Universitat Oberta de Catalunya
uoc.edu

Big Data y Seguridad

Jorge Luis Zambrano Martínez
Plan de Estudios del Estudiante
Big Data y Seguridad

Dirigido por:
Enric Hernández

04 de junio de 2018

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Big Data y Seguridad*
Nombre del autor:	*Jorge Luis Zambrano Martínez*
Nombre del consultor/a:	*Eric Hernández*
Nombre del PRA:	
Fecha de entrega (mm/aaaa):	06/2018
Titulación:	*Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones*
Área del Trabajo Final:	
Idioma del trabajo:	*Español*
Palabras clave	*Big Data; Seguridad; Spark*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El objetivo de esta memoria es analizar cual es la mejor arquitectura de Big Data que se adapte de manera sustancial al realizar una gran cantidad de ingestas de datos procesados en tiempo real suministrados por las Notarías Españolas.

Tomando como punto de partida el ecosistema de Hadoop, en donde se encuentra un stack completo de tecnologías Big Data, planteándonos resolver un problema cotidiano y que podrá ser extrapolable a muchos ámbitos de ingesta de datos que requieran de un procesamiento en tiempo real con su debido almacenamiento en un clúster.

Los resultados que se obtiene en este proyecto, por una parte es la creación de grandes cantidades de datos sintéticos con características similares a los datos de los servidores de las Notarías, con el fin de probar la arquitectura elegida, y por otra parte, la creación de una arquitectura virtualizada a pequeña escala totalmente funcional dentro de un clúster virtual, para la captura de datos en tiempo real, con su respectivo procesamiento y almacenamiento en el clúster virtual.

Abstract (in English, 250 words or less):

The goal of this paper is to know which is the best Big Data architecture that is substantially adapted to make a large amount of data processed in real-time provided by the Spanish Notaries.

Taking as a starting point the Hadoop ecosystem, where there is a stack of Big Data technologies, which they can solve a daily problem. Those technologies can be extrapolated in many data areas that require real-time processing with proper storage in a cluster.

The results obtained in this project are, on the one hand, the creation of large amounts of synthetic data with characteristics similar to the data of the servers of the Notary, in order to test the chosen architecture, and on the other hand, the creation of a fully functional, small-scale virtualized architecture within a virtual cluster, for real-time data capture, with their respective processing and storage in the virtual cluster.

Índice

Índice de Tablas

Índice de Figuras

1. Introducción y Objetivos

1.1 Planteamiento del Problema

En la actualidad la cantidad de información generada por distintos procesos en distintos campos profesionales y científicos se ven en la necesidad de procesarla, no de una manera tradicional, debido a que se crean grandes volúmenes de datos teniendo que ser almacenados en servidores o clústeres.

La idea del proyecto es analizar una arquitectura que resuelva la problemática que presentan las Notarías Españolas. Existen alrededor de unas 3,000 sedes entre los cuales están los Colegios Notariales, el Consejo, etc.

En cada sede existe un Notario que es el custodio de la información que físicamente esta almacenada en grandes libros propiamente de casos privados, que llevan registro con un completo secreto por ser datos sensibles llevando a la necesidad de que todo este funcionamiento sea de forma electrónica.

En cada notaría se cuenta con un servidor donde se tiene almacenado dicha información con su correspondiente protocolo electrónico que se intercomunican entre sí, enviándose constantemente información. La cantidad de información procesada es muy elevada por lo que necesita una respuesta en tiempo real teniendo un clúster Oracle escalable.

Dado este problema a resolver, es necesario realizar un análisis de una arquitectura distribuida para el procesamiento de grandes volúmenes de información que procese los dichos datos en tiempo real y de forma rápida. Con esta premisa, se analizará el tipo de arquitectura de Big Data más adecuada para cubrir las necesidades del problema planteado.

Para ello se tiene pensado analizar el ecosistema de Hadoop con varios servidores que sea distribuido y escalable, incluyendo los diferentes componentes que posean, y presentando que arquitectura es la más conveniente para resolver el problema de este escenario.

1.2 Objetivos

Como objetivo principal se tiene:

- Analizar y presentar qué tipo de arquitectura de Big Data llega a tener un mejor tiempo de respuesta frente a la inyección de la gran cantidad de información suministrada por las Notarías Españolas.

Como objetivos secundarios se tiene:

- Implantar un ecosistema de Hadoop con los diferentes componentes en un clúster virtual.
- Generar datos sintéticos por medio de un programa desarrollado, que permitan simular una situación real de datos con su formato requerido en el clúster virtual.
- Inyectar la cantidad de datos sintéticos generados al clúster con la arquitectura de Big Data escogida para analizar las tres directrices del Big Data correspondientemente en tiempo real.

1.3 Enfoque y método seguido

Para este proyecto, desarrollaremos un programa que genere datos sintéticos en un formato similar al que utilizan los servidores de las Notarías Españolas. Dicho programa creará de forma aleatoria 100.000 datos sintéticos aproximadamente, ejecutándose de 300 a 400 veces, con el objetivo de simular una situación real de Big Data.

Estos datos sintéticos tendrán características del servidor como los registros de cantidad de bytes, la dirección IP de origen y destino, tiempos de conexión, puerto por el que se esta conectando, los servicios que se esta ocupando, los tipos de accesos legítimos y los no legítimos.

La estrategia es poner a prueba las arquitecturas de Big Data del ecosistema Hadoop con su respectivo análisis, e incluso indagar la seguridad a la cual se le puede colocar ya que son datos sensibles que no pueden permanecer sin una apropiada seguridad.

Terminando con una virtualización de un clúster, al cual se implementará la arquitectura escogida del ecosistema Hadoop, y se realizarán pruebas de inyección de datos sintéticos en tiempo real al clúster virtual con su

correspondiente almacenamiento.

1.4 Planificación del Trabajo

Para montar una arquitectura con características de un Big Data, se debería disponer de un clúster físico. Sin embargo, no se dispone de dichos recursos, por ese motivo se debe realizar un clúster virtual para emular dichos comportamientos, siempre y cuando soporte los recursos el computador anfitrión.

El diagrama de Gantt se adjunta en el apartado de anexos, el cual detalla a mayor detalle las fases correspondientes del proyecto. A continuación, se detalla las fases del proyecto que se han seguido.

Tarea: Estudio de arquitecturas de Big Data - Hadoop	
Fecha inicio: 13-03-2018	Fecha fin: 23-03-2018
Descripción: Esta tarea representa la investigación de las arquitecturas que posee el ecosistema Hadoop, analizando por medio de artículos científicos cuál arquitectura se acopla mejor a las necesidades del escenario para este proyecto.	

Tarea: Desarrollo de aplicación generadora de datos sintéticos	
Fecha inicio: 24-03-2018	Fecha fin: 09-04-2018
Descripción: Esta tarea describe el desarrollo de la aplicación realizada en Python, que se encargará de elaborar los 100.000 datos sintéticos en cada unos de los archivos de valores separados por comas (CSV). Para que sea sustentable un gran volumen de datos, la aplicación elaborará alrededor de 400 archivos.	

Tarea: Instalación de entorno de simulación	
Fecha inicio: 10-04-2018	Fecha fin: 19-04-2018
Descripción: Esta tarea se debe montar el clúster virtual utilizando VirtualBox. Se implementara un nodo maestro, conjuntamente con dos nodos esclavos y un nodo NFS. Este clúster virtual se tomará aspectos funcionales como si se tratara de un clúster montado de manera física.	

Tarea: Implementación de arquitectura Big Data	
Fecha inicio: 20-04-2018	Fecha fin: 07-05-2018
Descripción: Esta tarea implementaremos la arquitectura Big Data seleccionada en el clúster virtual. Por lo que, la arquitectura Apache Spark y Apache Hadoop (HDFS) se implantará en el nodo maestro y en los nodos esclavos, creando un ecosistema completo.	

Tarea: Escritura de datos obtenidos	
Fecha inicio: 08-05-2018	Fecha fin: 21-05-2018
Descripción: Esta tarea se desarrolla de una aplicación en Python para la ingesta de los datos sintéticos con el módulo de Spark Streaming, y que sea en tiempo real tanto la lectura de los datos como la propia escritura de los datos. El almacenamiento de los datos sintéticos se hará de dos formas, i) en el nodo NFS y ii) por el HDFS.	

1.5 Estructura de la memoria

La presente memoria esta compuesta por los siguientes capítulos:

Capítulo 2: Se presentará una breve introducción con respecto a los clústeres donde funcionará la arquitectura escogida. También, una introducción del amplio ecosistema de Hadoop con sus distintas arquitecturas.

Capítulo 3: Se introducirá a más detalle las arquitecturas Spark y Hadoop, que se van a utilizar para en el presente trabajo, detallando sus componentes, ventajas que tengan cada uno de ellos.

Capítulo 4: Se presenta el entorno de simulación con un clúster virtual, e implementando las arquitecturas Spark y Hadoop (HDFS) en el mismo clúster con la generación de datos sintéticos. Además, la generación de datos sintéticos es en CSV, y la ingesta de estos datos al clúster para su correspondiente procesamiento y almacenamiento estará en formato JSON en el mismo clúster usando el NFS o el HDFS.

Conclusiones: Se presentan las conclusiones del trabajo que contendrá una reflexión final de los aspectos del trabajo y el por que se ha elegido dichas arquitecturas para cumplir los objetivos impuestos.

2. Estado del arte

2.1 Introducción

Hoy en día, dada a la importancia que se tiene en la generación, almacenamiento, procesamiento y visualización de datos en diferentes áreas con un coste accesible y de gran potencia de cálculo, se tiene la necesidad de crear un sistema de cómputo que trate en gran medida de los diversos paradigmas en cuanto a investigación, empresarial, educativo, etc.

Por ello, nace Big Data para solucionar problemas como el almacenamiento y procesamiento de grandes volúmenes de datos, así como de su interpretación y análisis. De este modo, podemos observar que al estar interconectados los usuarios, generan grandes cantidades de datos que pueden ser analizados. Los datos pueden llegar a ser estructurados y no estructurados, representando una complejidad en la forma que se tendría que almacenar y analizar dichos datos.

Por esta razón, aparece en el mercado como una solución inspirada en los documentos Google para el MapReduce y Google File System un framework denominado Apache Hadoop, dando un nuevo paradigma de almacenamiento y procesamiento de los datos. En el Capítulo 3, hablaremos a más profundidad de este framework.

Sin embargo, este framework con su ecosistema debe descansar sobre un conjunto de computadores convencionales interconectados entre sí denominados clústeres, y así poder aprovechar un alto rendimiento computacional, una alta tolerancia a fallos, un eficiente equilibrio de carga en cuanto a procesos y una vertiginosa escalabilidad.

2.2 Ecosistema de Hadoop

El crecimiento vertiginoso que ha construido Hadoop con su gran familia al dar soluciones para el almacenamiento, gestión, interacción y análisis de grandes cantidades de datos, siempre integradas en un amplio ecosistema de código abierto creado por la comunidad de Apache Software Fundation. Actualmente se desarrollan decenas de proyectos sin restricción alguna, al que pueden llegar a funcionar de forma independiente.

Los impulsos de desarrollar más proyectos de código abierto son las necesidades y posibilidades de mejorar el Sistema de Archivos Distribuidos de Hadoop (HDFS) y su motor de procesamiento de tareas (MapReduce). Algunos de estos proyectos ofrecen almacenamiento de datos más sofisticados, mayor rapidez (Spark), asignar recursos sobre Hadoop en aplicaciones ejecutadas de forma simultanea, son entre ellos algunos proyectos que potencian las capacidades que tiene Apache Hadoop para gestionar cantidades voluminosas de datos sean estructurados o no estructurados.

A continuación se mostrara algunos proyectos del ecosistema de Hadoop [3].

2.2.1 Apache Pig

Es un framework desarrollado inicialmente por Yahoo, que permite a los usuarios de Hadoop centrarse en el análisis de datos y menos en la creación de programas MapReduce. Se maneja a través de un lenguaje de scripting de alto nivel denominado PigLatin. Este lenguaje de programación Pig esta pensado para trabajar sobre cualquier tipo de datos.

2.2.2 Apache Avro

Es un sistema de serialización de datos que se usa para procesarlos y almacenar esos datos de forma que el rendimiento en tiempo sea efectivo. Dicha serialización puede basarse en archivo de texto plano, JSON, o binario. Además, esta optimizado para minimizar el espacio en disco necesario para nuestros datos.

2.2.3 Apache Hive

Es un sistema de Data Warehouse para Hadoop que le facilita el uso de agregación de los datos y el análisis de grandes conjuntos de datos que son almacenados en Hadoop. Además, proporciona métodos de consulta de los datos usando su propio lenguaje HiveQL y tiene interfaces JDBC/ODBC para conectarse a herramientas Business Intelligence.

2.2.4 Apache Chukwa

Es un sistema de captura los datos y de análisis que trabajo conjuntamente

con Hadoop para procesar y analizar grandes cantidades de datos, en el que incluye herramientas para visualizar y monitorizar los datos capturados.

2.2.5 Apache Mahout

Es un framework de algebra lineal distribuido y Scala Domain Specific Languages (DSL) matemáticamente expresivo, que esta diseñado para permitir que los matemáticos, estadísticos, y científicos implementen rápidamente algoritmos como clustering y de clasificación.

2.2.6 Apache HBase

Es la base de datos que posee Hadoop. Es usado por Hadoop cuando requiere de lecturas y escrituras en tiempo real y de acceso aleatorio a grandes conjuntos de datos. HBase no admite SQL y no sigue un esquema relacional.

2.2.6 Apache TEZ

Es un framework de programación de flujo de datos generalizado, construido en Hadoop YARN que proporciona un motor potente y a su ves flexible para ejecutar un arbitrario Gráfico Acíclico Dirigido (DAG) de una tarea para procesar datos en lotes y casos de usos interactivos.

2.2.9 Apache ZooKeeper

Es un servicio centralizado para mantener la información de configuración, nombrando, proporcionando sincronización distribuida y servicios grupales. Estos tipos de servicios son utilizados de una forma y otra por aplicaciones distribuidas.

2.2.10 Apache Spark

Es un motor de calculo rápido que procesa los datos de Hadoop a una velocidad superior a MapReduce en tiempo real. Además, proporciona un modelo de programación simple y expresivo que admite una amplia gama de aplicaciones incluyendo Extracción, Transformación y Carga (ETL), aprendizaje automático (machine learning), procesamiento de flujo de datos

(streaming) y cálculo de gráficos. En el Capítulo 3, hablaremos a más profundidad sobre este framework.

2.2.11 Apache Ambari

Es una herramienta basada en la web para aprovisionar, administrar y monitorizar clústeres de Apache Hadoop que incluye soporte para Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig y Sqoop. Además, proporciona un panel para ver el estado del clúster, como mapas de calor y la capacidad de observar las aplicaciones Hive, Pig y MapReduce visualmente junto con las características de diagnósticos del rendimiento de una forma sencilla.

2.2.12 Apache Cassandra

Es una base de datos vinculada a la escalabilidad y alta disponibilidad sin comprometer el rendimiento. Dada a su escalabilidad lineal y la probada tolerancia a fallos sea en hardware o en la nube, lo convierte en una plataforma perfecta para datos de misión critica. Es la mejor en su clase para replicar en múltiples centros de datos, ya que brinda una menor latencia a sus usuarios y puede sobrevivir a interrupciones regionales.

2.2.13 Apache Flume

Es un sistema distribuido para capturar, agregar y mover grandes cantidades de datos de diferentes orígenes a un repositorio central, simplificando el proceso de recolectar los datos para almacenarlos en Hadoop y analizarlos. Apache Chukwa y Flume son parecidos, la diferencia entre ellos radica en Chukwa esta diseñado para ser usado en lotes.

2.2.14 Apache Kafka

Es una librería cliente para procesar y analizar datos almacenados en Kafka y escribir los datos resultantes en Kafka o enviar el resultado final a un sistema externo. Se basa en conceptos importantes de procesamiento de flujo de datos, tales como distinguir adecuadamente el tiempo del evento y el tiempo de procesamiento, también soporta ventanas y posee una administración

simple pero eficiente del estado de la aplicación. Se lo puede integrar a una aplicación debido a que es una biblioteca ligera.

2.3 Clústeres: arquitectura y componentes

Como hemos observado, los framework de Big Data requieren de un alojamiento en un clúster para su potencial funcionamiento. Los clústeres poseen distintas connotaciones que se establecen de acuerdo al uso que se le presente o los servicios que ofrezcan, por lo que se le clasifica de la siguiente forma:

- Alta productividad: Este tipo de clúster es dedicado para trabajos independientes en donde cada trabajo es ocupado por un sola máquina o nodo, así se tiene que su latencia no es un punto critico y el Número de trabajos por unidad de tiempo.

- Alta prestaciones: Este tipo de clúster esta enfocado para aplicaciones que se ejecutan en instancias paralelas por lo que su primordial objetivo es reducir el tiempo de efectuar la tarea de dicha aplicación.

- Alta disponibilidad: Este clúster se encamina a la redundancia del almacenamiento de los datos y así se tiene una facilitación de sustitución de los datos en caso de fallo, es decir su objetivo es un funcionamiento continuo.

Además, un clúster para su correcta operación posee una gama de elementos cuyo objetivo es evitar la interferencia entre las operaciones de calculo o de entrada y salida (E/S) y promover una mayor eficiencia a la disponibilidad y seguridad, por lo que nos encontraremos en un clúster con:

- Nodo Maestro: Son nodos que son usados para facilitar al usuario el acceso a los recursos del cómputo, en donde se planifican las tareas que se pretende realizar, así como el espacio para su almacenamiento. Además, esconde los recursos, y da una visualización de un solo recurso computacional.

- Nodo de Cómputo: Son nodos que realizan los cálculos u operaciones asignadas proporcionalmente de las distintas aplicaciones que se ejecutan de forma paralela, y si se menciona de una disponibilidad se lo trata como una unidad de servicio que tiende a ser escalable.

- Nodo Administrativo: Son nodos que dan servicios administrativos, como un monitoreo del rendimiento y crear eventos para el clúster.
- Nodo de Infraestructura: Son aquellos nodos que proveen servicios primordiales para el clúster, como servicios de licenciamiento, de autenticación, planificación de tareas y en muchos casos de balance de carga.
- Nodo de Servidor de Archivos E/S: Son nodos, que permite el acceso a los recursos de almacenamiento que posee el clúster para los usuarios y las aplicaciones.
- Redes: Es el medio de comunicación por el cual interactúan para la gestión y control de los nodos, la transferencia de datos que presentan.

3. Arquitecturas de Big Data

3.1 Introducción

Las tendencias tecnologías se presentan a un ritmo vertiginoso y cada día se expande a un ritmo exponencial, abriendo las puertas hacia nuevos enfoques de entendimiento de la información generada por el usuario y las distintas decisiones que estas puede conllevar. Para ello describimos como una enorme cantidad de datos sean tomaría demasiado tiempo y a su vez sería muy costoso cargar a una base de datos relacional para su respectivo análisis.

Ahí es cuando se requiere de una arquitectura de Big Data que pueda procesar y/o analizar los petabytes y/o los exabytes de datos, mediante el uso de herramientas o frameworks. Este inmenso volumen de información esta representado de diferentes maneras como datos de dispositivos de audio, video, Sistemas GPS, Dispositivos móviles, automóviles, etc.

Toda esta variabilidad de información necesita que la velocidad de respuesta por parte de las aplicaciones que las analizan, sean lo más rápido y así obtener la información correcta en su momento oportuno.

Los tipos de datos que se exploran para un Big Data depende de muchas organizaciones que se cuestionan sobre qué información presentan para analizar y que es lo que se desea resolver, para ello se podría entender como una clasificación de cinco grupos, con la probabilidad que este se pueda extender a medida que existan los múltiples avances tecnológicos.

Dentro de la primera clase podríamos mencionar al contenido existente en la web e información que es muchas ocasiones obtenida en redes sociales como LinkedIn, Twitter, Facebook, Instagram, etc. Una segunda clase tenemos a aquellas tecnologías que permiten conectarse a otros dispositivos por medio de sensores o mediadores que capturan algún evento en particular ya sea la temperatura, presión atmosférica, velocidad, variables químicas presentes en sustancias como la salinidad, etc., dado a que se transmiten por medio de redes inalámbricas, alámbricas o híbridas hacia otro dispositivo que traduce ese conjunto de datos en una información significativa para el usuario. Como un tercer grupo tenemos a los grandes datos transaccionales que incluyen registros de facturación tanto en telecomunicaciones, debido a que esos datos transaccionales siempre están disponibles en formatos que no poseen

estructura o alguna estructura presente. En un cuarto grupo obtenemos información biométrica que incluye lectura de la retina, reconocimiento facial, huellas dactilares e incluso reconocimiento genético, debido a que este tipo de datos han sido información importante para las agencias de investigación. Y por último tenemos a los datos generados por humanos como por ejemplo la información que se almacena en un “call center” para llegar a establecer una llamada telefónica, correos electrónicos, documentos electrónicos, estudios de diferentes carreras, notas de voz, etc.

A continuación, se presentará la arquitectura recomendada para solventar el objetivo principal del proyecto.

3.2 Arquitecturas

Varias organizaciones remedian la problemática de grandes volúmenes de datos desde diferentes ángulos que conducen a preguntas como i) el diseño de entornos escalables, ii) proporcionar una tolerancia a fallos, y iii) diseñar soluciones eficientes. Así los problemas de Big data comparten cuatro características, conocidas como las cuatro V acorde a los autores [1]:

- Volumen: Se da a entender como el tamaño de los conjuntos de los datos que normalmente requieren ser almacenados de forma distribuida.
- Variedad: Entra en el contexto que un Big Data se compone de varios tipos diferentes de datos como sonido, texto, imagen y video.
- Veracidad: Se refiere a los sesgos, ruido y anormalidad de los datos.
- Velocidad: Se ocupa del ritmo al que fluyen los datos desde diversas fuentes como dispositivos móviles, Internet de las Cosas (IoT), redes sociales, etc.

Como se describió en el estado del arte, existen muchos framework para diferentes tareas pensadas en Big Data. Así tenemos que Apache Hadoop es la base de todas aquellos framework, en la cual prestaremos interés en un componente de Hadoop denominado Sistema de Archivos Distribuidos de Hadoop (HDFS), la cual describiremos a detalle más adelante. Sin embargo, el framework que se escoge para mitigar el problema presentado es Apache Spark, debido a que se necesita procesos en tiempo real y su velocidad de procesamiento es superior a otras framework.

Ambos son diseños tecnológicos pensados con la idea de operar en un mismo

contexto y procesar una vasta cantidad de datos. Así tenemos dos arquitecturas que son difíciles de comparar debido a que ambas realizan en muchas ocasiones cosas similares [2]. Para ello cabe resaltar que la arquitectura Spark no tiene su propia administración de archivos por lo que debe apoyarse del que posee el framework Hadoop con su HDFS.

Debido a eso lo más indicado sería una comparación de Spark con Hadoop MapReduce, ya que son en sí, más comparables a carácter de motores de procesamientos de datos.

Lo que Spark tiene a su favor es su velocidad y facilidad de uso. Además, ambas arquitecturas son compatibles entre sí, dando como una solución extremadamente eficaz en variedad de aplicaciones dentro del Big Data.

3.2.1 Hadoop

La arquitectura Hadoop [3] es de código abierto y provee soluciones para manejar Big Data junto con un extenso procesamiento de análisis, permitiendo a los desarrolladores procesar una gran cantidad de datos mientras oculta la complejidad de la ejecución paralela en cientos de servidores en un entorno de la nube.

Este framework esta escrito en Java, lo que permite el procesamiento de grandes cantidades de datos a través del clúster usando modelos de programación simple.

Así, hablando de esta arquitectura, nos centramos primordialmente en el Sistema de Archivos Distribuidos de Hadoop (HDFS) y MapReduce. Estos dos componentes son los más importantes de la arquitectura Hadoop.

El HDFS está inspirado en el Sistema de Archivos de Google (GFS) y proporciona un almacenamiento de datos escalable, eficiente y con réplicas en varios nodos que forma parte de un clúster.

Esta arquitectura es ampliamente conocido y utilizado por empresas como IBM, EBay, y Facebook que les permite realizar búsquedas, procesos de registros, análisis de video e imágenes.

Un detalle de esta arquitectura es su manejo en diferente tipos de datos almacenados en cualquier arquitectura, así se puede llegar a mezclar datos

estructurados, semiestructurados y no estructurados [4].

3.2.1.1 Sistema de Archivos Distribuidos Hadoop (HDFS)

El HDFS [3] es un sistema de archivos distribuidos que se puede almacenar datos en varios servidores, de formar paralela el acceso y su respectiva tolerancia a fallos.

Una de las diferencia que se tiene este HDFS frente a los otros sistemas de archivos distribuidos, que está diseñado para ser construido a partir de un componente básico de bajo costo que requiere que sea altamente tolerante a fallos.

Este sistema de archivos distribuido esta escrito en Java, y se ejecuta sobre el sistema de archivos que usualmente se usa en un equipo computacional y esta desarrollado para admitir aplicaciones con grandes cantidades de volúmenes de datos en el orden de terabytes y petabytes.

El funcionamiento que tiene este sistema es dividir un tamaño de bloque típico de 64 MB, por lo que es posible almacenar cada fragmento en un nodo diferente otorgando un mayor almacenamiento de archivos, en comparación de un NTFS que es un bloque de 4 KB siendo ineficiente.

La arquitectura que tiene el HDFS es maestro-esclavo. El clúster HDFS consta de un demonio llamado “NameNode” que le pertenece al nodo maestro para administrar el “namespace” del sistema de archivos distribuidos y saber dónde están los bloques de datos almacenados dentro del clúster, además de regular el acceso a los archivo por parte de los clientes.

En los nodos esclavos se almacenan los datos reales y proporciona la potencia de procesamiento para ejecutar las tareas. Consta de un demonio llamado “DataNodes”, que administra el almacenamiento físico de los datos reales de los nodos, como se puede observar en la Figure 1.

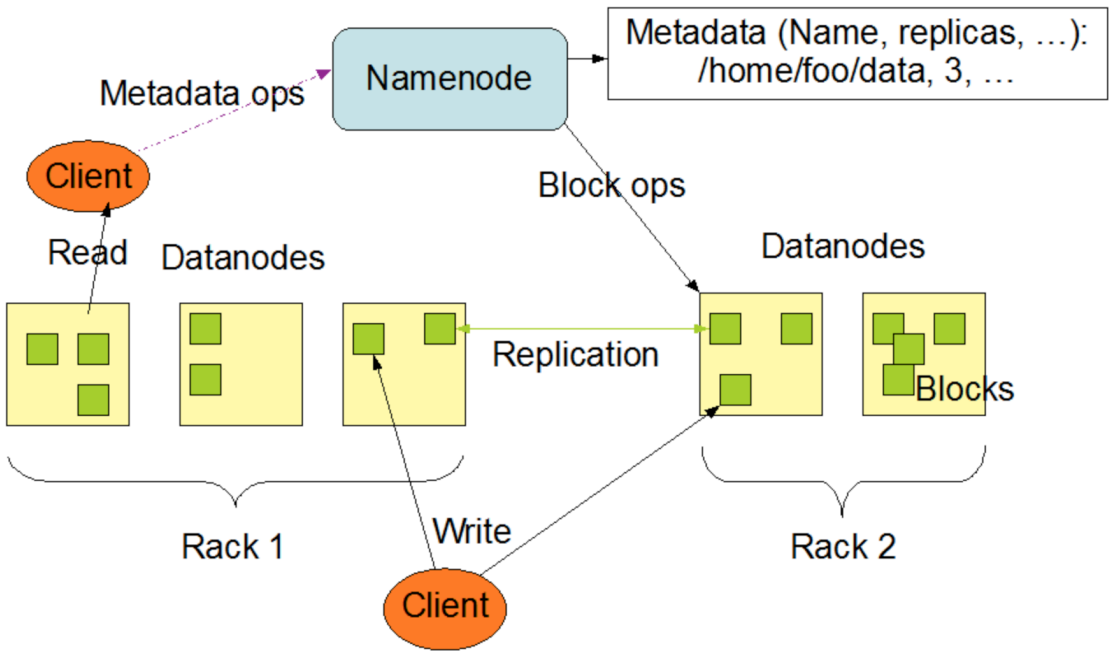


Figure 1. Arquitectura HDFS [3]

3.2.1.2 MapReduce

Es una técnica de procesamiento y un modelo de programación para la computación distribuida realizado en Java, que ha sido propuesta por Google como un nuevo modelo de programación para procesar y analizar grandes cantidades de datos.

Su función primordial es dividir y distribuir la carga de trabajo para aumentar la velocidad de cálculo. Esto permite llevar a cabo dos etapas distintas; la operación “Map”, que es escrita por el usuario en donde su objetivo es producir una clave o valor intermedio para cada par recibido en la entrada. Luego, la biblioteca MapReduce agrupa todas las claves intermedias asociadas con el mismo valor y las envía a la operación “Reduce”.

Como último paso, es fusionar los valores de la misma clave intermedia para crear un valor único asociado con esa clave y devolver una sola clave como salida para cada clave intermedia.

Después del procesamiento mencionado, reúne todos los resultados en un resultado final que será almacenado en el HDFS. Así mismo, MapReduce consta de un servidor maestro al cual se lo denomina JobTracker, que es el responsable de asignar tareas, y una cantidad de servidores esclavos llamados TaskTracker que son los responsables de ejecutar las tareas [3].

Esta técnica brinda una gran flexibilidad para administrar grandes base de datos distribuidas. Otra ventaja no es necesario poseer grandes capacidades de procesamiento para administrar, almacenar, aislar, y tratar en un tiempo relativamente razonable esa cantidad de datos. Por lo que nos brinda la principal ventaja de reducir fácilmente el procesamiento de datos en múltiples nodos dentro de un clúster, como se puede observar en la Figura 2.

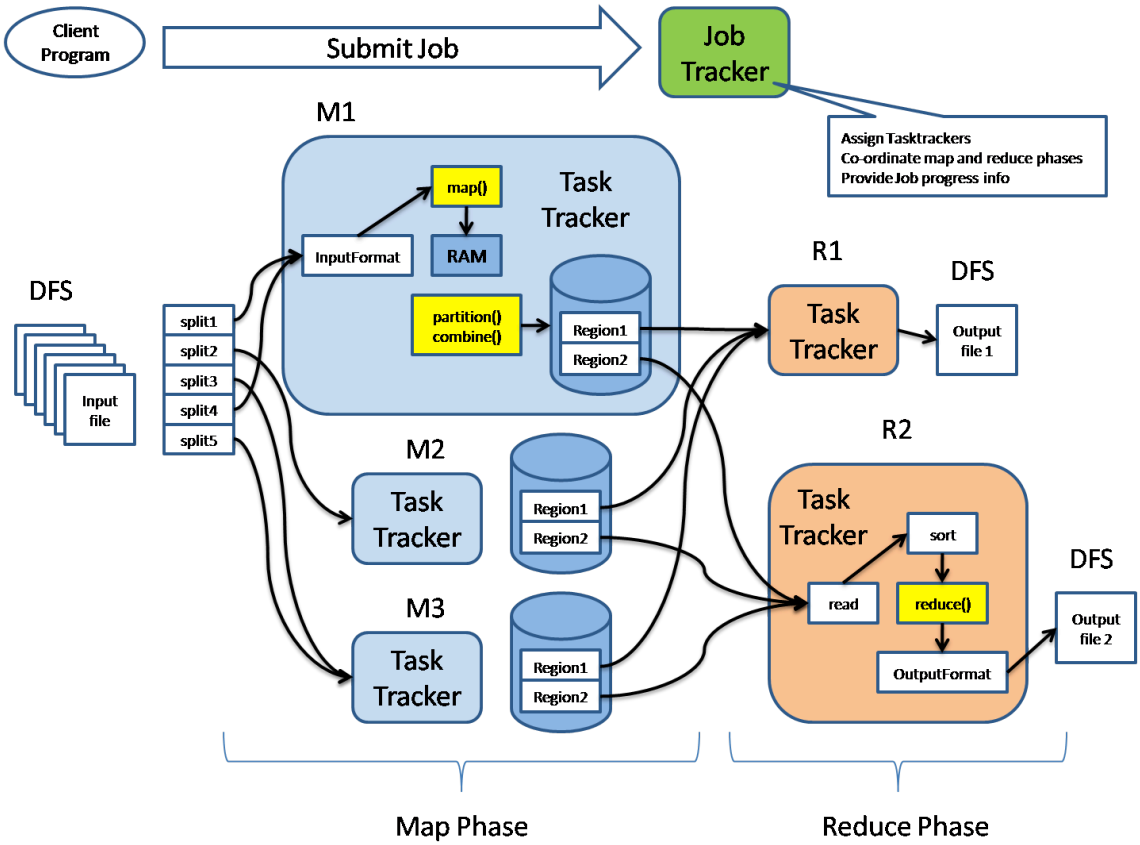


Figura 2. Arquitectura de MapReduce [3]

3.2.2 Spark

Este framework es de código abierto para el procesamiento de Big Data, que permite a los usuarios ejecutar aplicaciones de análisis de datos a gran escala dentro de un clúster. Siendo este framework uno de los subproyectos de Hadoop desarrollado en 2009 en el AMPLab de UC Berkeley [5].

Así, Spark posee varias ventajas en comparación a otras tecnologías de Big Data del ecosistema de Hadoop como MapReduce. Primero, ofrece un completo y unificado framework para direccionar a las necesidades del tratamiento de datos. Spark esta desarrollado en Scala que es programación funcional adecuada para sistemas distribuidos. Gracias a ello, Spark permite

que las aplicaciones en los clúster de Hadoop funcionen unas 100 veces más rápido en la memoria, y unas 10 veces más rápido en el disco de almacenamiento.

Esto llega a ser posible debido a la reducción del número de operaciones de lectura y escritura del disco donde se almacena los datos intermedios de procesamiento en la memoria del clúster, como se puede observar en la Figura 3, donde se expresa el tiempo de ejecución al aplicar una regresión logística tanto en Hadoop como en Spark.

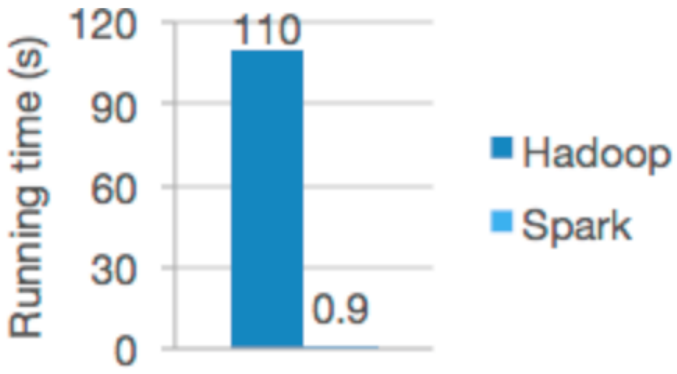


Figura 3. Tiempo de ejecución de una regresión logística [5]

Según los autores [6], llegan a la misma conclusión demostrando el menor tiempo de ejecución al usar Spark, debido a que realizan una prueba de ingesta de datos a Spark y a Hadoop por medio de una suite de Benchmark de aplicaciones del mundo real, llegándose a observar el tiempo de ejecución de cada uno de estos framework. Este benchmark esta desarrollado por Intel con la licencia de Apache, Así observamos en la Figura 4 que nos da los tiempos de ejecución de los benchmark aplicados en los dos framework.

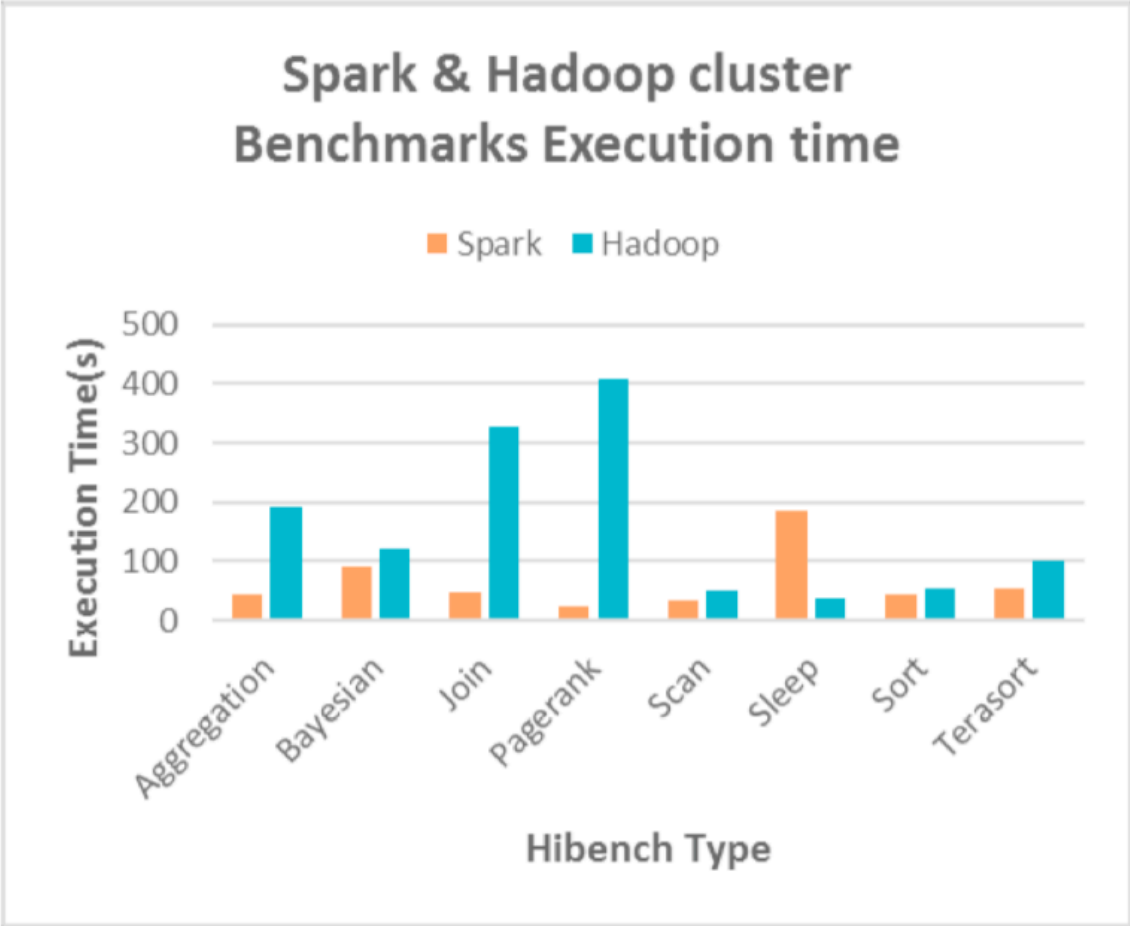


Figura 4. Tiempos de ejecución entre Spark y Hadoop [6]

A parte de de las operaciones de “Map” y “Reduce”, Spark incorpora consultas SQL, funciones de transmisión de datos, Machine Learning y algoritmos de grafico. Los desarrolladores pueden usar estas funciones en forma independiente o dentro de una cadena de procesamiento complejo.

Así, el “Resilient Distributed Dataset” (RDD) [3] llega a ser una colección particionada de registros de solo lectura que solo puede crearse mediante operaciones deterministas, y la abstracción principal para Spark.

RDD se lo puede observar como una tabla en una base de datos, y los datos se los puede tratar como cualquier otro tipo de datos ya que Spark lo almacena en diferentes particiones, permitiendo reorganizar los cálculos y optimizar su tratamiento de los datos. RDD es tolerante a fallos de datos, debido a que sabe como recrear y recalcular el conjunto de datos.

Como podemos observar en la Figura 5, esta representando los componentes que contiene Spark, los cuales mencionaremos a continuación [5]:

- Spark SQL: Puede mostrar conjuntos de datos de Spark por medio de la JDBC API y ejecutar las consultas convenientes del tipo SQL utilizando herramientas empresariales inteligentes y con su visualización tradicional.
- Spark Streaming: Se puede usar para el procesamiento de flujo de datos en tiempo real.
- Spark MLlib: MLlib es un framework de Machine Learning distribuido en Spark, que proporciona múltiples tipos de algoritmos de aprendizaje automatizados que incluyen clasificaciones, regresiones, clustering y filtrado colaborativo, y también proporciona funcionalidades como evaluación de modelos e importación de datos.
- GraphX: Es una nueva API para el tratamiento de gráficos que incluye la ejecución en paralelo.

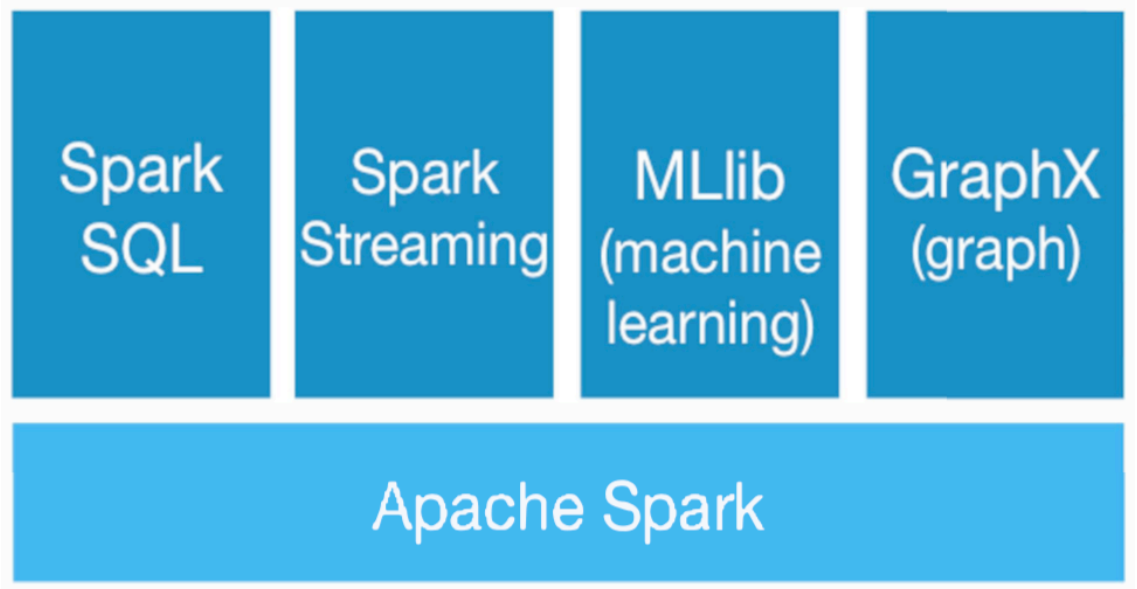


Figura 5. Componentes de Spark [5]

3.2.3 Apache Spark frente a otras tecnologías similares

Debido al gran conjunto de tecnologías que ofrece el ecosistema de Hadoop, hay discernir cual es el mejor para cumplir con el objetivo del proyecto. De este modo, detallando sus ventajas e inconvenientes podemos resaltar tres tecnologías del ecosistema de Hadoop: Apacha Kafka, Apache Flume, Apache Spark.

Apache Flume: es un framework ideal para pequeños proyectos de procesamientos por lotes (Batch) y casi en tiempo real, que toma datos de un punto a otro con procesamiento local, es decir no de enriquecimiento externo. El filtrado, la transformación y los múltiples destinos de empuje son motivos comunes para Flume. Este framework no es tan bueno usarlo si sus datos necesitan un enriquecimiento externo como tomar datos de base de datos externas o de servicios web, debido a que las transacciones y micro lotes pueden llevar a un reprocesamiento y depende de la aplicación para evitar duplicados.

- Ventajas presentes en Flume: i) múltiples fuente de datos (orígenes) y destinos (sumideros) que le permite mover datos a cualquier almacenamiento de datos; ii) facilidad de configuración y ejecución, iii) facilidad para filtrar y transformar datos por medio del uso de interceptores.

- Inconvenientes en Flume: i) resulta difícil realizar escalabilidad debido a que se debe cambiar la topología pipeline y añadir un nuevo destino; ii) el soporte de canales efímeros basados en memoria y canales que perduran basados en archivos, dado a la importancia de la durabilidad del mensaje puede que no este disponible en el destino hasta que se recupere el agente; iii) la falta de réplica de datos en un nodo diferente al usar el canal basado en archivos y se recomienda utilizar Nombre Alternativo del Sujeto (SAN) o RAID; iv) no es recomendado para el procesamiento complejo de eventos; v) presenta incompatibilidades con las últimas versiones y vi) dependencia de software de administración de terceros como Hortonworks o Cloudera.

Apache Kafka: es un framework de mensajería distribuida de alto rendimiento en el que varios productores envían datos a un clúster de Kafka y

a su vez los sirve a los consumidores. Llegando a ser adecuado para escenarios casi en tiempo real en proyectos de alto volumen de datos.

- Ventajas en Kafka: i) diferentes grupos de consumidores pueden absorber mensajes a diferentes ritmos; ii) la escalabilidad se ve reflejada en la capacidad de manejar picos de eventos con la flexibilidad de escenarios de código abierto; iii) ideal para proyectos de baja latencia.

- Inconvenientes en Kafka: i) no proporciona soporte nativo para el procesamiento de mensajes por lo que es necesario que se integre a herramientas como Spark Streaming para completar su trabajo; ii) requiere de muchas actualizaciones al año al traer nuevas características; iii) no hay un respaldo por parte de las compañías comerciales; iv) es necesario crearse un propio consumidor, que en otras arquitecturas del ecosistema de Hadoop ya están integradas.

Apache Spark: Es un framework excelente como procesador de flujo de datos a gran escala, que admite múltiples lenguajes de programación. Es también excelente para su uso en tareas de machine learning especialmente para grandes conjuntos de datos que se pueden procesar en paralelo, dividiendo los archivos. Spark Streaming es escalable para el proceso de flujo de datos en tiempo real y provee un alto nivel de abstracción llamado “discretized stream” (DStream) que representa un continuo flujo de datos como se muestra en la Figure 6.

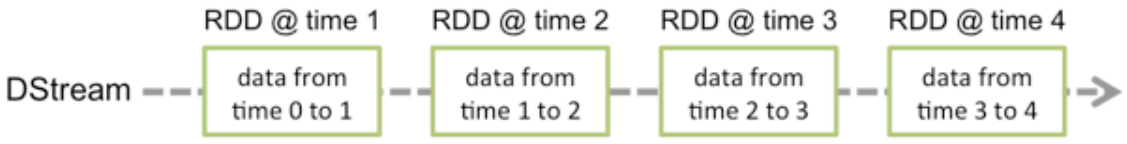


Figure 6. Flujo de datos con Spark Streaming

El DStream puede crear flujos de datos de entrada desde diferente orígenes como Kafka, Flume, Kenesis, o aplicando un alto nivel de operaciones en otro DStream. Cuando se maneja un subconjunto de datos mas pequeños, Spark no llega a ser tan eficiente. Asi, podemos observar en la Figura 7 el flujo de entrada y salida utilizando Spark Streaming.

Figura 7. Arquitectura Spark Streaming

- Ventajas en Spark: i) posee paralelismo simplificado, dado a que Spark Streaming crea tantas particiones del RDD como sea necesaria y cada una trabajando en un nodo worker por lo que se leerán datos de Kafka en paralelo; ii) alta escalabilidad y transparencia para la aplicación; iii) procesamiento complejo de mensajes con funciones de alto nivel; iv) mayor rendimiento que otras tecnologías de procesamiento de datos distribuidos como MapReduce, Hive, Pig; v) más rápido en el procesamiento de datos que Hadoop MapReduce.

- Inconvenientes en Spark: i) muy débil en la gestión de memoria; ii) PySpark no es tan robusto como Scala con Spark; iii) uso avanzado de flujo de datos por lo que se requiere de conocimientos avanzados; iv) para una alta disponibilidad es necesario de un Spark Master.

Una vez presentada las distintas características de las tres arquitecturas similares del ecosistema de Hadoop, se usará Apache Spark debido a sus ventajas como la escalabilidad, alto rendimiento, y en especial para realizar procesos en tiempo real optando por su módulo Spark Streaming debido a su rapidez en el procesamiento de grandes cantidades de datos cuando hay que escribir en base de datos o construir modelos de análisis de datos, y cumplir con el objetivo principal del trabajo de ingesta datos procesados. Además, se requerirá de Hadoop HDFS, para almacenar dicha ingesta de datos de forma distribuida por el clúster virtual. Esto nos garantiza una menor latencia en la ingesta de los datos que sean.

4. Simulación de entorno de Big Data

4.1 Situación actual

Como se ha mencionado con anterioridad, las arquitecturas de Big Data se lo puede usar en varios campos. Para este trabajo nos centraremos en un escenario de 3000 notarías aproximadamente que existen en España.

Para ello, se debe cubrir cualquier necesitada tecnológica, resolviendo las problemáticas de las notarías. Uno de esos problemas es la digitalización de los protocolos de las notarías, lo cual deben debe ser de un modo seguro y confiable. Motivo por el cual cada notaría tiene un servidor que envía información constantemente hacia los otros servidores, dándose a ocasionar accesos legítimos o ilegítimos que pueden poner en riesgo la información vital que se tienen en los servidores.

La cantidad de información es muy elevada para procesarla de una forma tradicional y que tenga una respuesta en tiempo real, por lo que es necesario una arquitectura de Big Data para el procesamiento de grandes volúmenes de datos que son generados por las 3,000 notarías realizando una inyección en tiempo real con su respectivo almacenamiento.

De este modo, se ha elegido a Apache Spark conjuntamente con Apache Hadoop HDFS, debido a sus ventajas del framework Spark como la rapidez en el procesamiento de datos y su respaldo en la integración en grandes compañías como IBM, Amazon, Microsoft y Google. Hadoop HDFS es escogido debido a su eficiencia en el almacenamiento distribuido que complementaria a Spark en un clúster.

4.2 Objetivos

Los objetivos que se persigue en la simulación del entorno de Big Data para el escenario de las notarías son:

- Crear un clúster virtual con un respectivo nodo maestro, nodos esclavos, y un nodo servidor de archivos (NFS).
- Generar datos sintéticos de forma automática con las distintas características ofrecidas.
- Implementar Apache Spark conjuntamente el HDFS de Apache Hadoop

en el clúster virtual.

- Crear un programa en Python para inyectar datos sintéticos en tiempo real con Spark Streaming y que se almacenen en el clúster virtual con NFS o HDFS.

4.3 Entorno virtual del clúster Big Data

Dado el escenario presentado, se debe elaborar un clúster virtual para desarrollar la inyección de datos e implementar la arquitectura de Big Data.

Como software de virtualización se usa Oracle VirtualBox versión 5.2.12 para arquitecturas x86 y amd64. El sistema operativo que se va a manejar en este entorno es Ubuntu en su versión 14.04LTS.

Como podemos observar en la Figura 8, para un clúster es necesario de un nodo maestro en el cual va a estar alojando Spark en modo Maestro, en los nodos esclavos serán los que alojen a Spark en modo trabajador (worker) o esclavo (slave), así para un modo de alta disponibilidad se incluye un nodo standby que no es más que otro nodo maestro, cuando falla el nodo maestro principal, y el nodo standby se activa de forma inmediata.

De este modo tendríamos un entorno de simulación para realizar las pruebas correspondientes, en la que se podrá corregir errores antes de llevarlo al clúster real.

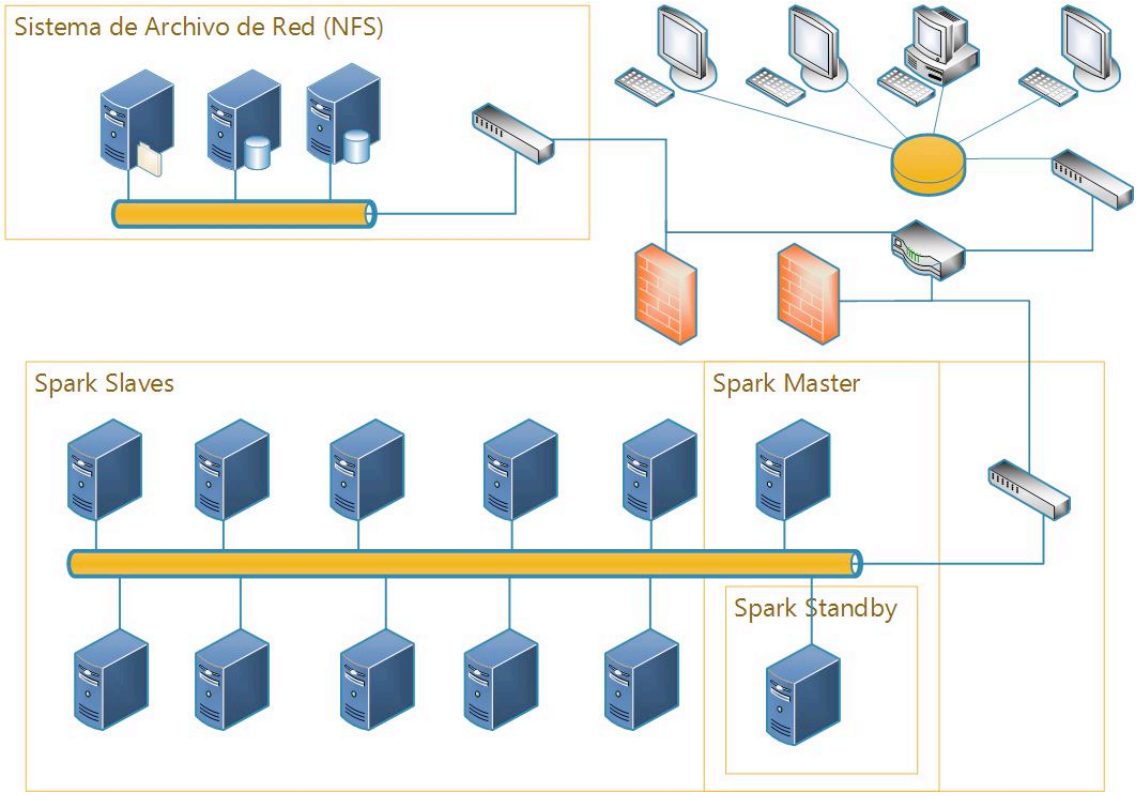


Figura 8. Virtualización de Clúster Big Data.

4.4 Generación de Datos sintéticos

Para la generación de datos sintéticos, se creó un programa basado en Python versión 2.7.6. Para esta creación de datos, toma las características principales detallados en la Tabla 1, las últimas cuatro características son las medidas de seguridad para los datos.

En la característica de “attack” va a tomar los valores de manera aleatoria de la Tabla 2, que son las características de detectores de intrusiones.

También se incluye la dirección IP de la máquina que se esta enviando y recibiendo la información. Debido a que si es una IP desconocida se pueda descargar dicho dato por medio de un procesamiento con Spark Streaming como filtro y no almacenar dicha información.

Y la última característica implementada en este conjunto de datos sintéticos generados es la creación de una contraseña encriptada, que será la misma para quien envíe y reciba los datos, en el que contendrá una “salt” mas un Algoritmo Hash Seguro (SHA-256) de 256 bits. Así, de forma aleatoria el programa escoge si crear una contraseña legitima o ilegitima, para poder

simular que un dato ha sido detectado como intruso y poder procesar dicha información, elevando el nivel de seguridad de datos sensibles.

Las características de los datos sintéticos van a ser generadas de forma aleatoria en todos sus campos, para tener una variabilidad alternante en los datos, dentro de un archivo con Valores Separados por Comas (CSV). La cantidad de datos en cada archivo será de alrededor de 100.000. En vista que es una pequeña cantidad de datos generada en un solo archivo, se ve la necesidad de crear iterativamente al menos 200-300 archivos, para tener una cantidad sustentable de datos sintéticos.

En la Figura 9 puede observarse, es un ejemplo de una pequeña fracción de un archivo generado con datos sintéticos. Dicho archivo contiene una cabecera de descripción de las características en la primera línea, y el cuerpo del archivo esta conformado por las demás líneas con el valor de las características creadas para este escenario.

Características	Descripción
duration	Número de segundos de la conexión
protocol_type	Tipo de protocolo, ejemplo: tcp, udp, etc
service	Servicio de red en el destino, ejemplo: http, telnet, etc
src_bytes	Número de datos en bytes del origen al destino
dst_bytes	Número de datos en bytes del destino al origen
flag	Estado de normal o error de la conexión
land	1= si la conexión es desde/hacia el mismo puerto/host; 0= otros
wrong_fragment	Número de fragmentos erróneos
urgent	Número de paquetes urgentes
attack	Tipo de ataque realizado, o “none” si no se realizó
sourceIP	Dirección IP del origen de los datos
targetIP	Dirección IP del destino de los datos
pass	Contraseña criptográficamente segura

Tabla 1. Característica básica de datos.

Ataque	Descripción
dos	denial-of-service (denegación de servicio), ejemplo syn flood
r2l	Acceso no autorizado de una máquina remota
u2r	Acceso no autorizado a una máquina local con super privilegios ejemplo: varios ataques “buffer overflow”
probing	Vigilancia y otras pruebas, ejemplo: escaneo de puertos

Tabla 2. Característica de detectores de intrusiones.

Figura 9. Ejemplo de un archivo CSV con datos sintéticos generados.

El código fuente del programa generador de datos sintéticos está adjuntado en la sección de Anexos con su correspondiente documentación.

4.5 Implementación de Apache Spark con HDFS en clúster

Una vez se tenga montado completamente el clúster virtual, procedemos a implementar los framework elegidos. Para ello es necesario tener instalado Java de antemano, para este clúster utilizaremos la versión 7. Además, tendremos que instalar tres paquetes “openssh-server”, “openssh-client” y “scala”, por medio del siguiente comando en todos los nodos:

```
sudo apt-gpaet install -y oracle-java7-installer openssh-server openssh-client scala
```

Para agregar el nombre de los nodos con el que se conocerán dentro del entorno del clúster es necesario modificar un archivo << /etc/hosts >> como se muestra en la Figura 10, en donde colocaremos la dirección IP con un nombre designado por el usuario, en este caso será: sparkMaster, sparkSlave02, sparkSlave03, y sparkNFS. Después copiaremos este archivo a todos los nodos del clúster.

Luego procedemos a realizar la respectiva configuración del programa SSH para poder acceder a los diferentes nodos del clúster sin colocar una contraseña, y permitir a los framework comunicarse entre si. Para ello lo realizamos con el siguiente comando:

```
ssh-keygen –t rsa –P “”
```

Lo que nos genera un archivo en << .ssh/id_rsa.pub >> en el master, y copiamos su contenido en un archivo creado denominado << .ssh/authorized_keys >> y distribuir este archivo creado a todos los nodos del clúster, usando los siguientes comandos:

```
cp .ssh/id_rsa.pub .ssh/authotized_keys
```

```
scp .ssh/authotized_keys sparkSlave02:.ssh/authotized_keys; scp .ssh/authotized_keys sparkSlave03:.ssh/authotized_keys; scp .ssh/authotized_keys sparkSlaveNFS:.ssh/authotized_keys
```

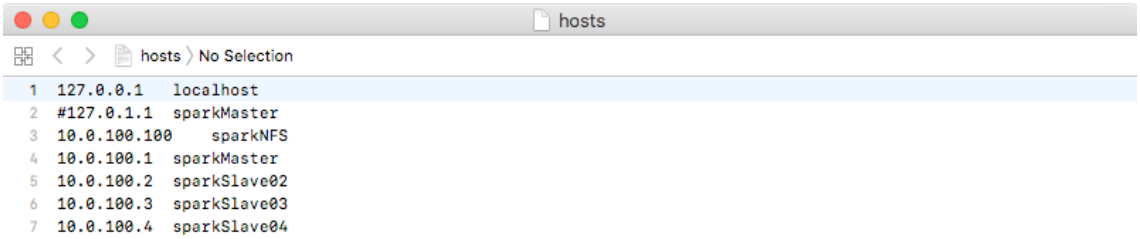


Figura 10. Archivo Hosts

De este modo tendremos listo el clúster virtual para implementar los framework de Spark y Hadoop.

La versión que se implementa en este clúster virtual es 2.1.0 con compatibilidad con Hadoop 2.6. Para el uso del HDFS, usaremos Hadoop con la versión 2.6, para que sean compatibles tanto Spark como el HDFS .

En este caso, usando el clúster virtual, he creado dos usuarios: “spark” y “hadoop”; con el objetivo de mantener ambas arquitecturas en diferentes partes del clúster.

Para instalar Apache Spark, tendremos que descargarnos desde la pagina oficial, y descomprimirlo. Para ello usamos la siguiente línea de comando que nos facilitará en dicho proceso:

```
wget https://archive.apache.org/dist/spark/spark-2.1.0/spark-2.1.0-bin-hadoop2.6.tgz; tar xzf spark-2.1.0-bin-hadoop2.6.tgz
```

La configuración del Spark comienza con editar el archivo << .bashrc >> que se encuentra localizado en el directorio “home” del usuario y colocar las variables de entorno tanto de Java, como de Spark así como se muestra en la Figura 11, esto nos da la disponibilidad de usar los ejecutables de Spark.

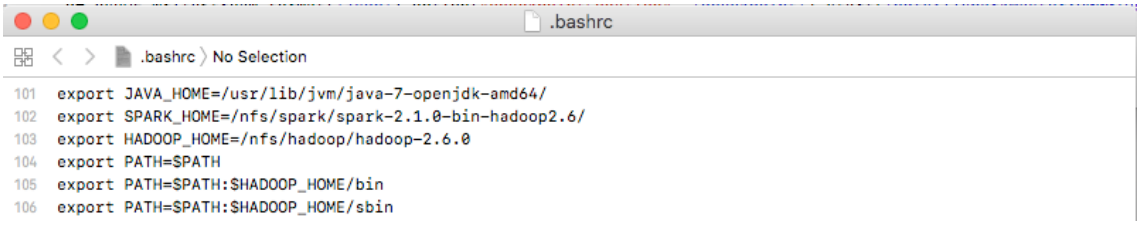


Figura 11. Configuración del archivo de entornos

Otras configuraciones que se debe realizar para Spark están localizados en << $SPARK_HOME/conf >> son: i) el archivo llamado “spark-env.sh”, que usa Spark para saber donde se encuentra Java y la cantidad de núcleos del procesador que va a utilizar cada nodo esclavo del clúster como se muestran en la Figura 12; y ii) el archivo llamado “slaves” se colocan los nombres de los nodos esclavos como en la Figura 13.

Figura 12. Archivo de entorno de Spark

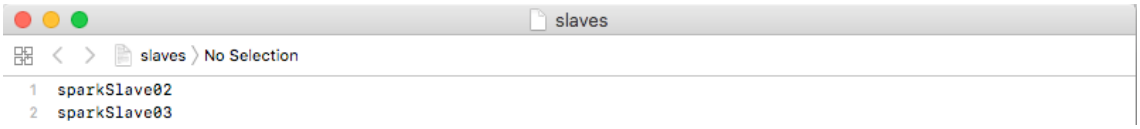


Figura 13. Archivo de esclavos de Spark

Para la distribución de la configuración de Apache Spark por los nodos esclavos no va a ser necesario realizar una réplica de la carpeta Spark, debido a que esta en marcha el Sistema de Archivos de Red (NFS). De este modo, se tiene instalado en el clúster virtual Apache Spark.

Para la implementación de Hadoop HDFS en el clúster, nos ubicamos en la carpeta raíz del usuario creado “hadoop”.

Para instalar Apache Hadoop en el clúster virtual, lo que realizamos es descargarlo desde la página oficial y descomprimirlo con el siguiente comando:

```
wget http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-2.6.5/hadoop-2.6.5-src.tar.gz; tar xzf hadoop-2.6.5-src.tar.gz
```

De igual manera que Spark, se debe colocar las variables de entorno del Apache Hadoop como se muestra en la Figura 11.

La configuración que se debe dar a Apache Hadoop se encuentra en la dirección <<$HADOOP_HOME/etc/hadoop>>, y modificar cuatro archivos.

El primero archivo es “hadoop-env.sh” y donde se debe colocar la dirección completa de la máquina virtual de Java, como se muestra en Figura 14.

Figura 14. Archivo de configuración del entorno de Hadoop

El segundo archivo de configuración para Hadoop es “core-site.xml”, se debe colocar la dirección del HDFS. Este valor está compuesto por la dirección IP o el nombre del nodo maestro conjuntamente con su puerto, así como en la Figure 15.

Figure 15. Archivo de localización del nodo Maestro

El tercer archivo de configuración para el HDFS es “hdfs-site.xml”, el principal componente que podemos modificar es la última propiedad de replicación. Esta propiedad indica cuantas veces los datos son replicados en el clúster a través de sus nodos.

Hay que tener en cuenta que ese valor debe ser menor o igual al número de nodos esclavos que se tiene en el clúster, en nuestro caso no aplicaremos dicha propiedad, y dejaremos el valor por defecto que es 1, como se muestra en la Figura 16.

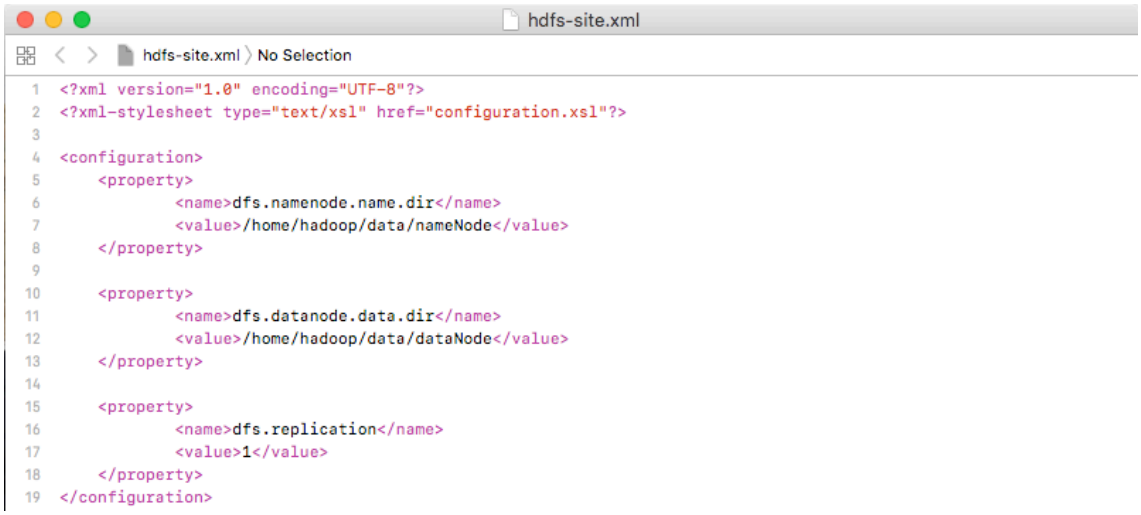


Figura 16. Archivo de ubicación del HDFS y réplicas

Y el último archivo a modificar es “slaves”, el cual Hadoop debe conocer cuáles son los nodos esclavos dentro del clúster. Para ello colocaremos el nombre de dichos nodos, como se observa en la Figura 17.

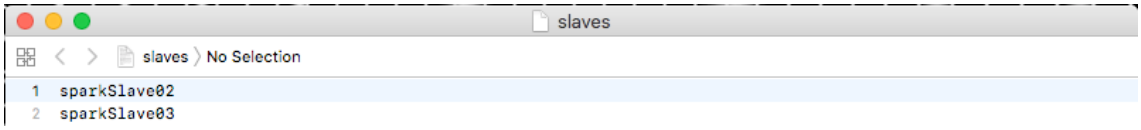


Figura 17. Archivo de configuración de nodos esclavos

Como último paso para la configuración de Apache Hadoop, el HDFS requiere estar formateado como cualquier sistema clásico de archivo. Para lograrlo, en el nodo maestro hay que ejecutar el siguiente comando:

```
hdfs namenode -format
```

Con estos pasos queda concluida la instalación y configuración de ambas tecnologías sobre el clúster virtual.

Para inicializar Spark, se debe ejecutar el siguiente comando:

```
sh /nfs/spark/spark-2.1.0-bin-hadoop2.6/sbin/start-all.sh
```

Para inicializar Hadoop, se tiene el siguiente comando:

```
sh /nfs/hadoop/ Hadoop-2.6.5/sbin/start-dfs.sh
```

Para ahorrar el tiempo en dar la ubicación completa de ambas tecnologías tanto para inicializar como para detener el servicio que ofrecen en el clúster, he creado unos script que se presentan en el Anexo.

Para conocer si esta en ejecución ambas tecnologías, Spark como Hadoop HDFS, simplemente hay que ejecutar el comando << jps >> y nos mostrara los demonios que están ejecutándose en background en los nodos del clúster, como se muestra en la Figura 18.

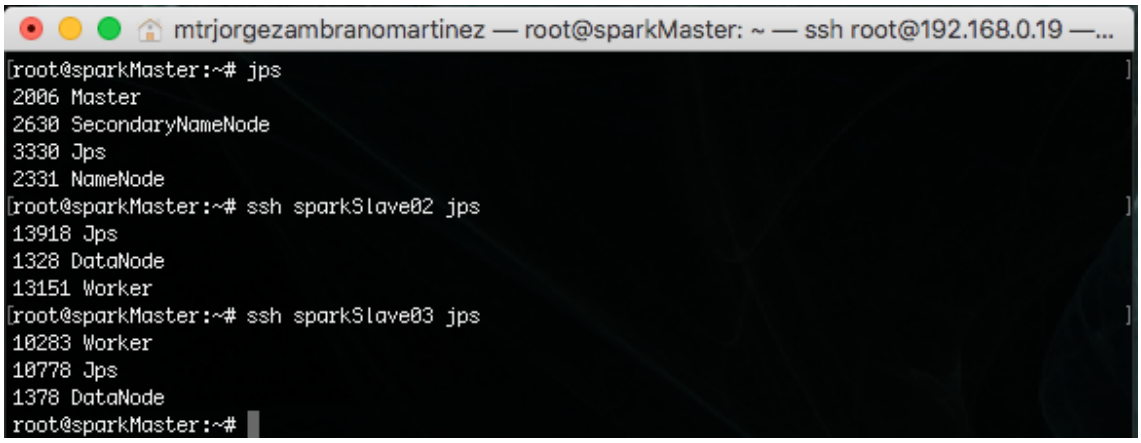


Figura 18. Verificación de ejecución de los frameworks

En el apartado de anexo, se encuentran las ilustraciones de los entornos web de Spark y Hadoop.

4.6 Inyección de datos en clúster Big Data

La inyección de datos al clúster virtual, se lo realiza por medio de un programa desarrollado en Python, como se mencionó anteriormente, se lo puede realizar en diferentes lenguajes de programación. He escogido realizarlo en Python debido a su sencillez tanto en la creación del programa como en la importación de las diferentes librerías, y su rápida ejecución sin pasar por una compilación.

La ventaja de Python, es crear programas con pocas lineas de código. Para hacer uso de una arquitectura, en ese caso de Spark Streaming, hay que importar librerías de PySpark, como el siguiente código.

```
import os,hashlib
from pyspark import SparkContext
from pyspark.sql import *
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.sql.streaming import *
from time import sleep
```

Sin embargo, es necesario la creación de una sesión en Spark, denominando el nombre que va a tener el programa con el siguiente código.

```
# Begin a session in Spark
spark = SparkSession.builder.appName("Spark01").getOrCreate()
```

Como hemos expresado, la generación de datos sintéticos conlleva una estructura para lo cual, el desarrollador debe colocar la estructura del archivo CSV con la que esta compuesta dichos archivos y en el mismo orden, como se tiene en el siguiente parte del código.

```
# Structure of data in csv files
userSchema = StructType()\
.add("duration", "string")\
.add("protocol_type", "string")\
.add("service","string")\
.add("src_bytes", "string")\
.add("dst_bytes", "string")\
.add("flag", "string")\
.add("land", "string")\n
.add("wrong_fragment", "string")\n
.add("urgent", "string")\n
.add("attack", "string")\n
.add("sourceIP", "string")\n
.add("targetIP", "string")\n
.add("pass", "string")
```

Como siguiente paso, es la lectura de los archivos creados, en este caso, Spark Streaming estará atento cuando un nuevo archivo es creado dentro de la carpeta al cual se le pide que escuche. Su código esta compuesto por varias opciones.

La primera es si se realizar una lectura de los archivos único o leer archivos en Streaming, para nuestro caso elegimos que lea en Streaming. La segunda opción es decirle con que carácter esta separado, en este caso usamos la coma. En la tercera opción ingresamos el esquema propiamente escrito con anterioridad. La cuarta opción es decirle al método que en los archivos que va a leer, existe una cabecera, la cual no debe ser almacenada. La quita opción es decir el Número de archivos por disparador de eventos debe capturar. Y la última opción es decirle en que carpeta va a esperar Spark Streaming para la ingesta de datos en el cluste.

Este tipo de captura de datos es el DStream, aunque como se mencionó con anterioridad se puede agregar la librería Kafka para PySpark u otro data stream como se ilustra en la Figura 7.

```
# Read in streaming all the csv files written atomically in a directory
csvDF = spark.readStream.option("sep", ",")\
.schema(userSchema)\
.option('header','true')\
.option("maxFilesPerTrigger", 1)\
.csv('/nfs/spark/big_data/data/')
```

Antes de escribir en el HDFS y/o en el NFS, gracias al framework Spark se puede elaborar cualquier procesamiento, como por ejemplo un filtro que decida que datos ingresan y cuales no. En este, caso ocuparemos la columna de “pass” que se encuentra dentro de los archivos creados sintéticamente, asi

poder filtrar los datos.

Ahora procedemos a procesar los datos en tiempo real, con la siguiente estructura de programación, se puede realizar una consulta a los datos de cuales poseen o no la contraseña segura que fue generada con el programa creador de datos sintéticos.

```
# processing data by secure password
legitimo = csvDF.select('*').where("pass ==
'9ff1105dbe6629c2fda93c47dc61bc94c0249dada9560642555a0809815153b7'")
ilegitimo = csvDF.select('*').where("pass !=
'9ff1105dbe6629c2fda93c47dc61bc94c0249dada9560642555a0809815153b7'")
```

Para el almacenamiento de los datos procesados por Spark, decidí a modo de prueba, que los datos que contengan la contraseña segura se almacenen en el HDFS y los datos que no tengan dicha contraseña segura se almacenen en una carpeta del NFS.

Las opciones que se tiene en este método son, el tipo de formato de salida que se le pueda presentar.

Otra opción que es de carácter obligatorio, la creación de un checkpoint para establecer que el contexto controle periódicamente las operaciones DStream para la tolerancia a fallos del nodo maestro.

La siguiente opción que he colocado es queryName que no es mas que asignarle un nombre a dicha operación de escritura. Y la última opción es quien comienza la ejecución del flujo o stream, sin embargo es ahí como parámetro interno que se le asigna una carpeta en donde enviar los archivos procesados, en este caso enviaremos al NFS los datos intrusos y al HDFS los datos legítimos.

```
# Write hacked data in NFS
a= ilegitimo.writeStream.format('json').option("checkpointLocation",
'checkpoint_intruso').option("queryName",'intruso').start('/nfs/spark/big_data/log_intrus
os')

# Write legal data in HDFS
b= legitimo.writeStream.format('json').option("checkpointLocation",
'checkpoint_legitimo').option("queryName",'Legal').start('hdfs://sparkMaster:9000/user/j
orzamma/big_data/log_legitimo')
```

Para que continúe la ejecución del Spark Streaming es necesario el siguiente

método, del cual solo termina la ejecución si se presenta otro método “stop” o introduce un parámetro en segundo del tiempo de vida de la ejecución.

```
# Wait for the execution to stop
a.awaitTermination(120)
b.awaitTermination(120)
```

En el Anexo incluyo el código completo del programa de Spark Streaming en Python.

Para lograr, dicha inyección de datos, es necesario generar los archivos que contienen los datos sintéticos, para lo cual se tendrá que ejecutar el primer programa generador de datos sintéticos en paralelo con el programa de inyección de datos.

Para lo cual el programa de inyección de datos debe ejecutarse con el siguiente comando.

```
spark-submit sparkStreaming_CSV.py
```

De este modo quedará en un estado de escucha en la carpeta, hasta cuando el programa generador de datos sintéticos crea los archivos, y es cuando comienza el procesamiento de los datos con Spark Streaming y almacenándolos en el HDFS y en NFS respectivamente. Esta ejecución esta capturada en Anexo.

El resultado que se obtiene es el almacenamiento en NFS cuando se detecta que el dato sintético posee una contraseña segura no correcta como se observa en la Figura 19.

Figura 19. Ejemplo de datos inyectados en el clúster Big Data en NFS

Del mismo modo, el resultado que se obtiene el almacenar en HDFS, aquellos datos que contengan la correcta contraseña segura. Además, Apache Hadoop HDFS nos permite visualizar la localización de los archivos a través de un portal web, que se puede acceder con:

```
http://direccion-IP:50070/explorer.html#/user/jorzamma/big_data/log_legitimo
```

Ambas formas de visualizar los datos almacenados en HDFS del clúster podemos observa en la

Figura 20.

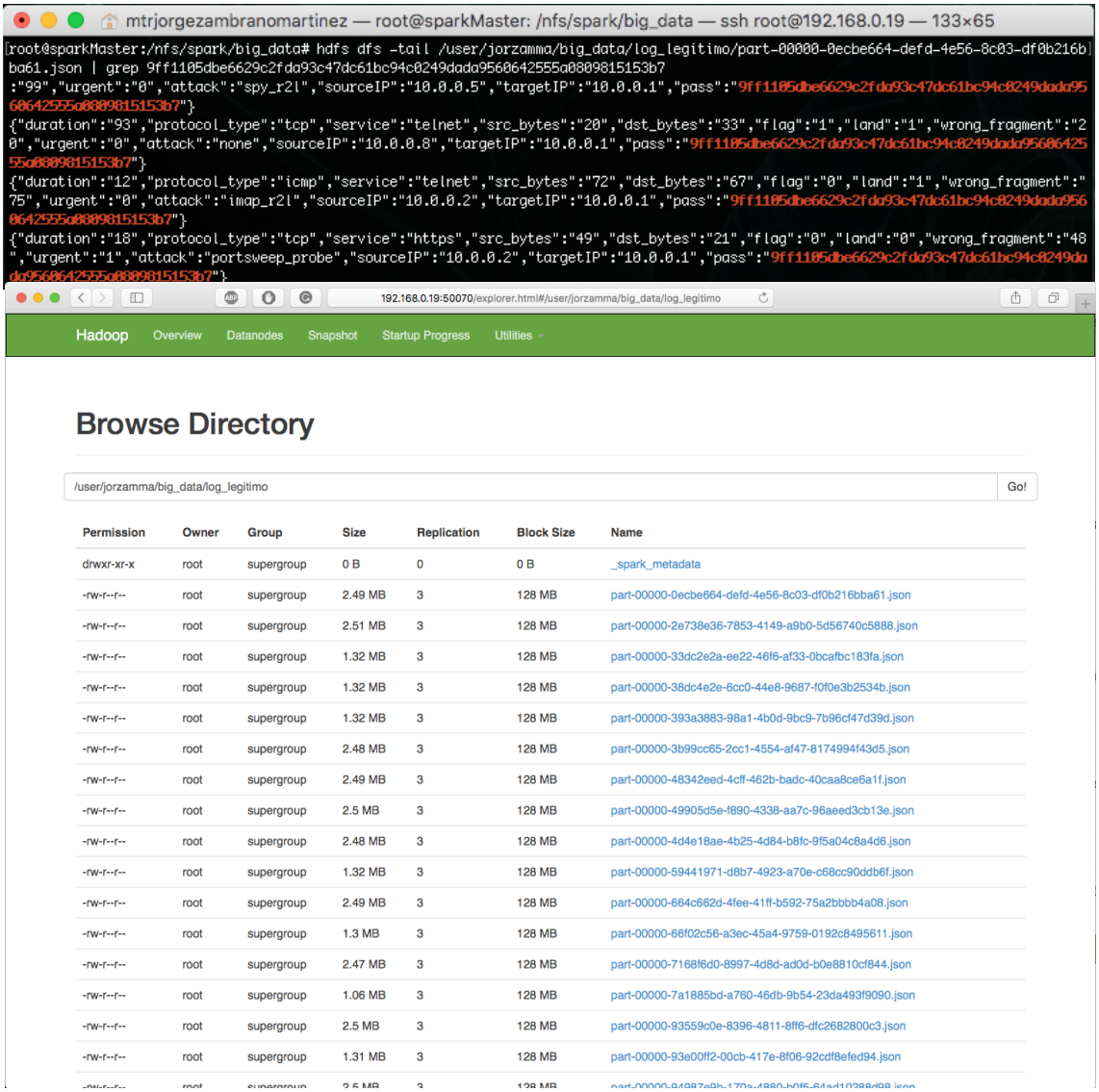


Figura 20. Ejemplos de datos inyectados en el clúster Big Data en HDFS

5. Conclusiones

Actualmente, en un mundo cada vez más digitalizado, diariamente se crean grandes volúmenes de datos en diferentes entornos tantos laborables, educativos, científicos, etc., por lo que es necesario crear arquitecturas que procesen, almacenen y visualicen enorme cantidades de datos. Estas arquitecturas deben responder a las cuatro V's (Volumen, Variedad, Veracidad, Velocidad).

Durante el desarrollo del trabajo se pueden observar varias tecnologías Big Data específicas para cada uso, con sus ventajas e inconvenientes bien definidas. Dado el objetivo del trabajo que es analizar las diferentes tecnologías de Big Data en especial del ecosistema Hadoop, para llegar a procesar, y almacenar los datos que generan alrededor de 3,000 servidores de las Notarías Españolas.

Ante este escenario, se llega a la conclusión de utilizar la tecnología Apache Spark por múltiples ventajas para este escenario como el paralelismo simplificado que se utiliza con un módulo llamado Spark Streaming, en el que crea varias particiones del "Resilient Distributed Dataset" (RDD) trabajando cada una en un nodo esclavo o worker. Otro punto de vista es la alta escalabilidad que se presenta, así como el procesamiento complejo de los datos con funciones de alto nivel. Esto nos conlleva a un alto rendimiento a comparación de otras tecnologías de datos distribuidos, y un procesamiento de datos más rápido que Apache Hadoop MapReduce. Además, es respaldado en la integración de esta tecnología en grandes compañías como IBM, Amazon, Microsoft y Google.

Sin embargo, todo el procesamiento de los datos que se tiene con Apache Spark es realizado en la memoria del clúster, por lo que es necesario que sea almacenado en algún sitio como en un Sistema de Archivos de Red (NFS) o en el mismo Apache Hadoop HDFS. En este caso hemos elegido los dos tipos de sistemas de almacenamiento, el NFS debido a la propia infraestructura de un clúster, que se observa con un solo entorno comunicados entre los distintos nodos en un mismo nodo de almacenamiento, y Apache Hadoop HDFS debido a i) la eficiente escalabilidad haciendo crecer el clúster simplemente añadiendo nuevos nodos sin la necesidad de hacer ajustes previos que modifiquen la estructura inicial o estar atados a características del diseño del clúster; ii) al almacenamiento a bajo costo por el hecho de no almacenar de las maneras

tradicionales como una base de datos si no que asigna datos en los computadores tradicionales abaratando costes económicos, iii) a la flexibilidad que presenta en agregar nodos y ganar almacenamiento, y iv) tolerabilidad a fallos permitiendo recuperar datos de forma segura, así un nodo haya fallado.

Así, se pudo realizar pruebas con estas dos tecnologías de Big Data escogidas sobre un clúster virtual montado en un computador personal, y simulando el escenario impuesto para este trabajo. Para dichas pruebas, se realizó la inyección de datos sintéticos generados por un programa creado en Python, para luego ser capturado esos datos con Spark Streaming, y llevarlos a un procesamiento antes de ser almacenado tanto en el NFS como en el HDFS.

Para trabajo futuro se puede implantar dicho clúster virtual en uno real, trayendo varias ventajas para las Notarías Españolas, además de crear dentro de los datos generados seguridad impuesta como las mencionadas en este trabajo, facilitando conocer que datos son seguros y cuales no para el clúster Big Data. Los aspectos que se tendría que considerar para pasar del prototipo de este proyecto a un entorno real son i) las características de los computadores debido a su vital importancia tanto en el procesamiento como en el almacenamiento; ii) la selección de un tipo de red con eficiente ancho de banda, con al menos dos vía de comunicación primordialmente para una tolerancia a fallos, y con una comunicación sin retardos ni perdidas de datos; iii) la replicación de los datos a otro sistema de archivos de red paralelo al principal; iv) disponibilidad de escalabilidad para un futuro tanto en la red como en los nodos del clúster.

6. Glosario

Streaming: Retransmisión continua sin interrupción

HDFS: Hadoop Distributed File System

NFS: Network File System

RDD: Resilient Distributed Dataset

IP: Internet Protocol

7. Bibliografía

[1] Oguntimilehin, A., & Ademola, E. (2014). A review of big data management, benefits and challenges. *Journal of Emerging Trends in Computing and Information Sciences , 5* (6), 433-438.

[2] Dobre, C., & Xhafa, F. (2014). Parallel programming paradigms and frameworks in big data era. *International Journal of Parallel Programming , 42* (5), 710-738.

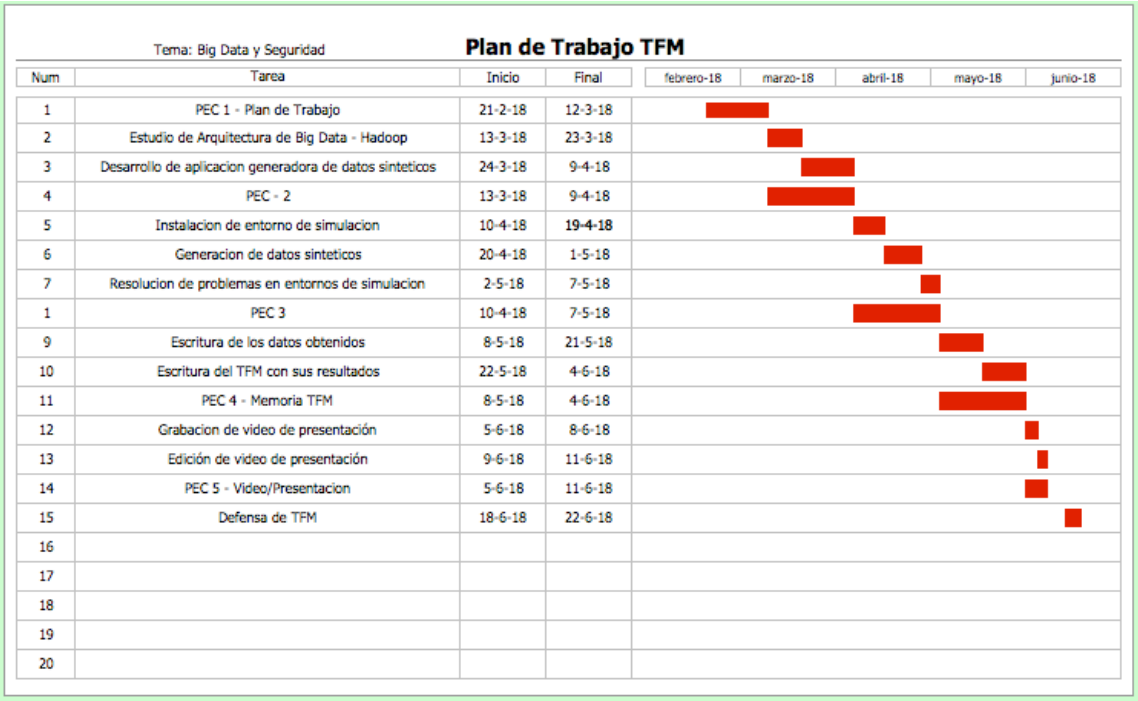
[3] Apache. (17 / 05 / 2018). *Apache Hadoop*. Consultado el 26 / 05 / 2018, a Apache Hadoop: http://hadoop.apache.org

[4] White, T. (2012). *Hadoop: The Definitive Guide* (Vol. 3). USA: O'reilly Media.

[5] Apache. (28 / 02 / 2018). *Apache Spark*. Consultat el 26 / 05 / 2018, a Apache Spark: http://spark.apache.org

[6] Samandi, Y., Zbakh, M., & Tadonki, C. (2016). Comparative study between Hadoop and Spark based on Hibench nenchmarks. *Cloud Computing Technologies and Applications IEEE* , 267-275.

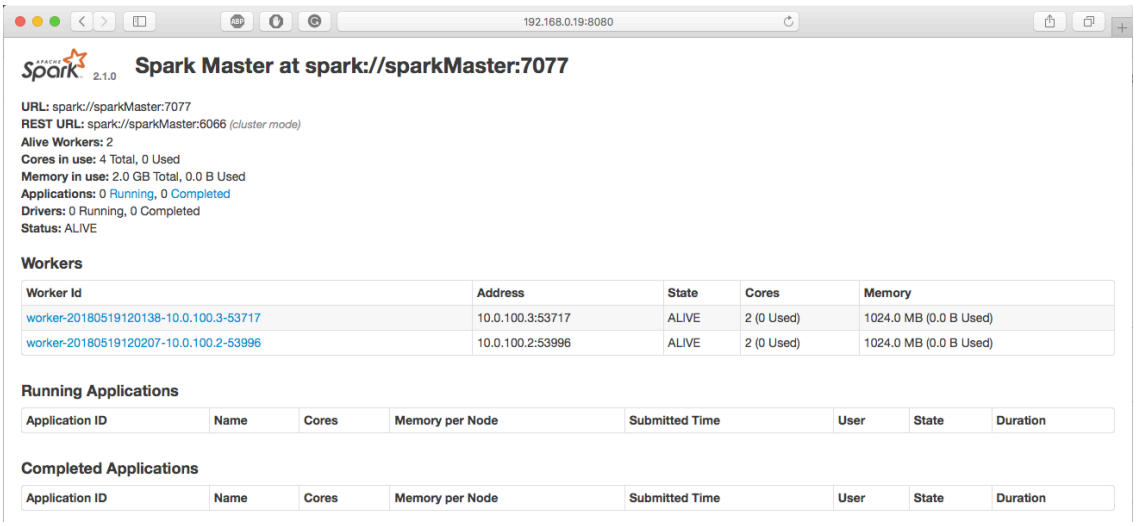
8. Anexos



Tema: Big Data y Seguridad — **Plan de Trabajo TFM**

Num	Tarea	Inicio	Final
1	PEC 1 - Plan de Trabajo	21-2-18	12-3-18
2	Estudio de Arquitectura de Big Data - Hadoop	13-3-18	23-3-18
3	Desarrollo de aplicacion generadora de datos sinteticos	24-3-18	9-4-18
4	PEC - 2	13-3-18	9-4-18
5	Instalacion de entorno de simulacion	10-4-18	19-4-18
6	Generacion de datos sinteticos	20-4-18	1-5-18
7	Resolucion de problemas en entornos de simulacion	2-5-18	7-5-18
1	PEC 3	10-4-18	7-5-18
9	Escritura de los datos obtenidos	8-5-18	21-5-18
10	Escritura del TFM con sus resultados	22-5-18	4-6-18
11	PEC 4 - Memoria TFM	8-5-18	4-6-18
12	Grabacion de video de presentación	5-6-18	8-6-18
13	Edición de video de presentación	9-6-18	11-6-18
14	PEC 5 - Video/Presentacion	5-6-18	11-6-18
15	Defensa de TFM	18-6-18	22-6-18
16			
17			
18			
19			
20			

Figura 21. Cronograma del plan de trabajo



192.168.0.19:8080

Spark 2.1.0 **Spark Master at spark://sparkMaster:7077**

URL: spark://sparkMaster:7077
REST URL: spark://sparkMaster:6066 *(cluster mode)*
Alive Workers: 2
Cores in use: 4 Total, 0 Used
Memory in use: 2.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20180519120138-10.0.100.3-53717	10.0.100.3:53717	ALIVE	2 (0 Used)	1024.0 MB (0.0 B Used)
worker-20180519120207-10.0.100.2-53996	10.0.100.2:53996	ALIVE	2 (0 Used)	1024.0 MB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Figura 22. Entorno Web de Apache Spark

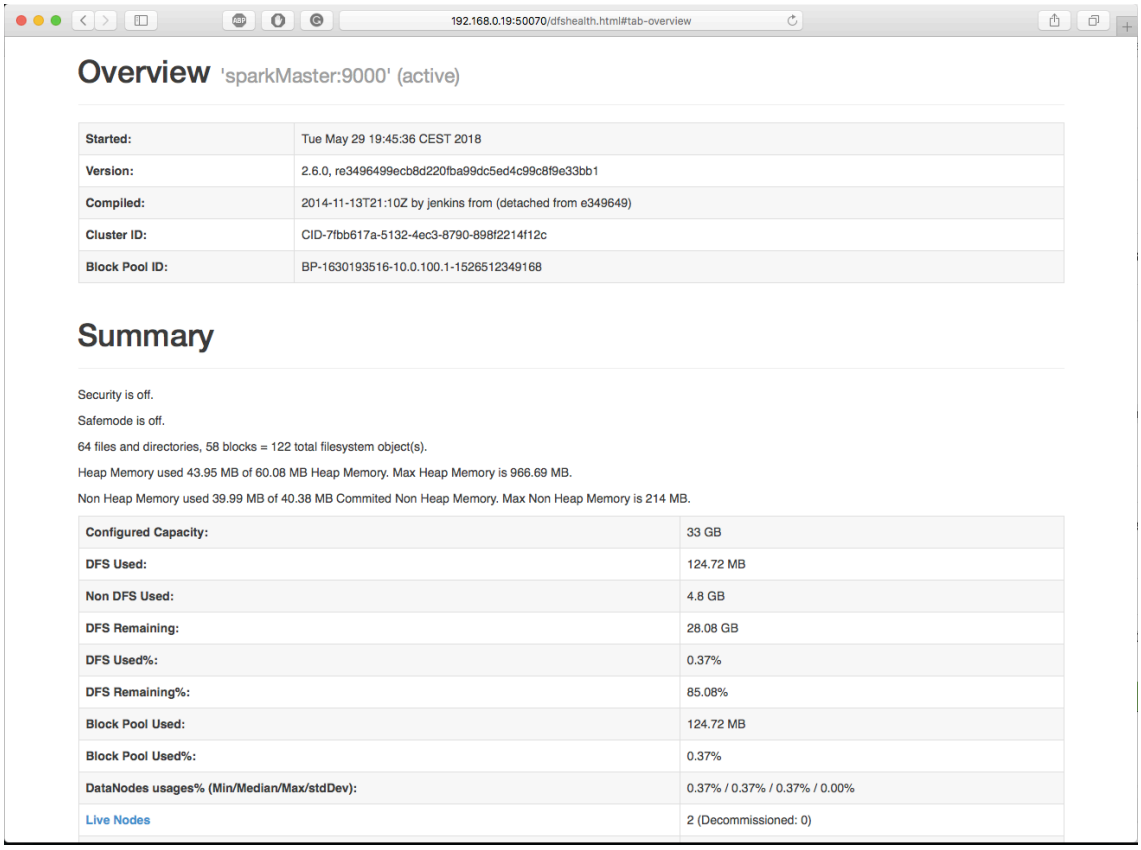


Figura 23. Entorno web de Apache Hadoop

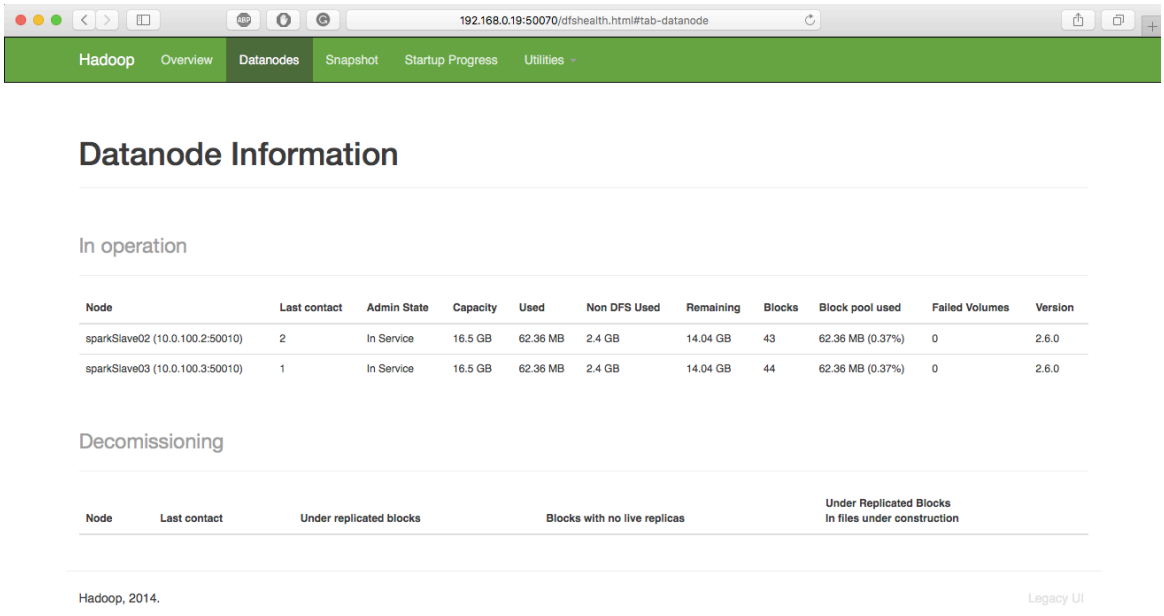


Figura 24. Información de los nodos de datos

GenData.py

GenData.py 〉 No Selection

```python
#!/usr/bin/env python
import os,sys,datetime,subprocess,time,random,hashlib

def genPassSecure():
    return (hashlib.sha256(('deimos') + ('test'.encode('hex'))).hexdigest())

def genPassFakeSecure():
    return (hashlib.sha256(('password') + (os.urandom(16).encode('hex'))).hexdigest())

def genFeatures():
    #basic Features
    duration=random.randint(1,100) #length (number of seconds) of the connection
    protocol_type=random.choice(['tcp', 'udp', 'icmp']) #type of the protocol, e.g. tcp, udp, etc.
    service=random.choice(['http', 'https', 'telnet']) #network service on the destination, e.g., http, telnet, etc.
    src_bytes=random.randint(1,100) #number of data bytes from source to destination
    dst_bytes=random.randint(1,100) #number of data bytes from destination to source
    flag=random.randint(0,1) #normal or error status of the connection
    land=random.randint(0,1) #1 if connection is from/to the same host/port; 0 otherwise
    wrong_fragment=random.randint(0,100) #number of 'wrong' fragments
    urgent=random.randint(0,1) #number of urgent packets

    attack=random.choice(['none', 'back_dos', 'buffer_overflow_u2r',
        'ftp_write_r2l','guess_passwd_r2l','imap_r2l','ipsweep_probe','land_dos','loadmodule_u2r','multihop_r2l','neptune_dos','nmap_probe','perl_u2r','phf_r2l','po
        d_dos','portsweep_probe','rootkit_u2r','satan_probe','smurf_dos','spy_r2l','teardrop_dos','warezclient_r2l','warezmaster_r2l']) #attack categories

    randomIP=('%s.%s.%s.%s'%(str(random.randint(1,254)),str(random.randint(1,254)),str(random.randint(1,254)),str(random.randint(1,254))))
    authorizedIP=['10.0.0.2', '10.0.0.3', '10.0.0.4', '10.0.0.5','10.0.0.6','10.0.0.7','10.0.0.8','10.0.0.9','10.0.0.10']
    #simulate IP normal and attacker
    sourceIP=random.choice(['10.0.0.2', '10.0.0.3', '10.0.0.4', '10.0.0.5','10.0.0.6','10.0.0.7','10.0.0.8','10.0.0.9','10.0.0.10',randomIP])
    targetIP='10.0.0.1'

    if sourceIP not in authorizedIP:
        password=genPassFakeSecure() #generate a fake password
    else:
        password=genPassSecure() #generate a real password

    return '%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s'%
        (duration,protocol_type,service,src_bytes,dst_bytes,flag,land,wrong_fragment,urgent,attack,sourceIP,targetIP,password)

def genFile():
    for i in range(1,30):
        print('Creating File log{}.csv'.format(i))
        time.sleep(1)
        with open('data/logss{}.csv'.format(i), 'w') as
            writefile:writefile.write('duration,protocol_type,service,src_bytes,dst_bytes,flag,land,wrong_fragment,urgent,attack,sourceIP,targetIP,pass\n')
            for j in range(0,10000):
                writefile.write('%s\n'%(genFeatures()))

def main():
    genFile()

if __name__ == '__main__':
    main()
```

Figura 25. Código Fuente del generador de datos sintéticos

sparkStreaming_CSV.py

sparkStreaming_CSV.py 〉 No Selection

```python
#!/usr/bin/env python
import os,hashlib
from pyspark import SparkContext
from pyspark.sql import *
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark.sql.streaming import *
from time import sleep

def main():
    # Begin a session in Spark
    spark = SparkSession.builder.appName("Spark01").getOrCreate()

    # Structure of data in csv files
    userSchema = StructType().add("duration", "string").add("protocol_type", "string").add("service","string").add("src_bytes", "string").add("dst_bytes",
        "string").add("flag", "string").add("land", "string").add("wrong_fragment", "string").add("urgent", "string").add("attack", "string").add("sourceIP",
        "string").add("targetIP", "string").add("pass", "string")

    # Read in streaming all the csv files written atomically in a directory
    csvDF = spark.readStream.option("sep", ",").schema(userSchema).option('header','true').option("maxFilesPerTrigger", 1).csv('/nfs/spark/big_data/data/')

    # processing data by secure password
    legitimo = csvDF.select('*').where("pass == '9ff1105dbe6629c2fda93c47dc61bc94c0249dada9560642555a0809815153b7'")
    ilegitimo = csvDF.select('*').where("pass != '9ff1105dbe6629c2fda93c47dc61bc94c0249dada9560642555a0809815153b7'")

    # Write files in format json

    # Show in console
    #a=ilegitimo.writeStream.format('console').start()

    # Write hacked data in NFS
    a= ilegitimo.writeStream.format('json').option("checkpointLocation", 'checkpoint_intruso').option("queryName",'intruso').start('/nfs/spark/big_data/
        log_intrusos')

    # Write legal data in HDFS
    b= legitimo.writeStream.format('json').option("checkpointLocation", 'checkpoint_legitimo').option("queryName",'Legal').start('hdfs://sparkMaster:9000/user/
        jorzamma/big_data/log_legitimo')

    # Wait for the execution to stop
    a.awaitTermination(120)
    b.awaitTermination(120)


if __name__ == '__main__':
    main()
```

Figura 26. Código fuente del uso de Spark Streaming para inyectar datos en el clúster.

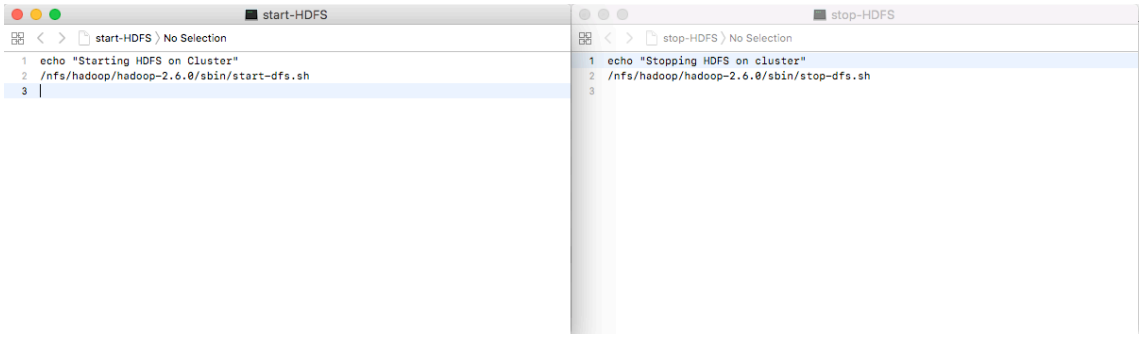


Figura 27. Script para inicializar y detener el Apache Spark.

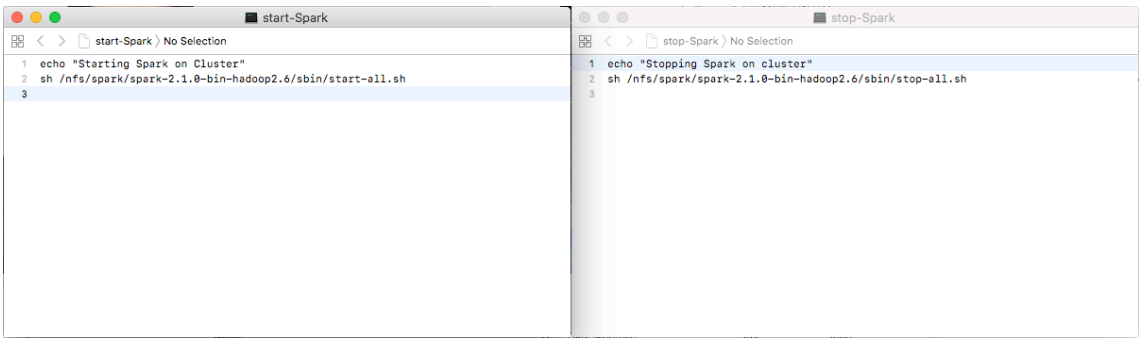
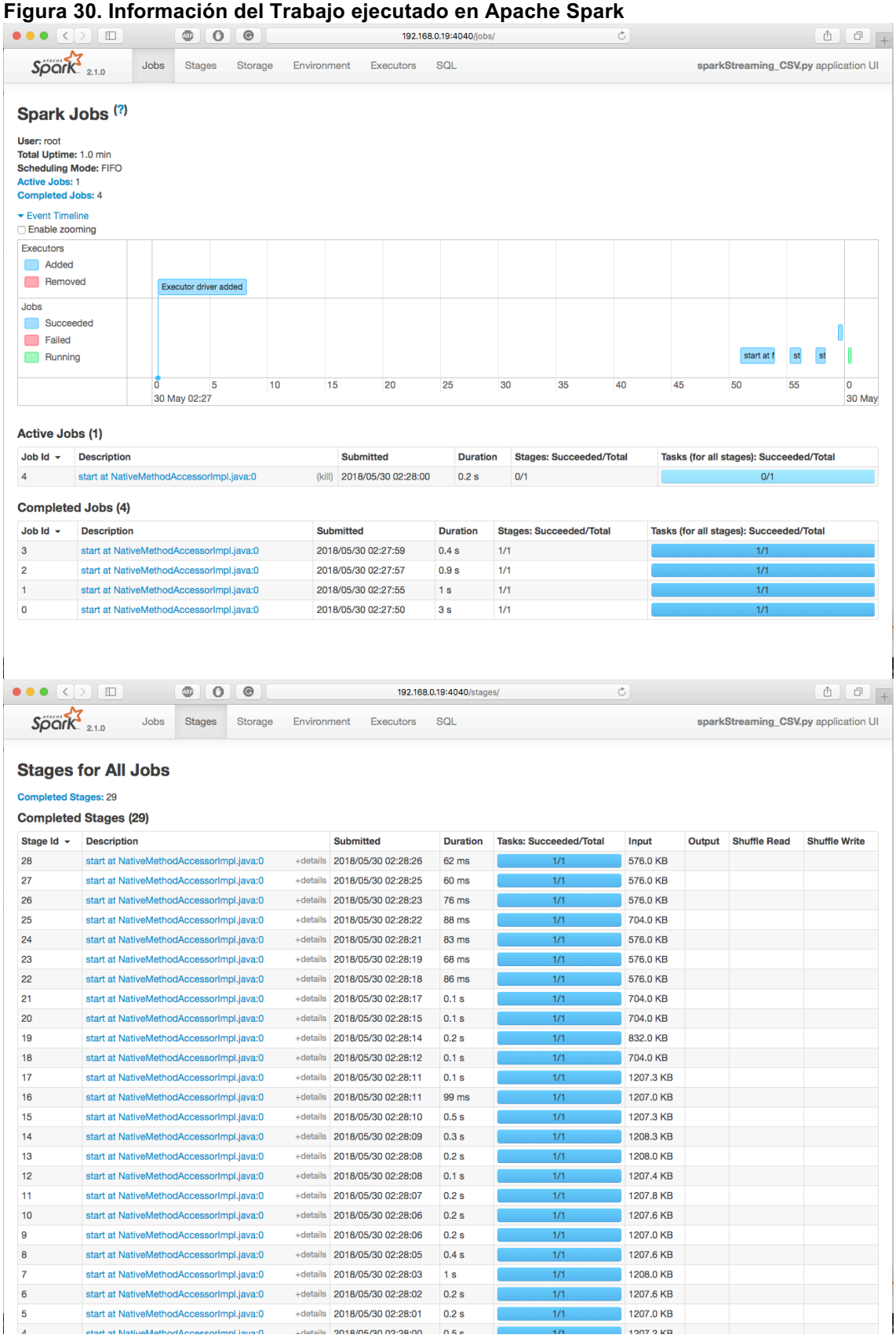
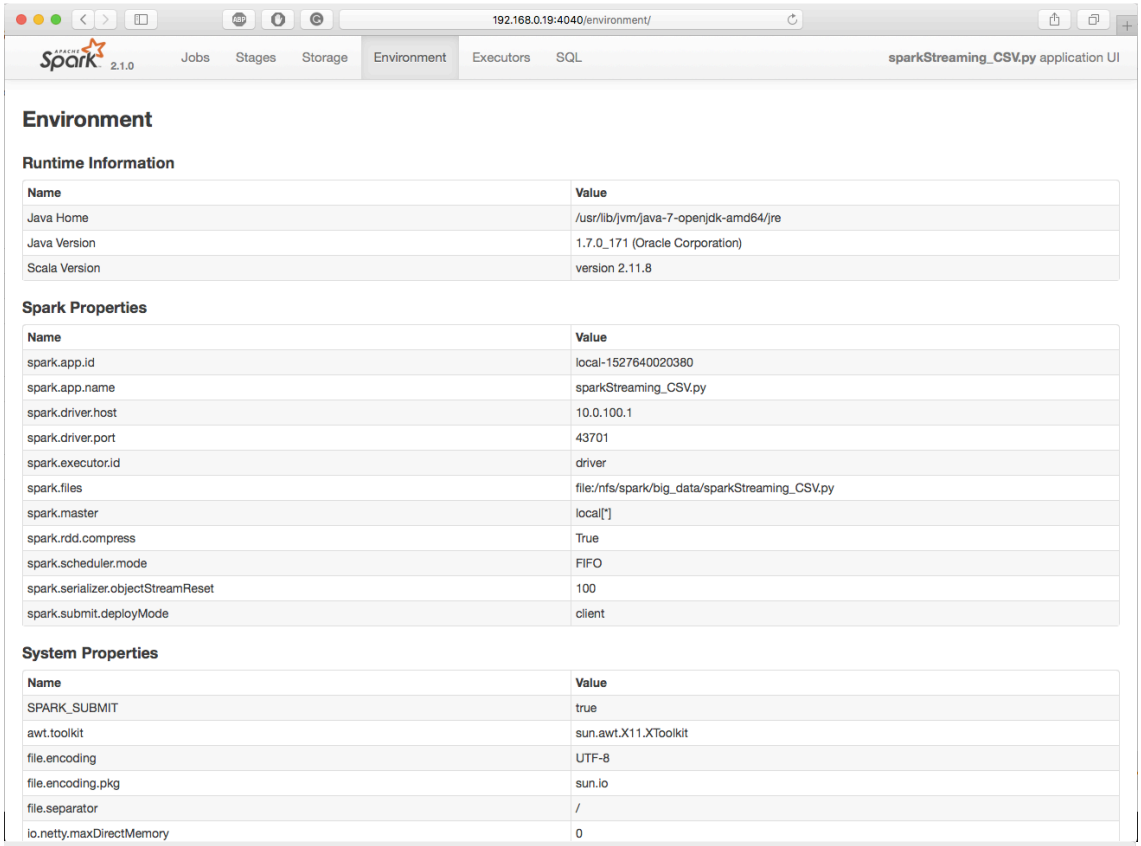


Figura 28. Script para inicializar y detener el HDFS.

Figura 29. Ejecución de los programas en paralelo.

Figura 30. Información del Trabajo ejecutado en Apache Spark





Environment

Runtime Information

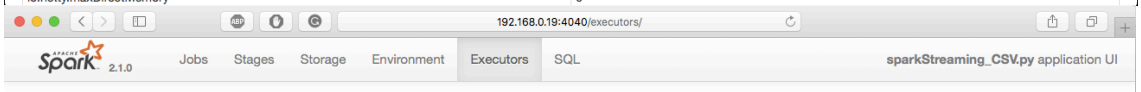
Name	Value
Java Home	/usr/lib/jvm/java-7-openjdk-amd64/jre
Java Version	1.7.0_171 (Oracle Corporation)
Scala Version	version 2.11.8

Spark Properties

Name	Value
spark.app.id	local-1527640020380
spark.app.name	sparkStreaming_CSV.py
spark.driver.host	10.0.100.1
spark.driver.port	43701
spark.executor.id	driver
spark.files	file:/nfs/spark/big_data/sparkStreaming_CSV.py
spark.master	local[*]
spark.rdd.compress	True
spark.scheduler.mode	FIFO
spark.serializer.objectStreamReset	100
spark.submit.deployMode	client

System Properties

Name	Value
SPARK_SUBMIT	true
awt.toolkit	sun.awt.X11.XToolkit
file.encoding	UTF-8
file.encoding.pkg	sun.io
file.separator	/
io.netty.maxDirectMemory	0



Executors

Summary

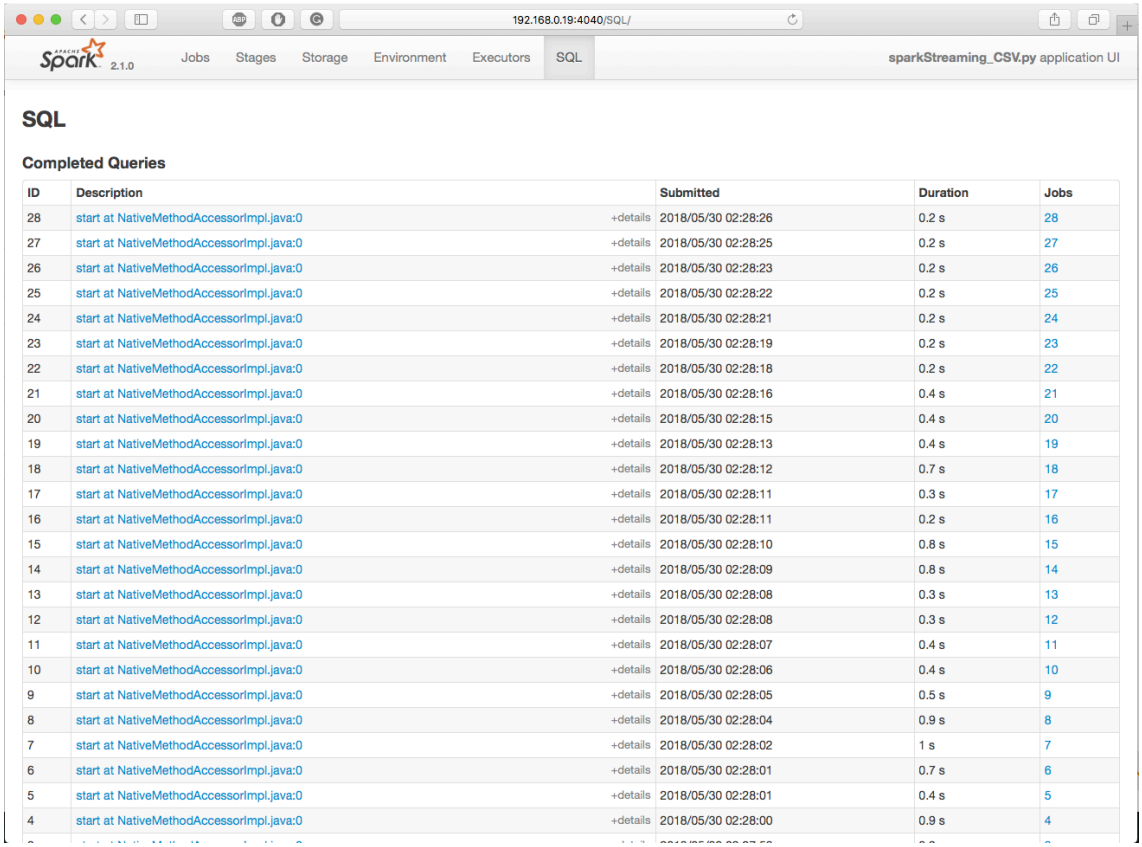
	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(1)	1	21.1 KB / 434 MB	0.0 B	1	0	0	29	29	10 s (0.9 s)	29.5 MB	0.0 B	0.0 B
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(1)	1	21.1 KB / 434 MB	0.0 B	1	0	0	29	29	10 s (0.9 s)	29.5 MB	0.0 B	0.0 B

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	10.0.100.1:44144	Active	1	21.1 KB / 434 MB	0.0 B	1	0	0	29	29	10 s (0.9 s)	29.5 MB	0.0 B	0.0 B	Thread Dump

Showing 1 to 1 of 1 entries Previous 1 Next


192.168.0.19:4040/SQL/
Spark 2.1.0
Jobs
Stages
Storage
Environment
Executors
SQL
sparkStreaming_CSV.py application UI
SQL
Completed Queries
ID
Description
Submitted
Duration
Jobs
28 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:26 0.2 s 28
27 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:25 0.2 s 27
26 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:23 0.2 s 26
25 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:22 0.2 s 25
24 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:21 0.2 s 24
23 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:19 0.2 s 23
22 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:18 0.2 s 22
21 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:16 0.4 s 21
20 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:15 0.4 s 20
19 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:13 0.4 s 19
18 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:12 0.7 s 18
17 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:11 0.3 s 17
16 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:11 0.2 s 16
15 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:10 0.8 s 15
14 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:09 0.8 s 14
13 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:08 0.3 s 13
12 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:08 0.3 s 12
11 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:07 0.4 s 11
10 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:06 0.4 s 10
9 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:05 0.5 s 9
8 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:04 0.9 s 8
7 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:02 1 s 7
6 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:01 0.7 s 6
5 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:01 0.4 s 5
4 start at NativeMethodAccessorImpl.java:0 +details 2018/05/30 02:28:00 0.9 s 4