la plataforma Big Data propuesta.

3.1 Despliegue de las Tecnologías

En las Figuras 3-1, 3-2 y 3-3 se aprecia la VPS creada mediante el gestor de contendores PROXMOX. Es posible ver el estado de la VPS en la Figura 3-1, y las propiedades en la Figura 3-2 y 3-3. Se le brindo a la VPS propiedades que permitieron la instalación y puesta en marcha de las tecnologías y el sistema Big Data como una plataforma unida y funcional. Se observa que la VPS cuenta con 8 GB de memoria RAM, 32 de disco duro y 4 *sockets* con dos núcleos por *socket*.

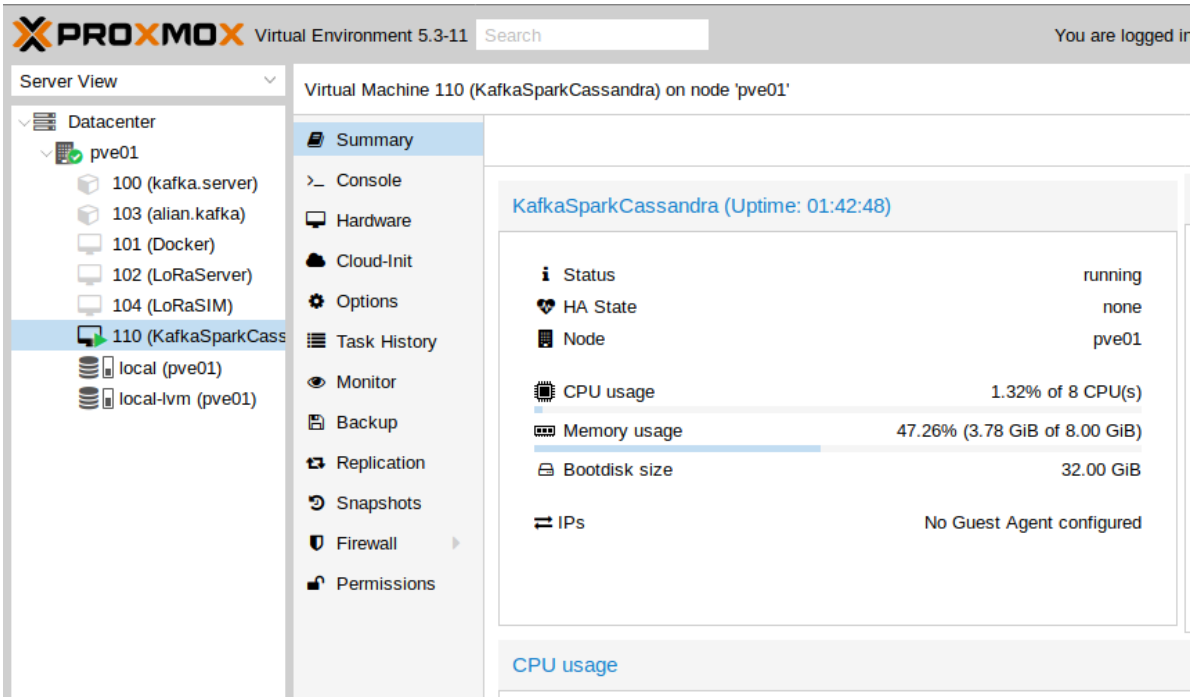


Figura 3-1. VPS creada desde el PROXMOX.

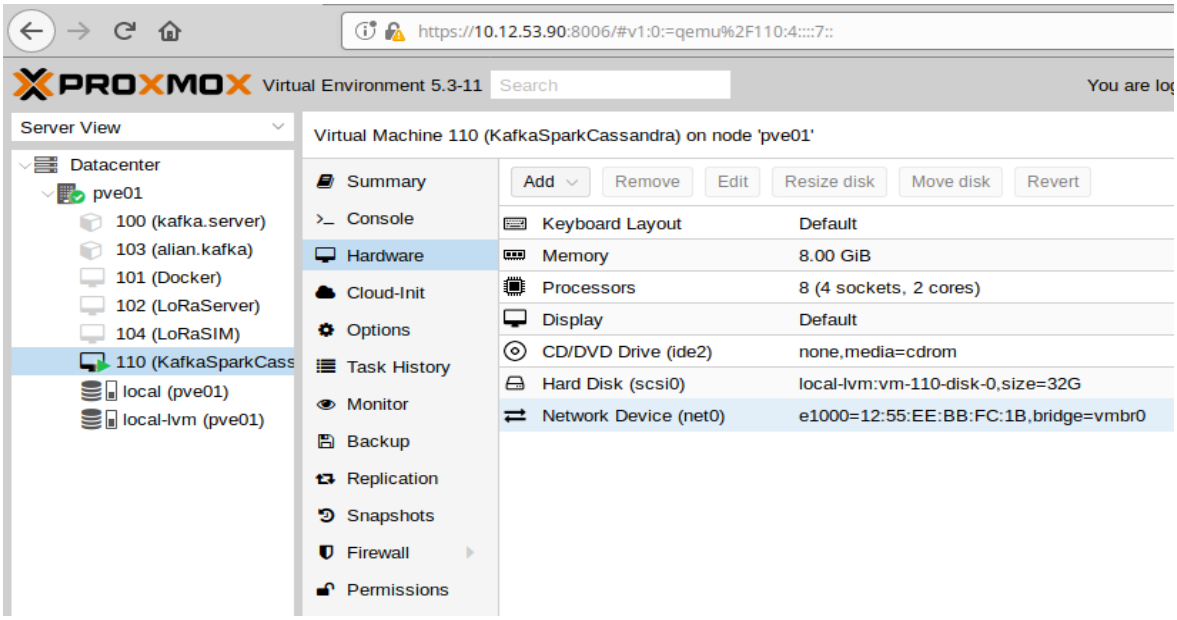


Figura 3-2. Propiedades de la VPS.

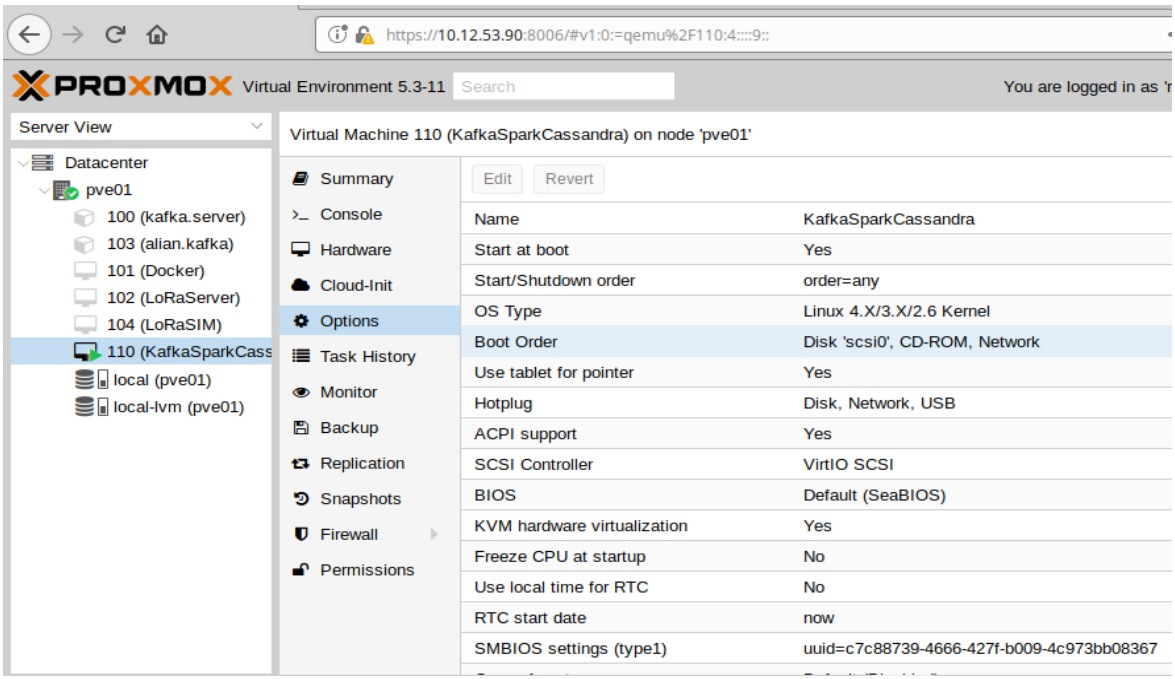


Figura 3-3. Propiedades de la VPS.

Con la VPS en funcionamiento y todas las tecnologías instaladas como se implicó en el capítulo 2 de este proyecto se accede mediante *ssh* desde la máquina local. Proporcionándole la dirección IP (10.12.53.15) de la VPS y la contraseña como se observa en la Figura 3-4.

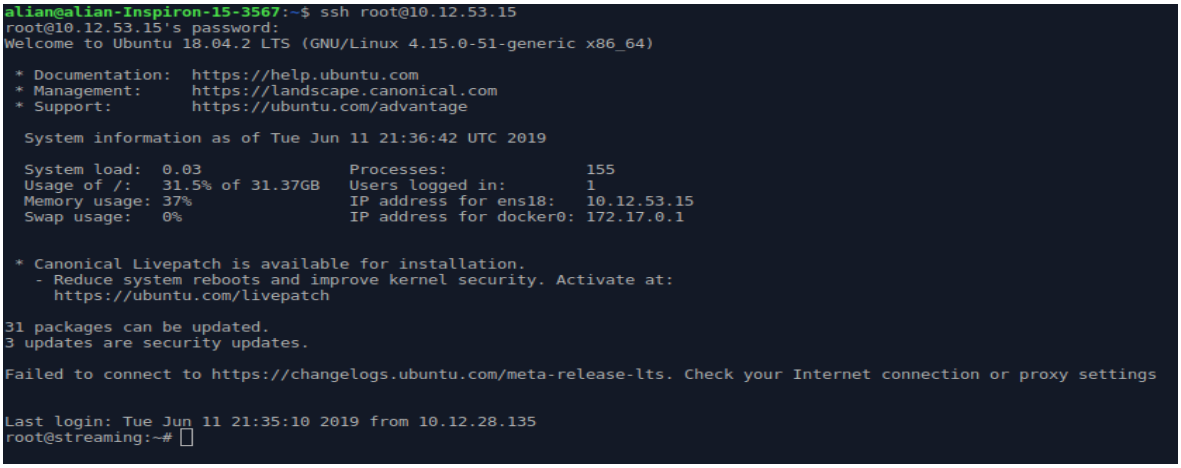


Figura 3-4. Acceso al directorio de la VPS desde el terminal mediante *ssh*.

Dentro de la VPS desde la ventana de consola se entra en el directorio de instalación de Apache Kafka y se inicia el servidor Zookeeper y Kafka mediante los comandos explicados en el capítulo 2. La Figura 3-5 y 3-6 muestra el servidor Zookeeper y Kafka respectivamente iniciados.

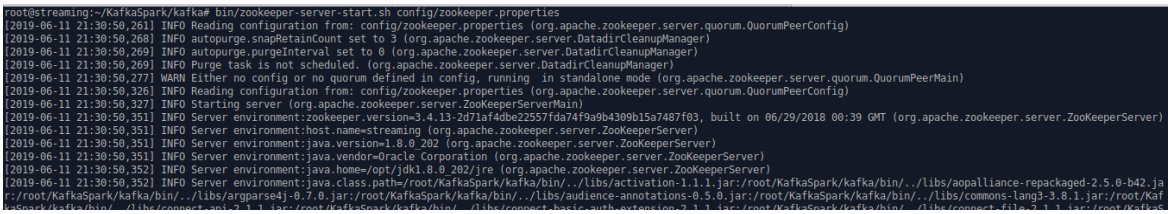


Figura 3-5. Ejecución del servidor Zookeeper

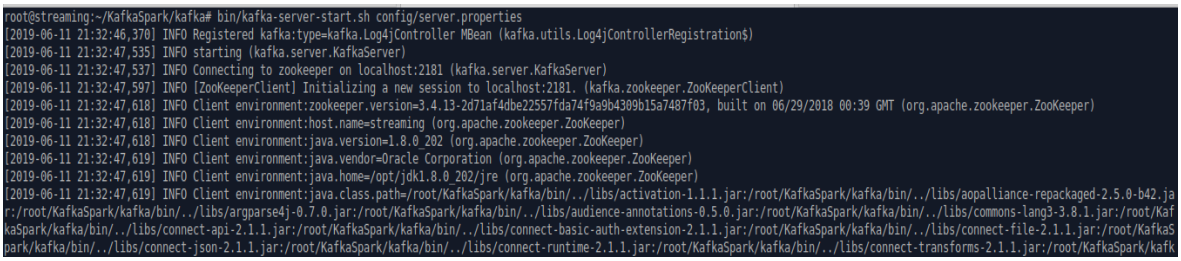


Figura 3-6. Ejecución del servidor Kafka

Mediante el comando *list* se puede ver el tema “my-topic” creado en la Figura 3-7.

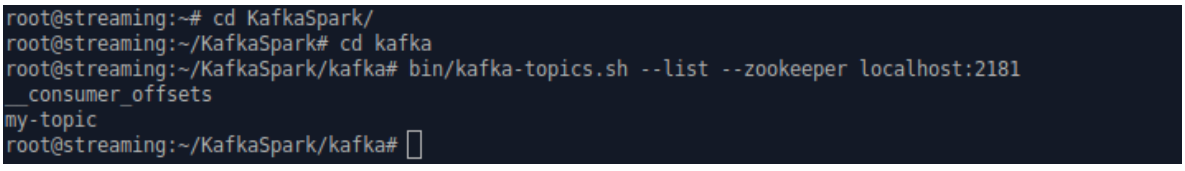


Figura 3-7. Lista de temas en Kafka.

Después se continúa con la inicialización del *master* y *worker* de Spark con ayuda de las líneas de comandos. Se entra en una nueva ventana de consola y se ejecutan los comandos como muestra la Figura 3-8.

```
root@streaming:~# cd KafkaSpark/
root@streaming:~/KafkaSpark# ls
cassandra  cassandra.jar  kafka  kafka.tgz  scala-2.12.8.deb  spark  spark-2.4.3-bin-hadoop2.7.tgz
root@streaming:~/KafkaSpark# cd spark
root@streaming:~/KafkaSpark/spark# ls
LICENSE  NOTICE  R  README.md  RELEASE  bin  conf  data  examples  jars  kubernetes  licenses  logs  python  sbin  yarn
root@streaming:~/KafkaSpark/spark#  ./sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /root/KafkaSpark/spark/logs/spark-root-org.apache.spark.deploy.master.Master-1-streaming.out
root@streaming:~/KafkaSpark/spark# ./sbin/start-slave.sh spark://streaming:7077
starting org.apache.spark.deploy.worker.Worker, logging to /root/KafkaSpark/spark/logs/spark-root-org.apache.spark.deploy.worker.Worker-1-streaming.out
root@streaming:~/KafkaSpark/spark#
```

Figura 3-8. Inicialización del *master* y *worker* de Spark.

En la Figura 3-9 se apreciar la *web UI* del *master* en la dirección http://10.12.53.15:8080 y el nuevo nodo *worker* iniciado con 8 CPUs y memoria de 6.8 GB.

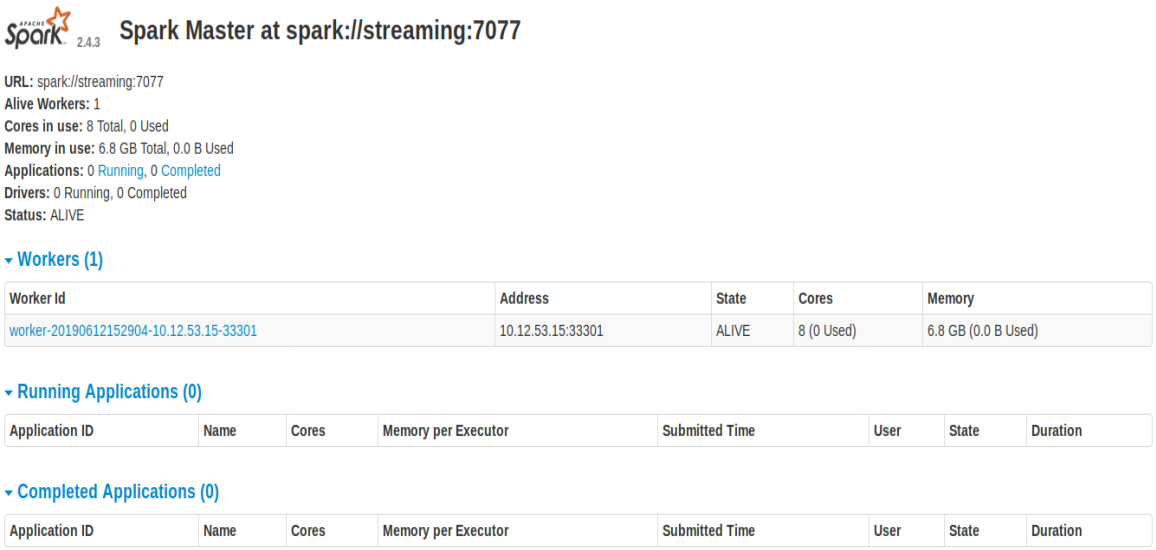


Figura 3-9. Web UI del *master*.

La Figura 3-10 muestra el *spark-shell* interactivo de Scala donde se le pasa la dirección del *master* "spark://streaming:7077" para iniciarlo.

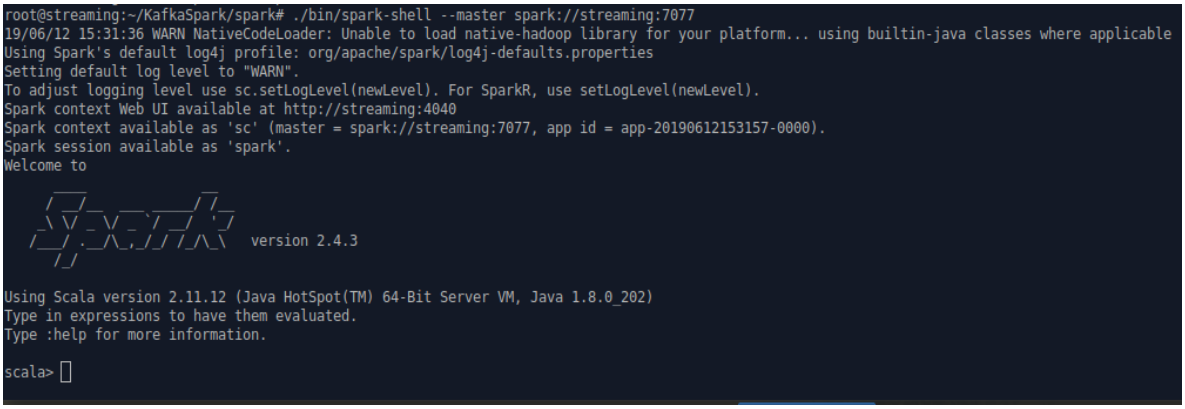


Figura 3-10. *Spark-shell* interactivo de Scala.

En la Figura 3-11 se observa que el servicio de Cassandra está activo, es decir, está en funcionamiento y lista para recibir datos, mediante el comando: service cassandra status.

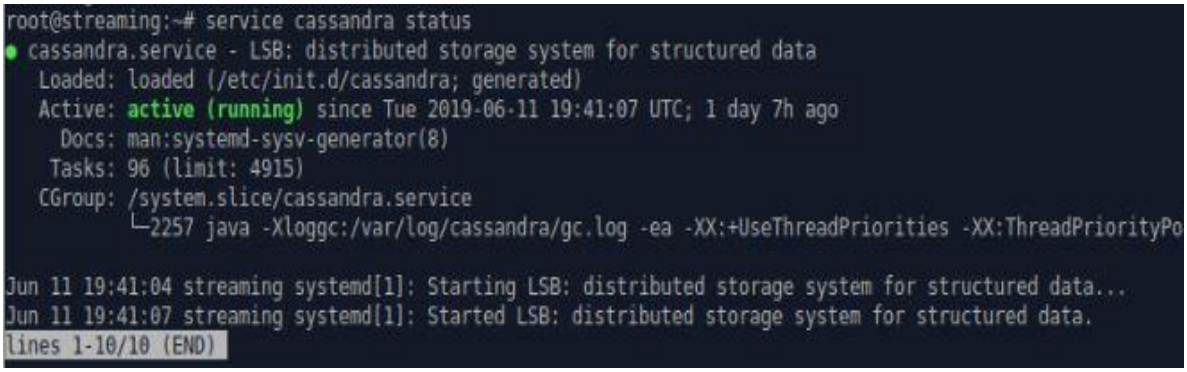


Figura 3-11. Servicio de Cassandra.

3.2 Funcionamiento de la plataforma Big Data

En la Figura 3-12 se observa cómo mediante la ejecución de la línea de comando para el cliente consumidor de Kafka, este recibe los datos de telemetría desde la plataforma ThingsBoard. Los datos están en formato JSON y tienen la estructura siguiente: {“temperatura”: valor de temperatura, “Volt”: valor de Volt, “humedad”: valor de humedad, “nivel_agua”: valor nivel_agua, “cantidad_personas”: valor de cantidad_personas}.

```
root@streaming:~# cd KafkaSpark/
root@streaming:~/KafkaSpark# cd kafka
root@streaming:~/KafkaSpark/kafka# bin/kafka-console-consumer.sh --bootstrap-server 10.12.53.15:9092 --topic my-topic
{"temperatura":26.6,"Volt":117.9,"humedad":78,"nivel_agua":50.1,"catidad_personas":98.6}
{"temperatura":25.3,"Volt":113.9,"humedad":75.5,"nivel_agua":52.9,"catidad_personas":97.2}
{"temperatura":27,"Volt":113.2,"humedad":73.4,"nivel_agua":51,"catidad_personas":100}
{"temperatura":26.8,"Volt":111.5,"humedad":70.6,"nivel_agua":56.1,"catidad_personas":99.2}
{"temperatura":25.8,"Volt":113.1,"humedad":70.6,"nivel_agua":54.1,"catidad_personas":92.6}
{"temperatura":28.1,"Volt":112.3,"humedad":73.1,"nivel_agua":56,"catidad_personas":92.7}
{"temperatura":29.1,"Volt":115.5,"humedad":75.5,"nivel_agua":53.2,"catidad_personas":93.1}
{"temperatura":27.6,"Volt":118.8,"humedad":75.1,"nivel_agua":57.3,"catidad_personas":94.2}
{"temperatura":29.6,"Volt":118.9,"humedad":70.9,"nivel_agua":57.6,"catidad_personas":96.8}
```

Figura 3-12. Cliente consumidor por consola recibiendo los datos desde ThingsBoard.

El desarrollo de la aplicación para comprobar el funcionamiento en tiempo real de la plataforma se realizó en IntelliJ IDEA como fue explicado en el capítulo anterior. La aplicación lee cada dato publicado en Kafka con 5 segundos de *batch*, los procesa, imprime los resultados y los guarda en la tabla "*value*" de Cassandra. La Figura 3-13 muestra la aplicación funcionando mientras se ejecuta desde IntelliJ IDEA, no ocurren errores en la compilación del programa y se aprecia que los datos están parseados y todo funciona correctamente.

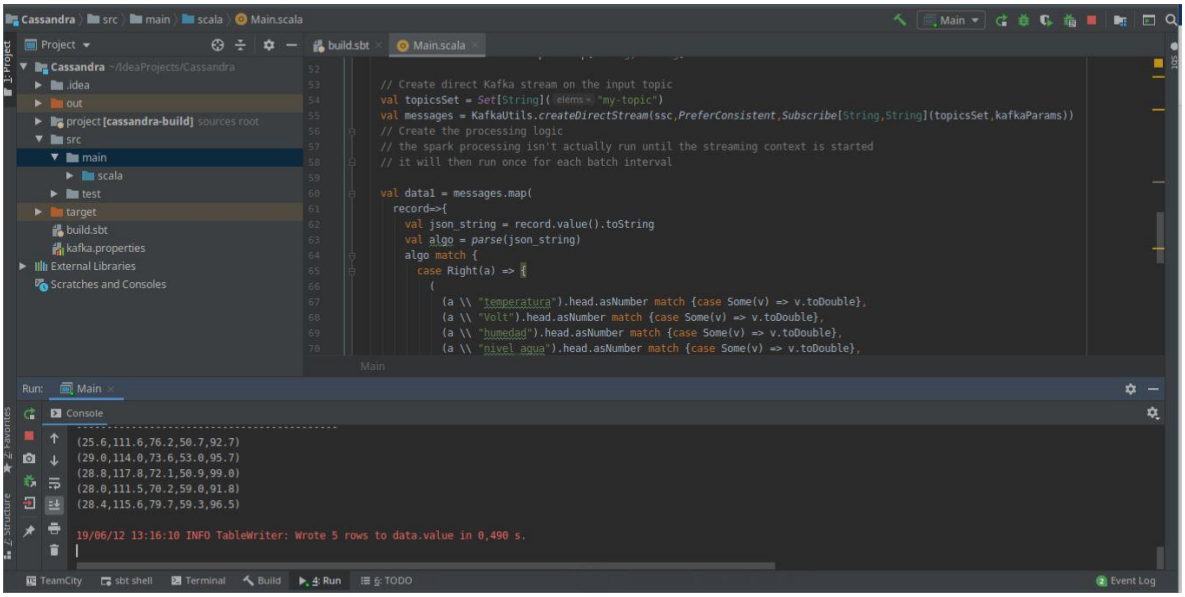


Figura 3-13. IDE IntelliJ IDEA con la ejecución de la aplicación.

En la Figura 3-14 se observa como desde el directorio de la VPS entramos al lenguaje de consultas de Cassandra para interactuar con la base de datos. Aquí seleccionamos el KeySpace "*data*" que se creó en el capítulo 2 y se verifica la tabla "*value*", donde se puede constatar que los datos son escritos en la tabla.

```
root@streaming:~/KafkaSpark# cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.9 | CQL spec 3.4.2 | Native protocol v4]
Use HELP for help.
cqlsh> USE data;
cqlsh:data> SELECT * from value;

 temperatura | Volt  | catidad_personas | humedad | nivel_agua
-------------+-------+------------------+---------+------------
        29.2 | 113.0 |             79.5 |    52.8 |       99.3
        25.2 | 117.9 |             77.6 |    55.3 |       90.8
        29.5 | 110.2 |             70.5 |    58.1 |       90.5
        29.1 | 118.4 |             76.0 |    51.3 |       91.0
        28.4 | 114.6 |             73.8 |    59.9 |       95.7
        27.9 | 112.1 |             73.3 |    54.8 |       95.0
        26.3 | 111.2 |             72.1 |    52.8 |       98.7
        25.5 | 115.4 |             74.2 |    53.1 |       94.1
        27.2 | 118.9 |             71.4 |    57.0 |       90.7
        25.0 | 114.5 |             78.8 |    51.3 |       97.6
```

Figura 3-14. Cassandra recibiendo los datos en la tabla "*value*".

La Figura 3-15 muestra coma la tabla crece mientras continúa recibiendo datos.

```
(5 rows)
cqlsh:data> select * from value;

 temperatura | Volt  | catidad_personas | humedad | nivel_agua
-------------+-------+------------------+---------+------------
        25.2 | 118.6 |             77.2 |    52.0 |       95.3
        29.5 | 115.3 |             71.9 |    50.9 |       92.6
        25.0 | 115.2 |             74.5 |    51.1 |       98.2
        29.6 | 114.7 |             78.5 |    53.9 |       91.5
        29.9 | 117.8 |             71.6 |    54.8 |       90.9
        28.8 | 114.5 |             78.1 |    51.7 |       90.4
        29.3 | 112.6 |             74.3 |    52.1 |       98.3
        26.2 | 119.4 |             77.8 |    50.2 |       94.7
        29.7 | 117.4 |             78.5 |    56.9 |       98.4

(9 rows)
cqlsh:data> select * from value;

 temperatura | Volt  | catidad_personas | humedad | nivel_agua
-------------+-------+------------------+---------+------------
        25.2 | 118.6 |             77.2 |    52.0 |       95.3
        29.5 | 115.3 |             71.9 |    50.9 |       92.6
        25.0 | 115.2 |             74.5 |    51.1 |       98.2
        29.6 | 114.7 |             78.5 |    53.9 |       91.5
        29.9 | 117.8 |             71.6 |    54.8 |       90.9
        27.8 | 111.2 |             78.6 |    50.2 |       91.6
        28.9 | 110.9 |             79.3 |    57.0 |       91.7
        28.5 | 111.8 |             79.5 |    58.6 |       92.5
        29.0 | 113.7 |             79.9 |    59.0 |       96.8
        28.8 | 114.5 |             78.1 |    51.7 |       90.4
        29.3 | 111.9 |             76.3 |    54.2 |       91.0
        26.2 | 119.4 |             77.8 |    50.2 |       94.7
        29.7 | 117.4 |             78.5 |    56.9 |       98.4

(13 rows)
cqlsh:data>
```

Figura 3-15. Tabla "*value*" recibiendo datos.

Hasta lo anterior la aplicación se había estado ejecutando desde IntelliJ IDEA en la máquina local, pero ahora se ejecuta desde la VPS con la utilización de SBT. En la Figura 3-16 se puede observar que la carpeta del proyecto "cassandra" se encuentra dentro del directorio KafkaSpark/. Aquí se accede a este directorio y con Internet disponible en la VPS se ejecuta el comando *sbt run* dentro de la carpeta. Se observa que la compilación es exitosa y empieza a ejecutarse el objeto *Main* de la aplicación, los datos procesados comienzan a aparecer en la ventana.

Se observa la conexión al clúster de Cassandra "Test Cluster" mediante la dirección 10.12.53.15 y el puerto 9042, primeramente, muestra "*0 rows*" escritas hasta que empiezan a llegar los datos. En la Figura 3-17 se puede constatar que se empiezan recibir los datos y a escribir "*5 rows*" por cada *batch* en Cassandra.

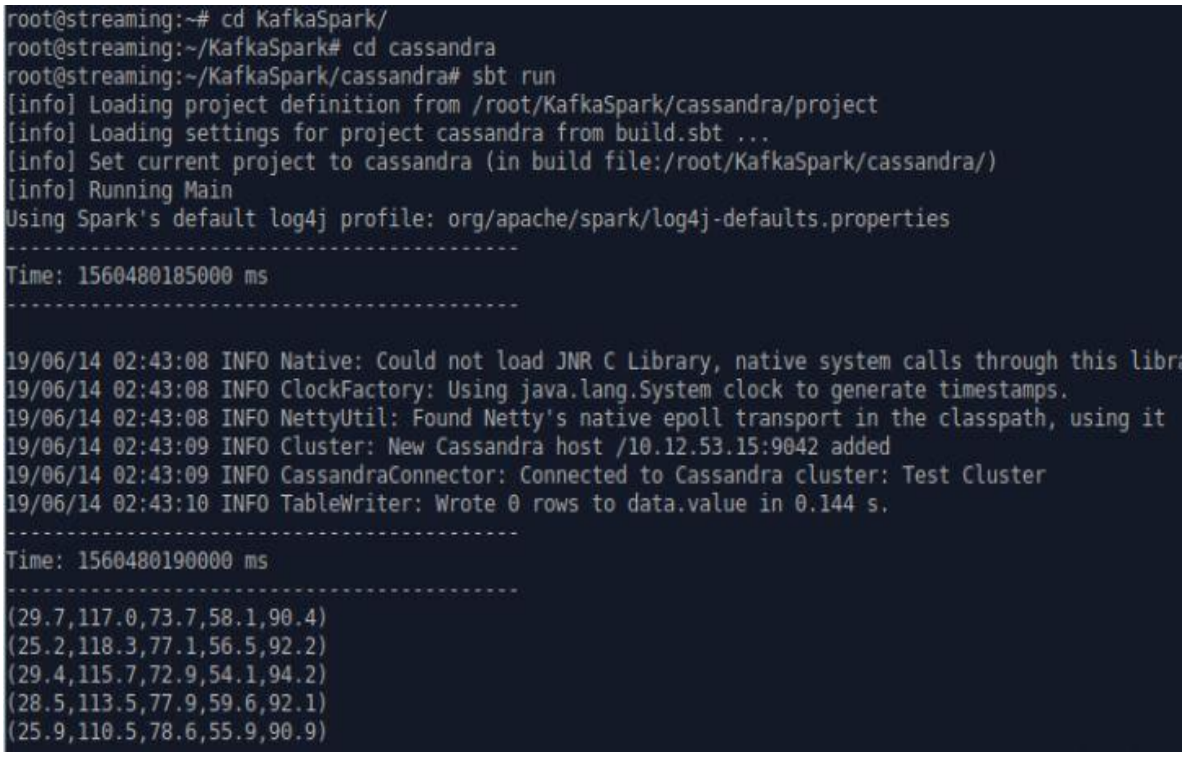


Figura 3-16. Ejecución de la aplicación mediante SBT desde la VPS.

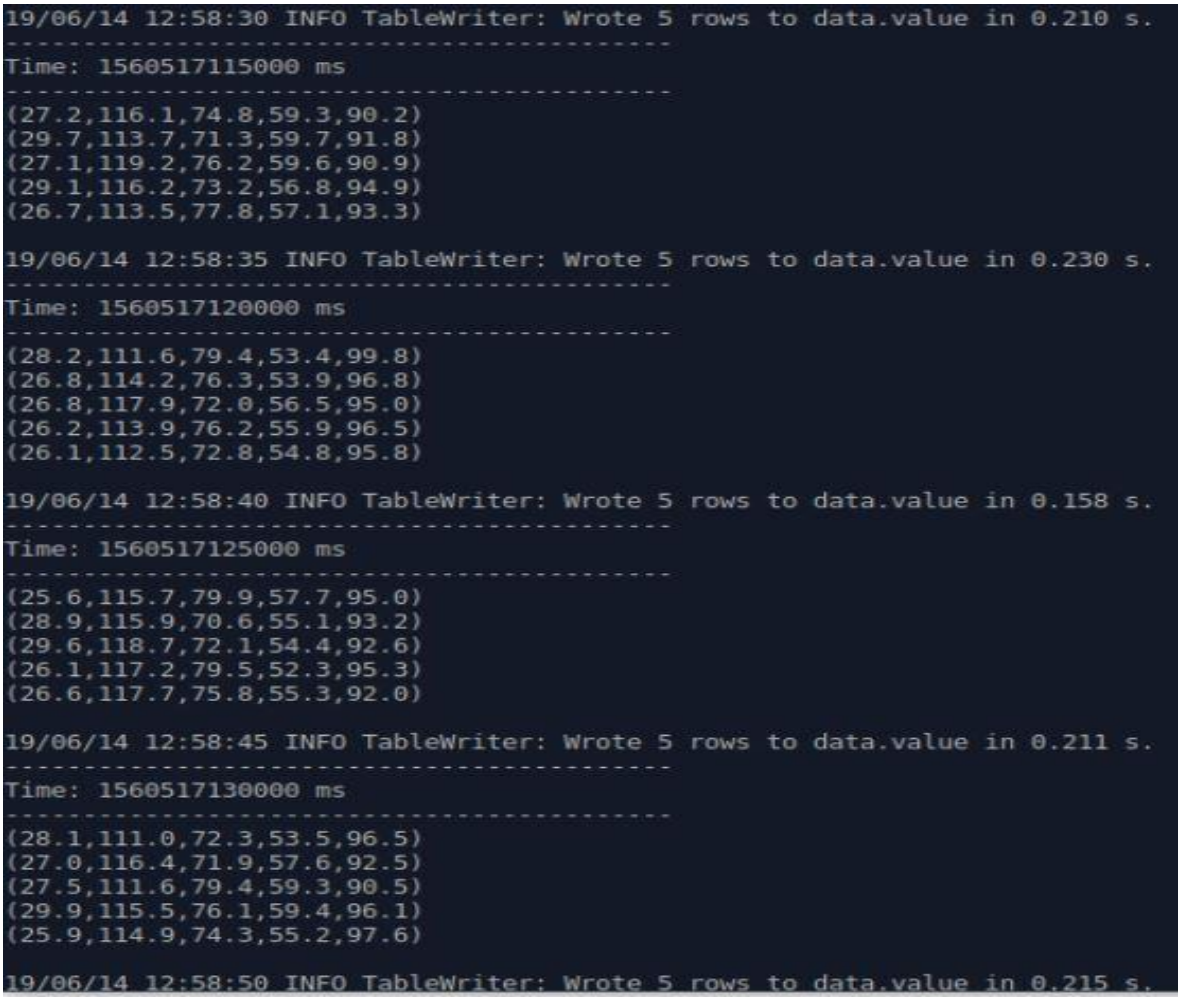


Figura 3-17. Aplicación ejecutándose.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

1 Big Data representa un gran potencial para todo tipo de empresas, aplicaciones y su uso puede llegar a proporcionar una gran ventaja en la situación que se implemente. Esta investigación se centró en el desarrollo de una arquitectura Big Data, pero este es un tema muy novedoso y se encuentra en un mundo en constante evolución.

2 Se ha implementado una plataforma viable para el procesado de datos IoT en tiempo real.

3 Se ha conectado con éxito la plataforma IoT ThingsBoard existente en la Universidad Central “Marta Abreu” de Las Villas a la plataforma Big Data desarrollada.

4 Se implementó con éxito la herramienta Apache Kafka capaz de recolectar los datos de la plataforma IoT en un tema de Kafka, para la gestión y procesado de los mismo en tiempo real por la herramienta de análisis. Apache Kafka resultó en una excelente herramienta para la adquisicion de datos en tiempo real, es una herramienta escalable y con un alto rendimiento.

5 Se ha instalado y configurado con éxito la herramienta Apache Spark para el análisis y procesado de los datos en tiempo real, así como la base de datos Cassandra para realizar el almacenamiento de los datos procesados.

6 Se realizó la integración sin problemas entre todas las tecnologías utilizadas.

7 Mediante el desarrollo de una aplicación programada en lenguaje Scala, se pudo poner a prueba la plataforma Big Data.

8 Se combrobó el funcionamiento de la plataforma con éxito donde los datos son recolectados por Kafka desde la plataforma IoT, son correctamente procesados por Spark Streaming y almacenados en la base de datos Cassandra.

9 Este proyecto ha tenido el alcance esperado, se consiguió una formación bastante extensa en las principales tecnologías que permiten el despliegue de una arquitectura Big Data en tiempo real y con la realización de un caso de uso práctico se pudo comprobar que Spark, y específicamente su módulo Spark Streaming resulta muy útil

en el manejo y explotación de grandes cantidades de datos, aunque en el mundo tan cambiante de Big Data siempre aparecerán otras tecnologías con funcionalidades semejantes que le harán competencia.

Recomendaciones

En este proyecto se han cumplido todos los objetivos establecidos, pero existen diversos aspectos en los que la plataforma puede ser expandida.

1 Se podría ampliar el sistema, creando más temas en Kafka para soportar más diversidad de datos y lograr que estos queden mejor etiquetados en diferentes temas según la naturaleza de los mismos, de este modo mediante una estructura más amplia y organizada se puede realizar un mejor análisis.

2 Se podría aprovechar la biblioteca MLib de Spark para la aplicación de algún algoritmo de *Machine Learning* sobre los datos y realizar predicciones o según el caso que se implemente. De este modo se puede llegar a un mejor conocimiento de los datos que se reciben y se podría realimentar hacía la misma plataforma IoT que envía la información.

3 Se puede implementar alguna herramienta para la visualización de la información procesada. Una alternativa puede ser suscribir una base de datos ElasticSearch al servidor de Kafka e instalar el plugin Kibana en ElasticSearch para permitir la visualización de los datos con gráficas y tablas.

4 Las aplicaciones Spark, Kafka y Cassandra están pensadas para trabajar en modo clúster, es decir, modo distribuido donde hay varias máquinas trabajando en paralelo. Para lograr un mejor rendimiento del sistema, montar un clúster con todas estas herramientas es un posible trabajo futuro.

REFERENCIAS BIBLIOGRÁFICAS

[1] «Las 5 V's del Big Data», *Mediapost*, 14-nov-2014. [En línea]. Disponible en: https://www.mediapost.es/quantic/las-5-vs-del-big-data. [Accedido: 06-jun-2019].

[2] J. M. Rodríguez Cano de Santayana, «Plataforma en Tiempo Real Big Data de procesamiento de sensores IoT para la predicción de la demanda de una franquicia de restauración», p. 82, 2018.

[3] A. I. Estaire Muñoz, «Estudio y puesta en marcha de un sistema de streaming en Big Data», p. 80, 2017.

[4] F. V. C. Nicolalde, «Análisis de Big Data en IoT para campos de Cadenas de Suministro Inteligentes», p. 146, 2018.

[5] J. Nonino, «Procesamiento de Datos en Tiempo Real Utilizando Docker, Zookeeper, Kafka y Storm», Especialización en Ingeniería en Sistemas de Información, Universidad Tecnológica Nacional, Córdoba, 2016.

[6] J. M. Montero, *Detección de fraude bancario en tiempo real utilizando tecnologías de procesamiento distribuido*. 2016.

[7] M. Pérez Esteso y J. C. Dueñas López, «Análisis de arquitecturas de procesado de streaming Big Data», Trabajo Fin De Máster, Universidad Politécnica de Madrid, Madrid, 2015.

[8] R. Cacho Zurrunero, «Desarrollo de una herramienta de análisis, procesado y visualización de fuentes de datos empleando técnicas de Big Data utilizando Apache Spark», p. 65, 2017.

[9] «Welcome to Apache Flume — Apache Flume». [En línea]. Disponible en: https://flume.apache.org/. [Accedido: 06-jun-2019].

[10] F. Fernández Garrido, «Arquitectura Big Data de ingesta en Real Time», 2017.

[11] L. Garcia Arce y D. Contreras Bárcena, «Implementación de una plataforma Big Data propia basada en soluciones OpenSource», Grado en Ingeniería Telemática, Universidad Pontificia Comillas, Madrid, 2016.

[12] «Apache Kafka», *Apache Kafka*, 06-jun-2019. [En línea]. Disponible en: https://kafka.apache.org/. [Accedido: 06-jun-2019].

[13] M. Kumar y C. Singh, *Building Data Streaming Applications with Apache Kafka: Design, develop and streamline applications using Apache Kafka, Storm, Heron and Spark*. Packt Publishing, 2017.

[14] N. Narkhede, G. Shapira, y T. Palino, *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale*. O'Reilly Media, Inc., 2017.

[15] «Apache ZooKeeper», 06-jun-2019. [En línea]. Disponible en: https://zookeeper.apache.org/. [Accedido: 06-jun-2019].

[16] «Apache Storm». [En línea]. Disponible en: http://storm.apache.org/. [Accedido: 06-jun-2019].

[17] «Apache Spark™ - Unified Analytics Engine for Big Data», 06-jun-2019. [En línea]. Disponible en: https://spark.apache.org/. [Accedido: 06-jun-2019].

[18] M. E. Verjaga-Felgueras, «Análisis de datos y extracción de conocimiento utilizando Big Data», p. 116, 2018.

[19] M. Kumar y C. Singh, *Building Data Streaming Applications with Apache Kafka*. Packt Publishing, 2017.

[20] «1. Introduction to Data Analysis with Spark - Learning Spark [Book]», 06-jun-2019. [En línea]. Disponible en: https://www.oreilly.com/library/view/learning-spark/9781449359034/ch01.html. [Accedido: 06-jun-2019].

[21] «Spark SQL & DataFrames | Apache Spark», 06-jun-2019. [En línea]. Disponible en: http://spark.apache.org/sql/. [Accedido: 06-jun-2019].

[22] «MLlib | Apache Spark», 06-jun-2019. [En línea]. Disponible en: http://spark.apache.org/mllib/. [Accedido: 06-jun-2019].

[23] «GraphX | Apache Spark», 06-jun-2019. [En línea]. Disponible en: http://spark.apache.org/graphx/. [Accedido: 06-jun-2019].

[24] «Spark Streaming | Apache Spark», 06-jun-2019. [En línea]. Disponible en: http://spark.apache.org/streaming/. [Accedido: 06-jun-2019].

[25] N. Neeraj, *Mastering Apache Cassandra*. Packt Publishing Ltd, 2013.

[26] A. Cassandra, «Apache cassandra», *Website Available Online Httpplanetcassandra Orgwhat--Apache-Cassandra*, p. 13, 2014.

[27] «Apache Cassandra», 06-jun-2019. [En línea]. Disponible en: http://cassandra.apache.org/. [Accedido: 06-jun-2019].

[28] S. Hoffman, *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd, 2013.

[29] «Cassandra vs. MongoDB». [En línea]. Disponible en: https://scalegrid.io/blog/cassandra-vs-mongodb/. [Accedido: 10-jun-2019].

[30] C. M. Martín Hernández, «Vitualización sobre plataforma opensource: Alternativas a las soluciones comerciales», 2011.

[31] «How To Install Apache Kafka on Ubuntu 18.04», *DigitalOcean*. [En línea]. Disponible en: https://www.digitalocean.com/community/tutorials/how-to-install-apache-kafka-on-ubuntu-18-04. [Accedido: 14-jun-2019].

[32] «Spark Streaming + Kafka Integration Guide - Spark 2.2.0 Documentation». [En línea]. Disponible en: https://spark.apache.org/docs/2.2.0/streaming-kafka-integration.html. [Accedido: 07-jun-2019].

[33] N. Neeraj, *Mastering Apache Cassandra*. Packt Publishing Ltd, 2013.

[34] thingsboard, «Kafka Plugin», *ThingsBoard*. [En línea]. Disponible en: https://thingsboard.io/docs/reference/plugins/kafka/. [Accedido: 07-jun-2019].

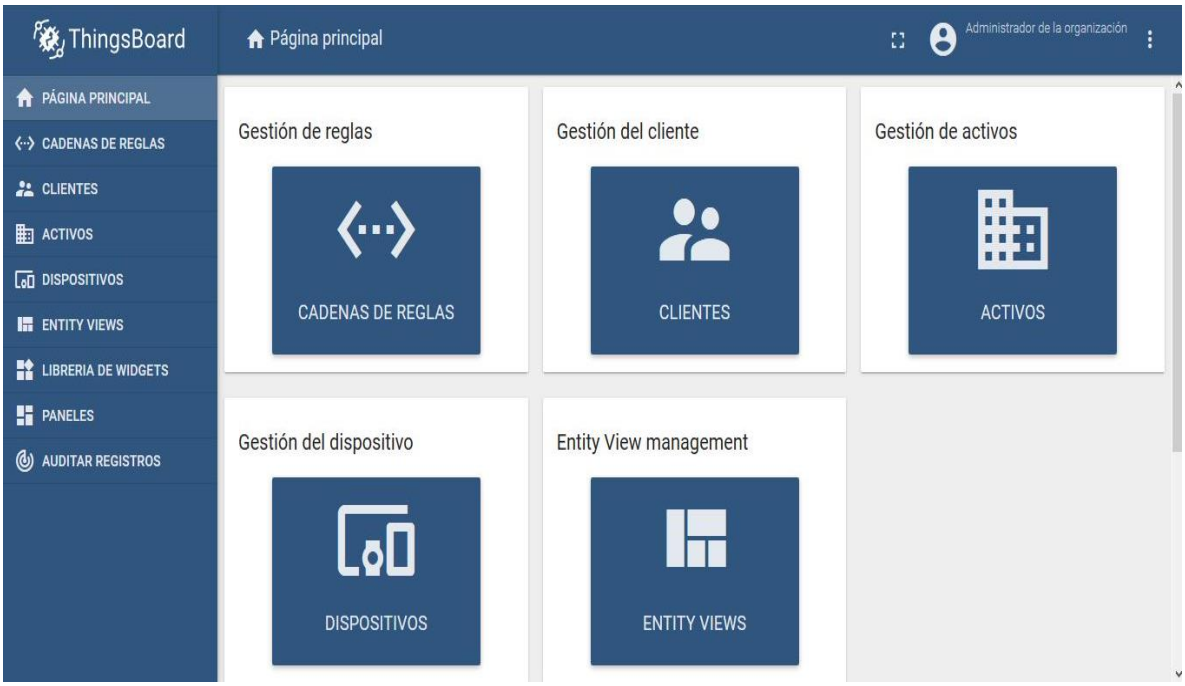
[35] «Spark Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher) - Spark 2.2.0 Documentation». [En línea]. Disponible en: https://spark.apache.org/docs/2.2.0/streaming-kafka-0-10-integration.html. [Accedido: 07-jun-2019].

[36] O. Roncal Andrés, «Entorno de desarrollo integral para el lenguaje de modelado Tara», 2014.

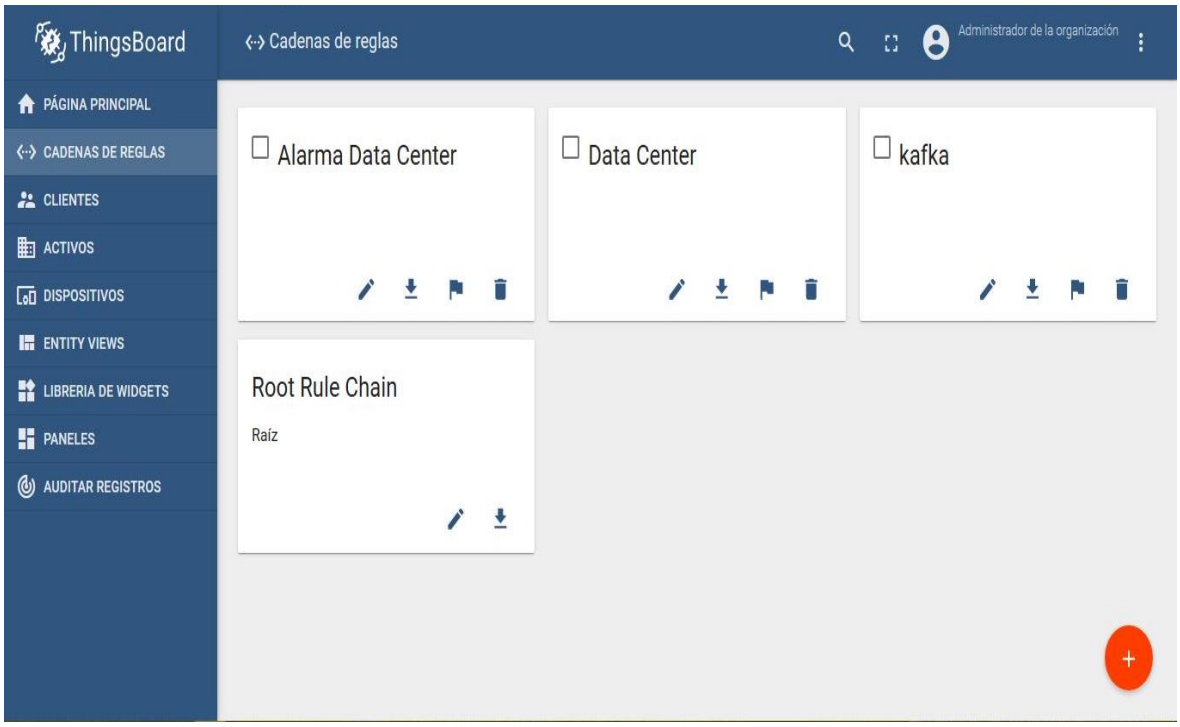
ANEXOS

Anexo I Configuración del complemento Kafka en Thingsboard.

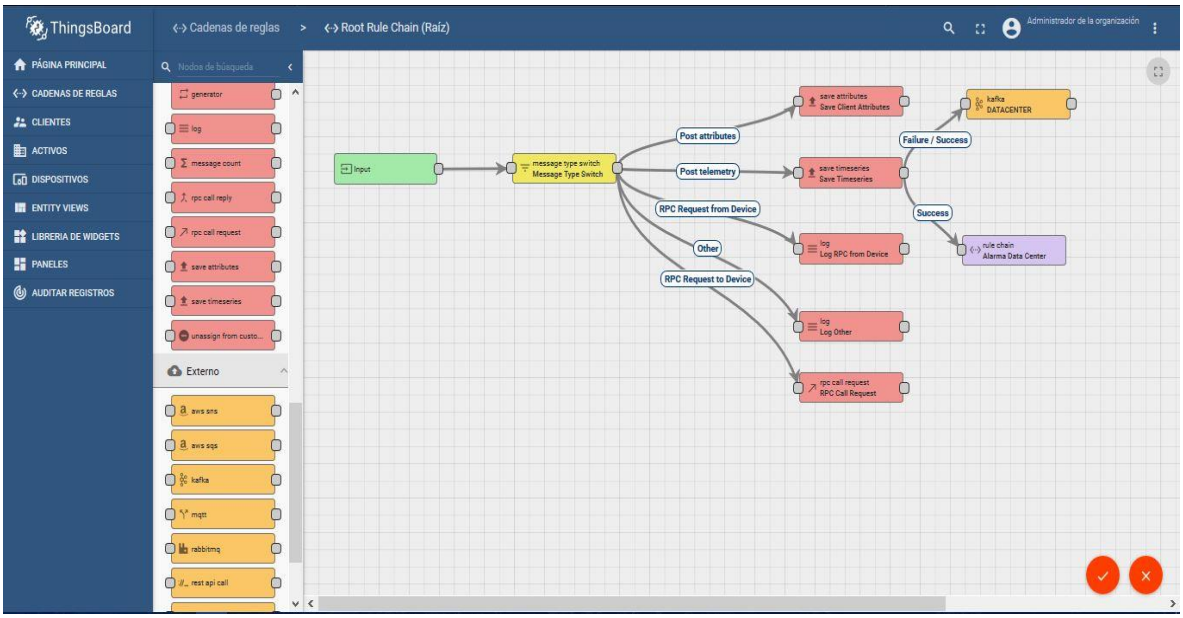
Se entra en la plataforma ThingsBoard:



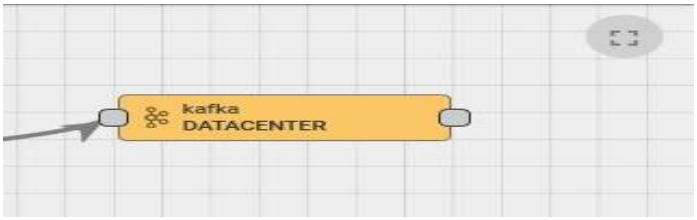
Ir al menú y se da clic en ‘Cadenas De Reglas’:



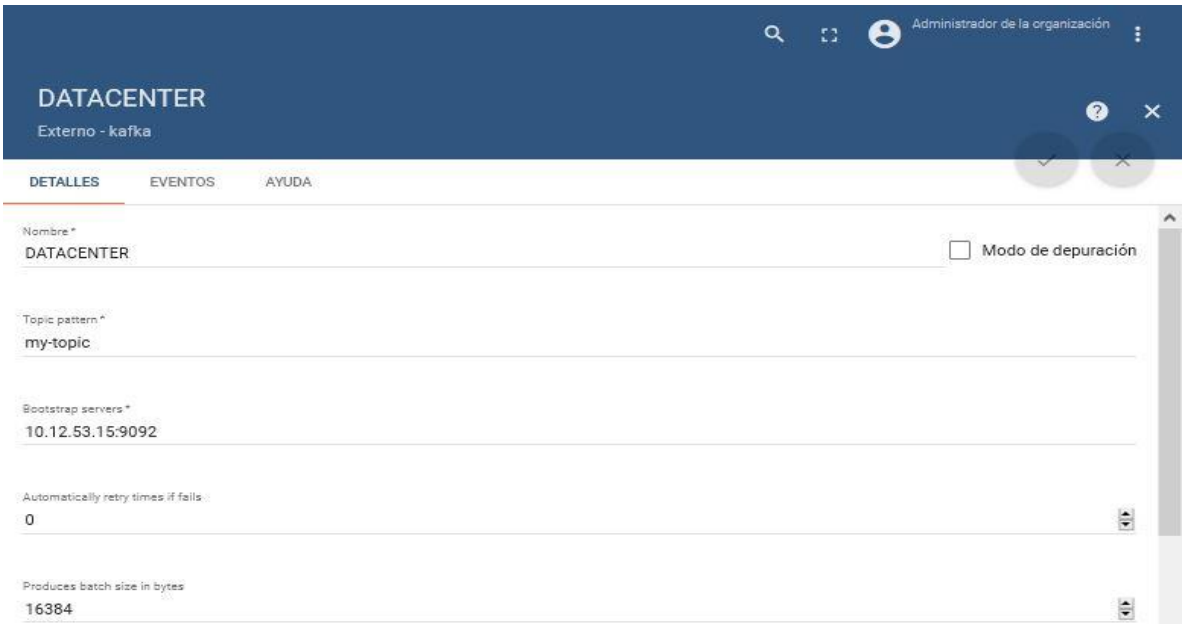
Se entra al componente ‘Root Rule Chain’:

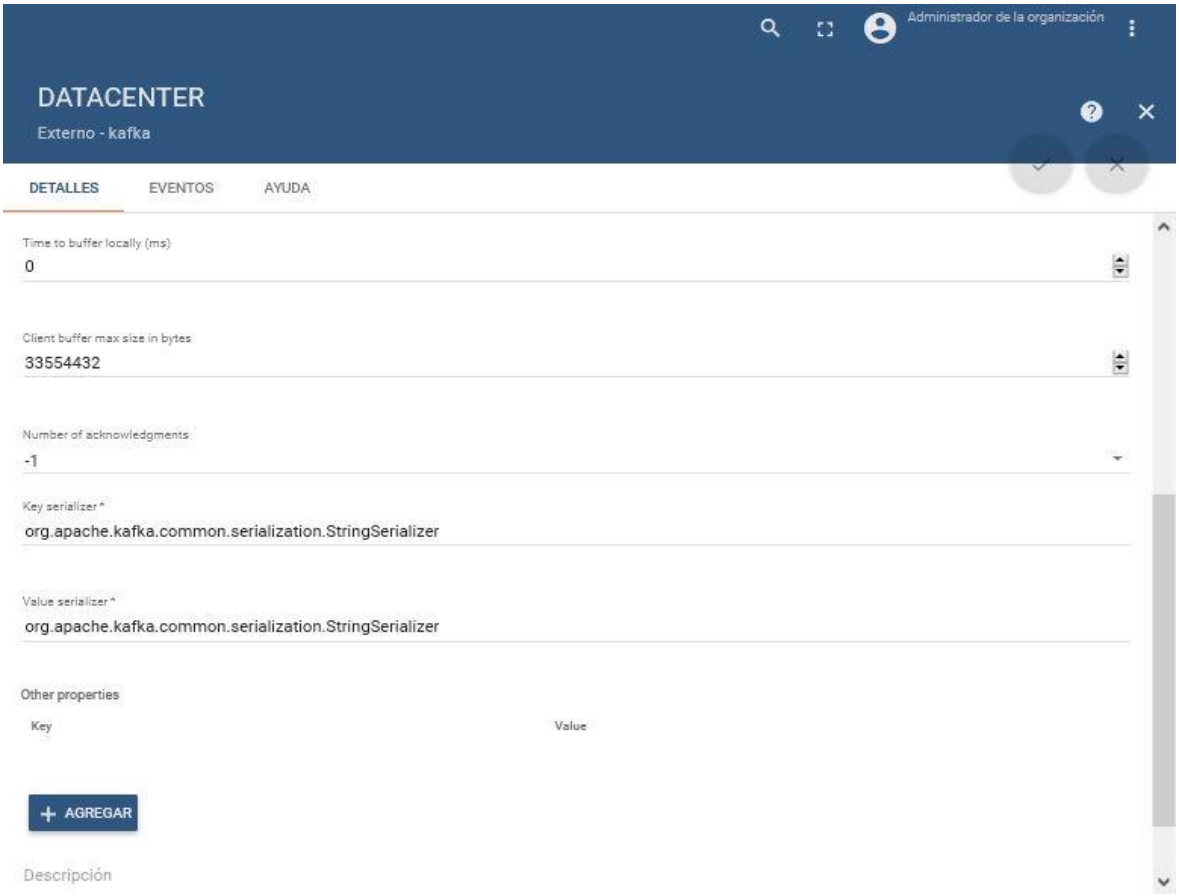


Se observa en la estructura el componente llamado “kafka DATACENTER”:



Se da doble clic sobre él y sale la siguiente ventana de configuración:





Una vez llenados los campos con la configuración del Kafka externo, se guarda y se aplican los cambios en la misma venta de configuración y luego en la ventana de raíz para que los cambios se apliquen satisfactoriamente. De esta forma se pueden enviar los datos al Kafka externo.

Anexo II Código del archivo built.sbt.

```
name := "cassandra"

version := "0.1"

scalaVersion := "2.11.8"

//spark

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.0" % "provided"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.3.0"

libraryDependencies += "org.apache.spark" %% "spark-streaming" % "2.3.0"
```

```
//spark + kafka
libraryDependencies += ("org.apache.spark" %% "spark-streaming-kafka-0-10" % "2.3.0").exclude("org.spark-project.spark", "unused")
libraryDependencies += "org.apache.kafka" % "kafka-clients" % "1.1.0"
//cassandra
libraryDependencies += ("com.datastax.spark" %% "spark-cassandra-connector" % "2.3.0").exclude("io.netty", "netty-handler")
// circle (json lib)
val circeVersion = "0.10.0"
libraryDependencies ++= Seq(
  "io.circe" %% "circe-core",
  "io.circe" %% "circe-generic",
  "io.circe" %% "circe-parser"
).map(_ % circeVersion)
```

Anexo III Código de la aplicación.

```
import org.apache.log4j.{Level, Logger}
import scala.util.parsing.json.JSON
// Importaciones para leer del archive kafka.properties
import java.io.FileReader
import java.util.Properties
import scala.collection.JavaConversions._
// Básicas importaciones de Spark
import org.apache.spark.streaming._
import org.apache.spark.sql.SparkSession
```

```
//Importa Circe (Json)
import io.circe._, io.circe.parser._
// Importa el conector de Spark para Cassandra
import com.datastax.spark.connector._
// Spark Streaming + Kafka imports
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010._
object Main {
  def main(args: Array[String]): Unit = {
    //Opcionalmente: apaga el logger para hacer muestras más claras.
    Logger.getLogger("org").setLevel(Level.ERROR)
    Logger.getLogger("akka").setLevel(Level.ERROR)
    val spark = SparkSession.builder
      .appName("SKC")
      .config("spark.cassandra.connection.host", "10.12.53.15")
      .config("spark.cassandra.connection.port", "9042")
      .master("local[*]")
      .getOrCreate()
    val ssc = new StreamingContext(spark.sparkContext, Seconds(5))
    // Creación de un timer que detiene el procesamiento luego de 60 segundos y muestra los
resultados
    val timer = new Thread() {
      override def run() {
        Thread.sleep(1000 * 5)
```

```
      ssc.stop()
    }
  }
  // lee del archive kafka.properties
  val kafkaProps = new Properties()
  kafkaProps.load(new FileReader("kafka.properties"))
  val kafkaParams = kafkaProps.toMap[String, String]
  // Creación del direct Kafka stream en el topic de entrada
  val topicsSet = Set[String]("my-topic")
  val messages =
KafkaUtils.createDirectStream(ssc,PreferConsistent,Subscribe[String,String](topicsSet,kaf
kaParams))
  // Creación de la lógica de procesamiento
  val data1 = messages.map(
   record=>{
     val json_string = record.value().toString
     val algo = parse(json_string)
     algo match {
       case Right(a) => {
         (
           (a \\ "temperatura").head.asNumber match {case Some(v) => v.toDouble},
           (a \\ "Volt").head.asNumber match {case Some(v) => v.toDouble},
           (a \\ "humedad").head.asNumber match {case Some(v) => v.toDouble},
           (a \\ "nivel_agua").head.asNumber match {case Some(v) => v.toDouble},
           (a \\ "catidad_personas").head.asNumber match {case Some(v) => v.toDouble}
```

```
          )
        }
       }
     }
   )
   data1.print() // Muestra para ver los resultados
// Salva cada RDD en la tabla “value” de Cassandra
   data1.foreachRDD(rdd => {
     rdd.saveToCassandra("data","value")
   })
   ssc.start() // Comienza el streaming context
   ssc.awaitTermination() // Bloquea mientras el streaming context está ejecutándose
  }
}
```