DISEÑO E IMPLEMENTACIÓN DE CLÚSTER SPARK Y ANÁLISIS DE DATOS DE TRÁFICO MEDIANTE EL USO DE LA LIBRERÍA GRAPHX

TRABAJO FIN DE GRADO 18/19

INGENIERÍA INFORMÁTICA

SANTIAGO MIGUEL GUBERN GONZÁLEZ

TUTORES:

JAVIER JESÚS SANCHEZ MEDINA

ANTONIO ANDRÉS OCÓN CARRERAS

RESUMEN

Diseño, configuración y evaluación de un clúster de Big Data basado en Apache Hadoop Yarn y Apache Spark para el procesamiento y análisis de datos de movilidad, y visualización de datos en R. Utilizando datos de movilidad de vehículos sonda proporcionados por el centro de investigación NEC en Heidelberg, Alemania, han sido preprocesados en R para poder crear grafos a partir de ellos gracias a la librería GraphX de Apache Spark. Utilizando el lenguaje de programación Scala se crean los grafos, se calculan diferentes métricas de centralidad de manera distribuida, y se realiza un análisis de speed-up comparando diferentes configuraciones del clúster. Creando un script en R, se realiza el análisis de los tiempos de ejecución y la representación de las métricas en mapas geográficos.

ABSTRACT

Design, configuration and evaluation of a Big Data cluster based on Apache Hadoop Yarn and Apache Spark, for the processing and analysis of mobility data. Using a probe vehicles mobility dataset provided by the NEC investigation centre in Heidelberg, Germany, after some pre-processing in R, the corresponding graphs are loaded int the Spark cluster by using the Apache Spark GraphX library. Then some graph centrality measurements are calculated distributedly. Afterwards, a Speed-up analysis is developed comparing several cluster configurations. Finally, R is used to analyse the Speed-up and to do some data GIS visualization of the centrality measurements obtained.

CONTENIDO

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN

1.1. ESTADO ACTUAL

Actualmente en todo tipo de entidad o empresa existen datos que en la mayoría de los casos no se usan todo lo que se debería. E incluso serían capaces de obtener muchos más datos de los que obtener información importante y de los que sacar ventajas competitivas o mejorar en sus procesos. Solo grandes empresas o aquellas que están más actualizadas y conocen los avances de los sistemas los aprovechan y sacan beneficio.

Este proyecto trata sobre el Big Data, el análisis de una tecnología para su procesamiento y la comprobación de su funcionamiento. El Big Data, antes que nada, que es el concepto que aparece cuando la cantidad de datos sobre los que se necesita trabajar son tan grandes que los sistemas tradicionales no son capaces de procesarlos. Estos temas son de gran interés para realizar un aprovechamiento de los datos y al ser relativamente tan modernos y poco explorados por la mayoría dan un gran valor a los conocimientos obtenidos y las conclusiones sacadas.

1.2. MOTIVACIÓN

Personalmente, no poseía ninguna idea sobre algún tema para basar este trabajo de fin de grado en ello, por lo que escogí uno de la lista ofrecida por la Universidad. Entre todos los que pude ver, leer y analizar, observé que había mucha demanda de aplicaciones para móviles o páginas webs. No pretendo menospreciar a dichos tipos de trabajos, pero no me llamaban la atención porque se trata de temas que ya he aprendido durante el curso en cierta medida, aunque me sirviera para especializarme y conocer en mucha más profundidad. Buscaba más bien un trabajo que me sirviera para aprender algo nuevo y que no conociera tanto.

En la carrera de Ingeniería Informática, no se imparte demasiado contenido sobre ciencia de datos, siendo una temática bastante importante. En un mundo en el que la información es lo más importante, para una empresa o entidad que quiera prosperar es

vital que se traten los datos correctamente y cuantos más datos se obtengan o se generen, más información y posteriormente, conocimiento, se podrá extraer de ellos. Es por ello, por lo que nace el término Big Data, que define a un gran conjunto de datos estructurados o no estructurados con los que las aplicaciones informáticas tradicionales no pueden trabajar adecuadamente y de manera rápida y eficiente.

Consiguiendo los objetivos planteados se introduce en la universidad este paradigma de la informática. Con el proyecto finalizado se permite que otros conozcan la manera de instalar, configurar y hacer funcionar un clúster Hadoop con Spark y utilizar los grafos para realizar diferentes tipos de cálculos interesantes que arrojen información importante sobre grandes cantidades de datos. Así como para sacar provecho a una serie de máquinas, ya desfasadas que se encuentran en el CICEI y que servirán para probar esta tecnología.

1.3. OBJETIVOS

De este trabajo de fin de grado se espera hacer un diseño y evaluación del funcionamiento y rendimiento de cara a su uso en producción de las tecnologías Apache Hadoop, Apache Spark y GraphX. Además de la creación de una documentación detallada del proceso total.

1.4. JUSTIFICACIÓN DE LAS COMPETENCIAS ESPECÍFICAS CUBIERTAS

TI02: *Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.*

El conjunto de computadoras de las que disponía el CICEI eran en un principio demasiado pobres en recursos para ejecutar las tecnologías software que se iban poner

en prueba. De este modo el trato a los datos y manera de procesarlos era en base a que los algoritmos iban a hacer que con cierta cantidad de datos la aplicación fallase por falta de memoria.

Otro ejemplo de esta competencia es el hecho de haber instalado la tecnología en las máquinas, que puedan comunicarse entre ellas de forma segura, o haber llevado a cabo una mejora de los recursos del clúster para permitir correcto y mejorado funcionamiento de los programas.

TI04: *Capacidad para seleccionar, diseñar, desplegar, integrar y gestionar redes e infraestructuras de comunicaciones en una organización.*

Dicha competencia se puso en desarrollo a la hora de configurar las máquinas para que pudiesen comunicarse entre ellas a través de la red local del CICEI. Todo ello sin permitir más conexiones y mensajes de los que son necesarios para que Hadoop, Spark y otros procesos puedan trabajar.

TI05: *Capacidad para seleccionar, desplegar, integrar y gestionar sistemas de información que satisfagan las necesidades de la organización, con los criterios de coste y calidad identificados.*

Esta competencia se pudo poner en práctica cuando, al tener computadores muy pobres en recursos, tuvimos que decidir si comprar unos nuevos o mejorar de alguna manera los ya existentes. Con el fin de no gastar demasiado y realmente obtener mejor rendimiento, decidimos realizar la compra de módulos de memoria para añadirlos e incrementar drásticamente el potencial. Esto es debido a que el tipo de algoritmos que se ejecutar necesitan bastante memoria RAM, que anteriormente no poseía.

1.5. APORTACIONES

Con la finalización de este trabajo de fin de grado se incluyen múltiples aportaciones relevantes a nuestro entorno que describirán a continuación en esta sección.

La más importante y obvia aportación es la que comprende el diseño e implementación de una arquitectura basada en Hadoop/Spark para la creación y manejo de datos organizados en forma de grafos. En el entorno actual de las islas y España, son muy pocos los que trabajan con estas tecnologías y con Big Data, por lo que con este trabajo estamos a la cabeza de este tema, además de la utilización de grafos, que como se verán, aportarán mucha información sobre los datos.

Esta información obtenida aparece de la siguiente aportación que consiste en el cálculo de diferentes medidas de centralidad usando distintas librerías. Se ha tratado de explotar el paralelismo implícito de Hadoop y Spark y conocer sus límites en las condiciones disponibles. Colateralmente a esta aportación, debido a la falta de recursos para la creación de un clúster medianamente potente, se desarrolló un sistema para el análisis por partes del dataset tratado, de manera que aun con pocos recursos se obtienen los resultados buscados. Este tipo de técnicas son de gran importancia al enfrentarse a problemas del mundo real en el ámbito del Big Data donde no todo será perfecto y donde nunca funcionan las cosas a la primera sin adaptarse y evolucionar.

Por último, se han desarrollado aplicaciones para la visualización de los resultados de la centralidad y el análisis del funcionamiento utilizando R. Este lenguaje, en el mundo de la Ciencia de Datos es de gran importancia en la actualidad, lo que implica un añadido sustancial al trabajo realizado y aprendizaje obtenido.

Todas estas aportaciones se han documentado de manera detallada, así, la memoria de este TFG es en si mismo una aportación de valor para las personas que la utilicen para iniciarse en el uso de estas tecnologías y técnicas.

2. DESARROLLO

2.1. INTRODUCCIÓN

2.1.1. METODOLOGÍA

La metodología elegida al principio de la realización de este trabajo consistía en los siguientes pasos:

1. Comenzar con el desarrollo de un curso en la plataforma educativa Udemy sobre Apache Hadoop y su montaje en un clúster[1]. Este curso permite montar un clúster usando máquinas virtuales que utilizamos como entorno de pruebas antes de configurar el software en las máquinas físicas del CICEI.
2. A medida que se realiza el curso y tras finalizarlo se dedica un tiempo para seguir profundizando en la tecnología de manera teórica y práctica.
3. Realizar un análisis del dataset cedido por el centro de investigación NEC en Heidelberg, Alemania, en virtud de un convenio de colaboración con la ULPGC.
4. Desplegar lo aprendido en el clúster físico y realizar el análisis y evaluación de su funcionamiento.

Esta planificación se llevó a cabo sin problemas, salvo que la última fase era más profunda y compleja que lo planeado desde el inicio. Realmente en esta fase se siguió el proceso iterativo de crear código ejecutable en el clúster, comprobar resultados y repetir aplicando cambios y añadiendo funcionalidades. Todo ello con el fin de demostrar el funcionamiento de las tecnologías y contemplar los límites de la infraestructura.

[1] Enlace al curso aquí.

2.1.2. TECNOLOGÍAS Y MEDIOS

En la realización de este trabajo se han usado las siguientes tecnologías y medios específicos:

- Apache Hadoop y Apache Spark para la gestión y creación del clúster.
- La librería de Apache Spark, GraphX, para el procesamiento de los datos en forma de grafos.
- Linux (Ubuntu) como sistema operativo.
- Scala lenguaje de programación para los programas ejecutables en el clúster.
- R para la transformación de los datos y análisis de los resultados.
- Multicomputador (MIMD) como infraestructura de prueba, proporcionado por el CICEI para la implementación y desarrollo de pruebas.
- Dataset de tráfico cedido por el centro de investigación NEC en Heidelberg, Alemania, en virtud de un convenio de colaboración con la ULPGC.
- Curso de la plataforma Udemy: Monta un cluster Hadoop Big Data desde cero.

El clúster, que se encuentra en el CICEI, consiste en 8 máquinas con las siguientes especificaciones finales:

- 2 procesadores AMD Opteron 246 a 1995Mhz, que poseen 1 núcleo y 1 hilo de procesamiento cada uno.
- 4 módulos de 2GB DDR1 a 400Mhz de memoria RAM. Al inicio del trabajo solamente se disponían de 2GB de memoria RAM, pero era evidente que no iba a ser factible para el procesamiento de Big Data.
- 2TB de almacenamiento en disco HDD, conectado por Sata 1.

Durante el tiempo de pruebas y aprendizaje se hizo uso de mi portátil para correr en él 3 máquinas virtuales donde se desplegó la tecnología. Sus especificaciones son:

- Procesador Intel Core i7-7500U desde 2.90GHz hasta 3.50GHz. Posee 2 núcleos y 4 hilos de procesamiento.
- 16GB de memoria RAM DDR4 a 2133MHz.
- 1TB de disco duro HDD.

2.2. CONCEPTOS TEÓRICOS

2.2.1. BIG DATA

Big data es la cantidad enorme de datos que además pueden ser no estructurados. Si tenemos una base de datos relacional en Oracle o MySQL, por ejemplo, estos datos estarían en tablas, columnas, etc., es decir, estructurados. En cambio, si los datos son obtenidos de otras fuentes, lo más normal es que se encuentren sin ninguna estructura concreta.

Las grandes empresas tecnológicas del mundo generan cada segundo grandes cantidades de datos que, para poder extrapolar la información que contienen, necesitan de entornos Big Data, ya que el Hardware y Software estándar no es capaz de procesarlo todo en unos tiempos razonables.

El Big Data normalmente se puede definir con cinco Vs: Volumen, Variedad, Velocidad, Veracidad y Valor.

- Volumen: como su nombre indica, grandes cantidades de datos que superan las capacidades de procesamiento de sistemas tradicionales.
- Variedad: estos datos se pueden encontrar estructurados (bases de datos), no estructurados (datos de páginas webs p.ej.) o semiestructurados (ficheros de logs p.ej.) .
- Velocidad: se consiguen tiempos de extracción de información mucho más rápidos que con otros sistemas de procesamiento.
- Veracidad: al existir tal cantidad de datos y de la variedad de estos, se debe de dudar de la veracidad que posean. Es por ello por lo que se necesita de un proceso de limpieza y verificación de los datos que se van a procesar.
- Valor: se trata de la importancia y las consecuencias que pueda tener la información obtenida a partir del procesamiento de los datos.

Debido a la falta de capacidad de las tecnologías tradicionales para el procesamiento de datos se necesitan nuevas estrategias. Para solucionar esta problemática se utiliza la computación distribuida para datos y procesos, que como veremos es como trabaja Apache Hadoop.

2.2.2. APACHE HADOOP

Apache Hadoop es un framework de software distribuido de datos y procesos. Es un sistema de gestión de clústeres que permite la escalabilidad. Del mismo modo puede trabajar con sistemas relativamente poco potentes en cuanto a Hardware, desde el punto de vista de un servidor. Un ejemplo de servidor "barato" tendría alrededor de 8 o 16 GB de memoria RAM y 4 núcleos por nodo. Aun así, es capaz de trabajar con sistemas mucho más pobres como se ha hecho en este trabajo de fin de grado, que poseíamos máquinas con 2 núcleos, 2 hilos de procesamiento y 8 GB de memoria RAM. Esto es posible, ya que Hadoop está diseñado para ejecutarse en servidores de bajo coste y con una gran tolerancia a fallos, es decir, en el caso en el que un nodo se estropee, los procesos que se estaba ejecutando en él, serán traspasados a otro.

Apache Hadoop sigue la filosofía de divide y vencerás. Esto significa que, en el mejor caso, si tenemos un proceso que tarda 10 minutos en un sistema tradicional, en Hadoop y con 10 máquinas trabajando de forma distribuida tardaría 1 minuto. En el mundo real estos niveles de mejora de velocidad no llegan a darse prácticamente en ningún caso, aunque sí que existe una notable mejora de tiempos con respecto al sistema no distribuido.

El conjunto de computadoras, llamado clúster, será dividido por Apache Hadoop en maestro y esclavos. Los nodos maestros gobernarían a los esclavos que son los que realmente realizan el procesamiento de los datos.

Apache Hadoop se compone de varias librerías de código abierto:

- Hadoop Common: librerías base de Hadoop.
- MapReduce/Yarn: librerías para los procesos.
- Hadoop Distributed File System (HDFS): librería que compone el sistema de ficheros de Hadoop.

A estas librerías básicas de Hadoop se añaden muchas otras para obtener mejoras y funcionalidades como Spark, HBase, Hive, entre otras. En este trabajo de fin de grado se instalará y utilizará Spark, más concretamente, Spark GraphX.

Hadoop está construido con el lenguaje de programación Java, lo que implica que cualquier máquina capaz de correr Java será capaz de ser parte de una infraestructura Hadoop. Esto permite tener un alto rango de tipos de computadoras que pueden ejecutarlo, incluso en un mismo clúster, funcionando en entornos heterogéneos.

2.2.3. HADOOP HDFS

Se trata del sistema ficheros distribuido de Hadoop pensado para hardware no especializado y de bajo coste, posee una gran tolerancia a fallos, es escalable y permite almacenar gran cantidad de datos. Debido a la necesidad de trabajar con tantos datos, HDFS permite un acceso de alto rendimiento y la posibilidad de acceso a flujos de datos desde los mismos ficheros.

Del mismo modo que Hadoop, HDFS posee una arquitectura maestro/esclavo. Se llama NameNode al servidor maestro y DataNodes a los esclavos. Esta estructura permite añadir ficheros que internamente son divididos en 1 o más bloques que se reparten entre los diferentes DataNodes, o nodos esclavos.

En un clúster existirá un NameNode que se encarga de manejar el sistema de ficheros y regula el acceso por parte de los clientes. Realizará operaciones como abrir, cerrar y renombrar a los ficheros y directorios, así como determinar el lugar en el que se envían o se encuentran los bloques de los ficheros en los diferentes nodos esclavos. Es decir, toda la información del espacio de nombres del sistema de archivos y los metadatos de los ficheros se guarda en el NameNode. En caso de error del NameNode todo el sistema de archivos cae, ya es el que controla todo. Para evitar pérdida de información e intentar solucionar esto, existe un proceso llamado Secundary NameNode.

El Secundary NameNode, se dedica exclusivamente a guardar puntos de control de los metadatos del NameNode. Este proceso no es capaz de suplir al NameNode en caso de fallo, pero será necesario cuando se recupere para obtener el estado del sistema de ficheros antes de la caída. La forma de trabajo que tiene junto con el NameNode es la siguiente:

- El NameNode, en su tarea de gestionar los datos en HDFS, va a ir realizando cambios en el sistema. Estos cambios se irán guardando en unos ficheros llamados “edits_000xxx”. Realmente únicamente en otro fichero denominado “edits_inprogress_000xxx”, que contiene todos los cambios actuales realizados.
- Cada cierto tiempo, el Secundary NameNode se encarga de generar un nuevo fichero llamado “fsimage_000xxx” que es una instantánea de los cambios y el estado del sistema. Este fichero lo crea a través del “fsimage_xx000” anterior junto con los cambios del “edits_inprogress_00xxx”, que pasa a ser un fichero llamado “edits_000xxx”, y genera un “edits_inprogress_000xxx” nuevo. Este proceso se lleva a cabo al cabo de un tiempo determinado o tras haber alcanzado un tamaño determinado el fichero “edits_inprogress_000xxx”.

Además, existirán tantos DataNodes como esclavos de HDFS haya, normalmente uno por nodo. Estos se encargan del manejo del almacenamiento del nodo en el que corren. Son responsables de servir las solicitudes de lectura y escritura sobre sus bloques, así como realizar la creación, borrado y replicación de los bloques según las instrucciones del NameNode. Los DataNodes mientras están activos envían cada cierto tiempo un latido al NameNode para que este conozca el estado de estos y conocer si alguno no se encuentra activo.

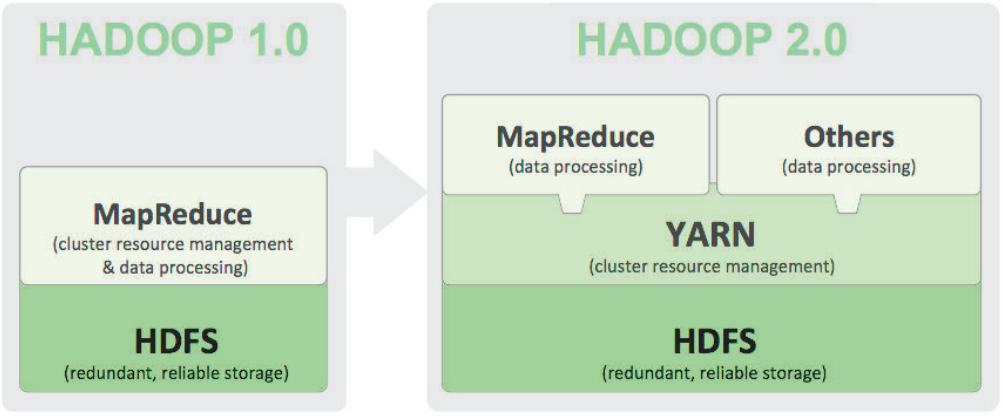
La jerarquía de archivos de HDFS es como la tradicional de muchos sistemas de archivos, es decir, un usuario puede crear directorios y ficheros dentro de ellos. Existe la posibilidad de crear, borrar, mover entre directorios y renombrar ficheros. También posee permisos de usuario y permite aplicar cuotas. Sin embargo, no se pueden crear enlaces simbólicos, pero no se descarta que lo implementen en futuras versiones.

A la hora de guardar ficheros en el sistema de archivos, HDFS los convierte una secuencia de bloques por archivo. Cada uno de ellos es replicado en diferentes DataNodes para obtener una mayor tolerancia a fallos, lo que demuestra la gran seguridad que posee para almacenar ficheros de gran tamaño en el clúster. Esta arquitectura permite que se mejore la fiabilidad y la disponibilidad de los datos y reduce el uso del ancho de banda en los procesos. El número de veces que se replica un bloque se llama factor de replicación y puede ser configurado, así como el tamaño de los bloques en los que se dividen los ficheros. Por defecto el factor de replicación es de 3, se repite cada bloque en 3 DataNodes, y los ficheros se dividen en bloques de 128MB.

2.2.4. HADOOP YARN

Existen dos algoritmos de procesamiento de datos en paralelo, el MapReduce y el Yarn. MapReduce proviene de la versión 1 de Hadoop y Yarn aparece en la versión 2 y adelante. En nuestro caso trabajamos sobre Hadoop 3.1.0, por lo que utilizaremos Yarn.

Actualmente, no es recomendable trabajar con MapReduce, aunque se sigue utilizando con Yarn, formando parte de uno de sus componentes. Este algoritmo estaba pensado para procesos Batch, que son aquellos que no necesitan de ningún control, supervisión o interacción directa del usuario. Además, se encarga tanto del procesamiento de los datos como de la gestión del clúster. Al contrario, Yarn admite tanto procesos batch como interactivos y divide la carga con procesos para datos y otros para gestión del clúster. Con Hadoop 1.0, con MapReduce, existen problemas de escalabilidad en clústeres bastante grandes y problemas de rendimiento en aplicaciones que no son creadas para ser de tipo Batch.



1 Comparativa de versiones Hadoop. Fuente: *Hadoop 1 vs Hadoop 2- The Major Difference You should know*, hdfstutorial.com

Como se puede ver en la imagen, en Hadoop 1.0 MapReduce se encargaba de gestionar el clúster y procesar los datos, mientras que en Hadoop 2.0 en adelante, Yarn se encarga de la gestión de recursos de las máquinas y MapReduce, entre otros, del procesamiento de datos. Esta configuración permite que las operaciones sean más efectivas y reduce la carga computacional de aquellos nodos encargados de ejecutar los cálculos a los datos.

Hadoop hace uso de MapReduce ya que los datos sobre los que se trabajan no suelen estar indexados y son demasiado grandes como para realizar consultas simples y rápidas. Esto conlleva a que se tengan que analizar todos los datos exhaustivamente. El algoritmo es de tipo Batch e implementa procesos en paralelo de forma que distribuye las tareas a través de los nodos del clúster. MapReduce se compone de 3 funciones, la función *map*, las funciones *shuffle* y *sort* y la función *reduce*. La función *map* se encarga

de dividir el problema y mandarlo a las máquinas donde se calcula concurrentemente. El *shuffle* y el *sort* es la fase intermedia en la que se ordenan y consolidan los datos generados por el *map*. Y, por último, La función *reduce* se dedica a reunir los resultados obtenidos de la fase intermedia y los recombina, paralelamente, para obtener una respuesta simple final.

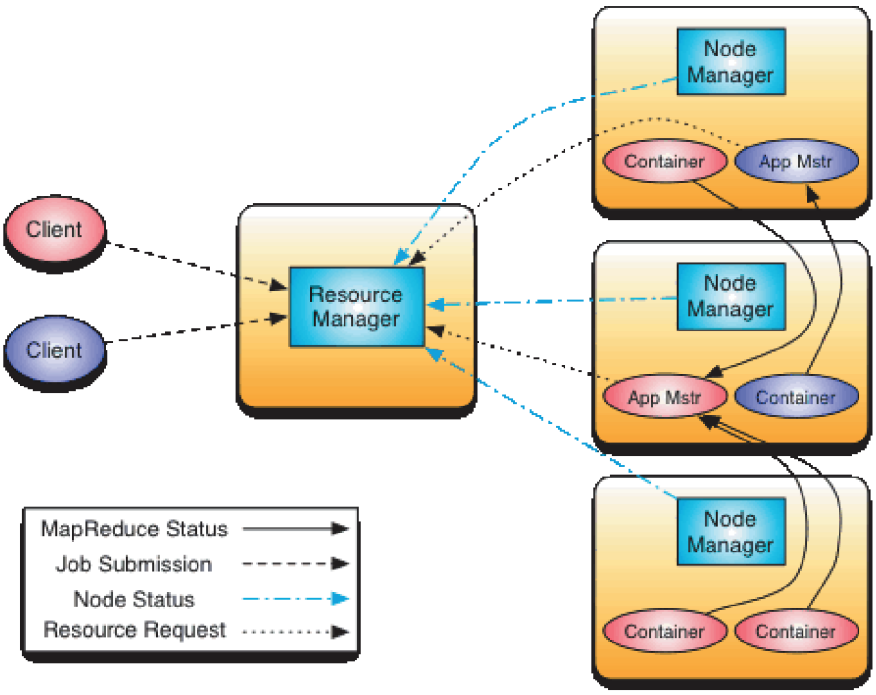
Pasando a comentar la arquitectura YARN, se compone de tres tipos de procesos:

- ResourceManager: Es el proceso maestro y que gestiona todos los recursos del clúster.
- NodeManager: Se trata del proceso esclavo que se encarga de monitorear y gestionar los recursos del nodo en el que se encuentre.
- ApplicationMaster: Es el encargado de gestionar la aplicación a la que pertenezca, su ciclo de vida y planificación.

Entonces, el nodo ResourceManager se dedicará a planificar, monitorear y gestionar los recursos del clúster completo, es el nodo maestro y lo controla todo. Además, el ResourceManager posee dos subcomponentes importantes:

- Scheduler: se trata del proceso que determina cómo se planifican los trabajos y decide qué recursos se van a explotar para cada aplicación.
- ApplicationManager: se trata del proceso encargado de arrancar los ApplicationMaster en base a la información de los recursos que le remita el Scheduler.

El proceso de ejecución de una aplicación comienza cuando un cliente lanza una aplicación contra la infraestructura, es entonces cuando el ApplicationManager, dentro del ResourceManager y gracias a la información del Scheduler, envía la orden de abrir un ApplicationMaster dentro de uno de los nodos. A continuación, el ApplicationMaster, con la ayuda del Scheduler, se ocupará en lanzar los contenedores encargados del procesamiento de los datos (un contenedor es un conjunto de recursos de memoria y CPU encargados de ejecutar una tarea). Esta arquitectura tiene la finalidad de que cada proceso acceda a sus bloques de datos en local.



2 Gráfico de los proceso de Yarn. Fuente: *Documentación oficial Apache Hadoop Yarn*

2.2.5. APACHE SPARK

Apache Spark es un entorno de procesamiento de datos de forma distribuida y en paralelo. Es capaz de trabajar eficientemente a gran escala e implementa procesamiento en tiempo real al contrario que MapReduce, que utiliza procesos tipo Batch. Es mucho más rápido que MapReduce y trabaja de forma masiva en memoria. Actualmente está reemplazando a MapReduce en proyectos Big Data.

Esta tecnología está implementada de menara que puede funcionar sin un gestor del clúster detrás sobre el que se apoye. En ese caso lo controlaría todo él, sin embargo, es posible montarlo sobre Apache Hadoop y utilizar Yarn como gestor. Es, del mismo modo, compatible con Mesos y Kubernetes entre otras tecnologías. En este trabajo de fin de grado se va a utilizar Apache Spark sobre el clúster de Apache Hadoop ya que son totalmente compatibles. Esto permite usar HDFS, usar con MapReduce y se pueden lanzar aplicaciones Spark sobre Yarn.

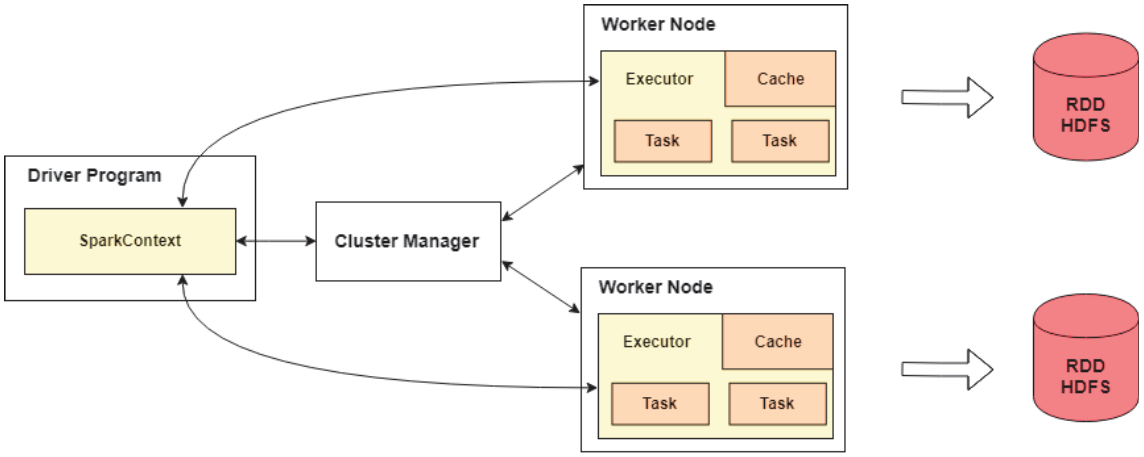
Apache Spark está construido en Scala y permite aplicaciones escritas es otros lenguajes como Java, Python y R. Además, dispone de consola de comandos interactiva para estos lenguajes.

Spark está compuesto de un componente que se trata del núcleo o *core* que se encarga de la gestión de la memoria, la recuperación ante fallos, la planificación y distribución de trabajos, la monitorización y el acceso al almacenamiento. Y un conjunto de librerías específicas que son:

- Spark SQL: librería centrada en el manejo de datasets en base a consultas SQL. Esta librería forma parte del núcleo de Spark a partir de las últimas versiones.
- Spark Streaming: librería centrada en el procesamiento de datos en tiempo real (streaming data).
- Spark MLlib: librería centrada en procesos de Machine Learning.
- GraphX: librería centrada en el manejo de grafos.

En la base de la arquitectura de Apache Spark se encuentran los Resillient Distributed DataSet o RDD, que son una estructura de datos especializada para trabajar en memoria y tolerante a fallos. Básicamente, se trata de colecciones de registros de solo lectura que se particionan entre los nodos y que se manejan el paralelo. Pueden contener objetos de Python, Scala, Java, R o personalizados. Normalmente son creados a través de otros RDDs o desde fuentes externas como HDFS.

Las aplicaciones de Apache Spark se ejecutan como grupos de procesos independientes y coordinados por el objeto SparkContext del programa principal, también llamado Driver Program. El SparkContext es capaz de conectarse a varios tipos de gestores de clúster como el de Spark en sí, Hadoop, Mesos o YARN. Estos gestores reservan recursos para las aplicaciones según estas los necesiten. Una vez conectado Apache Spark al clúster, en cada uno de los nodos necesarios se crea un componente llamado Executor, que es aquel proceso en el cual la aplicación se va a ejecutar realizando las tareas pertinentes. También se hace uso de una caché en cada nodo ya que, como he explicado, Spark es un ejecutor de procesos en memoria. Los datos que procesar se pueden obtener de los propios RDDs ya creados de Spark o directamente a ficheros en HDFS.

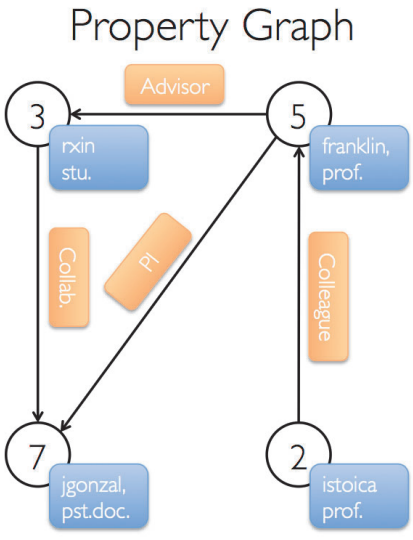


3 Gráfico de los procesos de Spark. Fuente: elaboración propia

2.2.6. SPARK GRAPHX

Como ya se ha mencionado, GraphX es una API de Apache Spark para trabajar con grafos de manera paralela y distribuida. Para conseguir esto, se implementan los RDG, que se pueden definir como una abstracción funcional de los RDDs y contienen colecciones de objetos que, en vez de asociar registros o columnas normales, se asocian a vértices y aristas de un grafo. Lo interesante es que como programador o como usuario de Spark podemos acceder a los datos como grafos o como una colección tradicional, lo que permite muchas posibilidades de implementación de programas. Otra característica interesante de trabajar con GraphX es que ya tenemos una gran cantidad de librerías que tienen algoritmos preparados para poder trabajar de forma fácil con grafos, de forma que podemos construir aplicaciones con muy poca cantidad de líneas y sobre todo mejorar enormemente el rendimiento y velocidad.

Un grafo es un grupo de nodos o vértices que están unidos por arcos o aristas que representan relaciones entre los elementos del conjunto. En GraphX, se representa como un multigrafo[1] dirigido[2] con objetos definidos por el usuario anexados a cada vértice y arista. Cada vértice posee un identificador único y no se impone ninguna restricción de orden en los vértices. Entonces, los arcos son parejas de identificadores de nodos que representan el vértice origen y el destino. Tanto a los vértices como a los arcos se les puede anexar objetos definidos por el usuario.



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

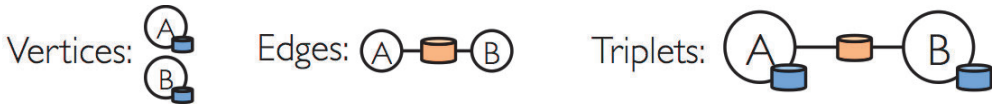
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

4 Ejemplo de grafo en GraphX. Fuente: *Guía de programación oficial Spark GraphX*

[1] Grafo que posee arcos dobles, es decir, arcos que relacionan los mismos vértices. Se pueden encontrar nodos que estén conectados por más de una arista.
[2] Un grafo es dirigido cuando las aristas tienen un sentido definido. En un grafo no dirigido los arcos son relaciones simétricas que no apuntan a ninguno de los vértices.

GraphX nos permite obtener todos los vértices, las aristas de un grafo y, además, los llamados *triplets*, que son la unión de los nodos relacionados y sus aristas.



5 Tipos de objetos que forman un grafo en GraphX. Fuente: *Guía de programación oficial Spark GraphX*

La implementación de los grafos se corresponde a una pareja de dos tipos de colecciones RDD, una para los vértices y otra para las aristas. Los vértices se implementan con la clase `VertexRDD[VD]` y las aristas con la clase `EdgeRDD[ED]`.

2.3. INSTALACIÓN Y CONFIGURACIÓN

A continuación, se llevará a cabo la explicación paso a paso de la instalación de todos los servicios y tecnologías necesarios. Se comenzará con la preparación de una estructura mono nodo en una máquina virtual donde todos los elementos funcionen correctamente. Pasando después a la instalación con múltiples máquinas virtuales trabajando conjuntamente de manera distribuida. Quedando como último paso la implementación de todo lo aprendido en el clúster físico que proporciona el CICEI.

Para trabajar con máquinas virtuales se utiliza el software de virtualización VirtualBox donde se configuran y ejecutan las máquinas virtuales y con sistema operativo CentOS, no hay problema en utilizar otra distribución de Linux. Con el fin de poder probar de manera relativamente profunda es necesario disponer de un equipo con al menos 16GB de RAM, aunque con 8GB también se podría trabajar y probar por encima alguna de las características de la arquitectura.

Para comenzar se debe descargar VirtualBox desde la web oficial e instalarlo en el computador. También es necesario descargar una imagen del sistema operativo elegido, en nuestro caso CentOS. Se debe descargar desde su web la versión 6.7 de 64 bits en formato ISO.

2.3.1. CREACIÓN MÁQUINA VIRTUAL

Una vez todo descargado se abre VirtualBox y se comienza el proceso de creación de las máquinas virtuales. Se clica en el botón de “Nueva” y configuramos las siguientes características:

- Nombre: nodo1.
- Tipo: Linux.
- Versión: Red Hat (64-bit).
- Tamaño de memoria: teniendo en cuenta el computador sobre el que se va a ejecutar y que se deben de crear 3 máquinas virtuales, llegué a la conclusión de darle 4GB de memoria a cada máquina ya que mi portátil cuenta con 16GB.
- Disco duro: es necesario crear un disco duro virtual VDI (VirtualBox Disk Image) reservado dinámicamente de 50GB en un directorio donde deseemos guardar la máquina.

Esta configuración nos creará una máquina virtual con otras características que debemos modificar a continuación clicando en el botón de “Configuración”. Se abre una ventana que contiene todas las características de la máquina. Para trabajar de manera más fluida entre las máquinas y el anfitrión, nos desplazamos a la pestaña “Avanzado” del menú “General” y en las opciones de “Compartir portapapeles” y “Arrastrar y soltar” seleccionamos “Bidireccional”. Esto permite, como sugieren los nombres, copiar y pegar contenido y ficheros entre los diferentes entornos, el anfitrión y la máquina virtual.

Otra configuración que modificar, en caso de que la máquina anfitriona lo permita, sería el número de procesadores asignados a la máquina virtual. Esta opción se encuentra en el menú “Sistema”, pestaña “Procesador”. En teoría cada máquina debería disponer de 4 núcleos para trabajar con Hadoop, pero asignaremos 2 para comenzar a probar sin mucha carga de procesamiento.

Para permitir que cuando arranque esta máquina virtual pueda comenzar a ejecutar el instalador del sistema operativo que hemos descargado anteriormente, nos dirigimos al menú “Almacenamiento”, seleccionamos el CD que aparece como “Vacío” y clicamos en el icono del disco a la derecha. A continuación, seleccionamos el archivo ISO del sistema operativo desde la carpeta donde lo hayamos descargado.

Por último, clicamos en aceptar para guardar todos los cambios realizados. Ya podríamos iniciar la máquina y seguir todos los pasos del instalador del S.O. como si fuese una máquina física normal.

2.3.2. INSTALACIÓN Y CONFIGURACIÓN DEL SISTEMA OPERATIVO

Una vez arrancada la máquina, creada en la sección anterior, aparecerá la pantalla de instalación del sistema operativo. Elegiremos la opción “Install or upgrade an existing system” sin la necesidad de comprobar el disco de instalación antes de la misma. La configuración de CentOS es la siguiente en orden de aparición dentro del instalador:

- Idioma y teclado en español.
- Instalación en los dispositivos de almacenamiento básicos porque no tenemos ningún dispositivo especializado.
- Aceptamos que descarte todos los datos del disco duro.
- El nombre del host le pondremos “nodo1”, lo que nos permite trabajar posteriormente dentro del clúster virtual de manera más cómoda.

- Creamos una contraseña para el usuario root.
- Indicamos que la instalación utilice todo el espacio de nuestro disco virtual.
- Para realizar pruebas dentro del curso que se realizó y fuera de él es necesario seleccionar la instalación “Desktop”, que posee un entorno gráfico.
- Reiniciamos tras haberse completado la instalación y creamos un usuario, por ejemplo, hadoop.

Tras esto ya dispondremos de una máquina virtual con CentOS y con la que probaremos el funcionamiento de Apache Hadoop y Spark.

Tendríamos todo listo para comenzar a trabajar con esta máquina, pero existe un ligero problema que nos lo dificulta. Se trata de que resulta difícil y poco fluido salir de la máquina virtual ya que hay que estar pulsando la tecla de “Ctrl Der”. Además, la resolución es bastante pequeña y no es escalable.

Para solucionarlo es necesario instalar un componente llamado “Guest Additions”. Debemos acceder al menú superior de la ventana y concretamente a la pestaña “Dispositivos”. En ella clicamos en “Insertar imagen de CD de las <Guest Additions>” que lo que hace es insertar un disco virtual con componentes adicionales que permiten solucionar estos problemas. En CentOS aparecerá una ventana de instalación que cancelaremos, abrimos una terminal e iniciamos sesión con el usuario root (`su - root`). Nos desplazamos hasta el directorio “`/media/VBOXADDITIONS_x.x_x`” e instalamos ejecutando el fichero dentro de este directorio:

```
cd /media/VBOXADDITIONS_x.x_x
./VBoxLinuxAdditions.run
```

En el caso de que la instalación falle, seguimos los pasos que se muestran por consola y reintentamos la instalación hasta que se complete. Normalmente se pide que se instalen ciertos paquetes del núcleo o el compilador gcc.

Tras haber superado todos estos pasos, cerramos la sesión actual y volvemos a iniciar sesión para que se reinicie la pantalla y se usen los paquetes instalados. Si todo ha ido correctamente el escritorio de CentOS se debería de reescalar al tamaño de la ventana y podríamos mover el ratón libremente entre el anfitrión y la máquina virtual sin el uso de la tecla “Ctrl Der”.

2.3.3. PREPARACIÓN DE HADOOP

Para poder instalar y ejecutar Hadoop perfectamente, antes, hay que terminar de configurar la máquina. Como se ha explicado, Hadoop está implementado en Java y para realizar pruebas y programas que ejecutar en la tecnología es necesario tener instalado los JDK de Java en la máquina. Podemos ver la versión instalada de Java y si se trata de una JRE o una JDK con el comando "`java -version`". Además, con el comando "`alternatives --config java`" podemos ver las versiones instaladas en busca de una JDK y seleccionarla como la que se va a usar. Normalmente no se encontraría instalado por defecto en el sistema, por lo que se debe de acceder a la web de Oracle y descargar los binarios necesarios. De la lista de posibles versiones de java elegimos la última de java 8 y la que está enfocada a Linux x64, descargamos el formato rpm ya que estamos en CentOS.

Como root, accedemos a la carpeta de descargas e instalamos el JDK con el siguiente comando:

```
rpm -ivh jdk-8uXX-linux-x64.rpm
```

(-i: instalar el paquete, -v y -h: opciones de impresión del proceso por consola)

Una vez acabado el proceso seguirá seleccionada la versión de Java que teníamos anteriormente, para cambiarla tecleamos el comando "`alternatives --config java`" y elegimos el número correspondiente a la versión recién instalada. Con esto ya poseemos Java perfectamente instalado y preparado para compilar los futuros programas y ejecutar Apache Hadoop.

2.3.3.a. SSH

Apache Hadoop para comunicarse con los nodos que tiene a su disposición utiliza el protocolo SSH y las conexiones las realiza utilizando el nombre que dispone de cada máquina, en nuestro caso "nodo1". Aunque disponga nuestra estructura de un solo nodo, los servicios maestros comunicarán con los esclavos a través de SSH incluso si se encuentran en la misma máquina. En este momento, si hacemos el comando "ping nodo1" aparecerá un mensaje comunicándonos el desconocimiento del nodo1. En el caso en el que nos encontremos en un entorno real deberíamos añadir las direcciones de cada máquina nuestro DNS, pero no es el caso. La solución entonces sería editar el

fichero `/etc/hosts`. Añadimos la línea "`<ip de la máquina>  nodo1`" al archivo, como usuario root o con `sudo`, y si ejecutamos "ping nodo1" ahora debería de funcionar y responder perfectamente.

Para conseguir que se comuniquen los futuros nodos entre ellos de manera fluida, es necesario que cada nodo posea una clave privada y otra pública que conozcan los otros. Lo que se debe de hacer será generar las claves con el siguiente comando:

```
ssh-keygen -t rsa
```

El destino de las claves se encuentra en el directorio `/home/<usuario>/.ssh/id_rsa/`, donde se generará la clave privada (`id_rsa`) y la clave pública (`id_rsa.pub`). Esta última, deberá de ser copiada a un nuevo fichero llamado `authorized_keys`, donde se encontrarán todas las claves públicas de las máquinas de confianza. Con esta configuración podremos hacer un ssh y gracias a las claves no tener que introducir manualmente la contraseña de acceso del usuario.

2.3.4. INSTALACIÓN DE APACHE HADOOP

En esta sección se va a descargar, instalar y configurar Hadoop. Comenzaremos accediendo al portal web de Apache Hadoop y descargando de allí una de las versiones disponibles. Accediendo al apartado de descargas de la web podremos observar las múltiples versiones que se pueden descargar, además existe la posibilidad de buscar alguna otra más antigua. En el curso de Udemy, donde se nos explicaba los conceptos básicos Hadoop y su instalación, se hacía uso de la versión 2.7.2. Esa fue la versión elegida para instalar en las máquinas virtuales, pero en clúster físico del CICEI se instaló la versión 3.1.0, que es la versión estable más moderna. No existen grandes diferencias a la hora de la instalación y uso realizado en este trabajo de fin de grado.

De la web de Apache Hadoop obtendremos un fichero comprimido en el directorio de descargas con los binarios necesarios para instalarlo. Mediante una terminal, nos desplazamos al directorio en cuestión utilizando el comando cd y descomprimimos el contenido del fichero descargado con el siguiente comando:

```
tar xvf hadoop-2.7.2.tar.gz
```

(-x: extraer, -v: mostrar proceso por consola, -f: usar el fichero descrito)

Al acabar creará un directorio llamado hadoop-2.7.2 donde se encontrarán todos los binarios de Hadoop. Una buena práctica es que este contenido se encuentre en la carpeta /opt. Entonces, crearemos una nueva carpeta llamada hadoop y moveremos el contenido desde el directorio de descargas al recién creado. Para conseguir esto es necesario o ser el usuario root (`su - root`) o tener permisos sudo (`sudo <comando a ejecutar>`).

Creamos un nuevo directorio:

```
mkdir /opt/hadoop
```

Asignamos permisos al usuario hadoop para que pueda trabajar y configurar el software sin necesidad de ser root:

```
chown hadoop /opt/hadoop
```

Movemos el contenido descargado al nuevo directorio:

```
mv /home/hadoop/Descargas/hadoop-2.7.2/* /opt/hadoop/
```

Entre el contenido que acabamos de desempaquetar y mover encontraremos los siguientes directorios:

- Directorio `bin/`: dentro podremos encontrar los scripts y comandos más importantes para realizar la gestión de Hadoop, por ejemplo: los comandos `hdfs`, `hadoop` o `yarn`; que se explicarán más adelante.
- Directorio `etc/`: es aquí donde encontraremos todos los ficheros referidos a la configuración de Hadoop.
- Directorio `lib/`: contiene librerías de Hadoop.
- Directorio `libexec/`: podremos encontrar comandos y scripts para configurar los diferentes servicios.
- Directorio `sbin/`: scripts de funcionalidades y utilidades como arrancar o parar los servicios.
- Directorio `share/`: contiene paquetes de Hadoop, librerías, ejemplos, scripts, etc.

El siguiente paso para tener el entorno perfectamente configurado que nos permita trabajar mejor y hacer que Hadoop pueda funcionar sería añadir las variables de entorno adecuadas. Las variables de entorno que posee cada usuario se pueden encontrar, en CentOS, en el fichero `.bashrc` en el directorio raíz del usuario. En el caso en el que existan más usuarios que se dispongan a utilizar Hadoop, será necesario añadir las variables de entorno al fichero `/etc/bashrc` en CentOS para que todos los usuarios obtengan esta configuración. Las variables que añadir en alguno de estos ficheros son las siguientes:

```
export HADOOP_HOME=/opt/hadoop
export JAVA_HOME=/usr/java/jdk1.8.0_161
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

En la primera línea hemos añadido el directorio donde se encuentra Hadoop instalado; en la segunda línea hemos creado la variable `JAVA_HOME` con la dirección donde se encuentra instalada nuestra versión de Java para que Hadoop pueda acceder a dicho directorio; y, por último, hemos añadido a la variable `PATH`[1] los diferentes comandos y scripts de Hadoop que se encuentran en los directorios `bin` y `sbin` de Hadoop.

Una vez editado y guardado el contenido debemos reiniciar la consola para aplicar los cambios. Tras esto, podremos ejecutar el comando `hadoop version` a modo de

[1] Se trata de la variable que contiene las rutas donde se hace la búsqueda de ejecutables

prueba para comprobar que todo se encuentra en buen estado y bien configurado hasta el momento.

En este momento incluso se pueden hacer pruebas con un script de ejemplo de MapReduce que se encuentra en el archivo `share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.0.jar`[1] dentro de Hadoop. Para ejecutar un `.jar` en Hadoop con MapReduce se usa el siguiente comando:

```
hadoop jar <archivo.jar> <parámetros>
```

Los resultados de una ejecución en Hadoop o Spark normalmente se guardan en un directorio, que contiene los siguientes ficheros:

- El fichero vacío _SUCCESS que simplemente indica que se ha finalizado correctamente la ejecución.
- Un conjunto de ficheros con la forma `part-r-N` o `part-N`, donde la N es el número, con 5 dígitos, de la partición de los resultados en el caso en el que los resultados estén particionados por la naturaleza de Hadoop (p.ej.: `part-00000`, `part-r-00000`).

[1] Para listar las clases y scripts contenidos se debe de usar el comando "`jar tf <archivo.jar>`"

2.3.5. CLÚSTER PSEUDO DISTRIBUIDO

Un clúster pseudo distribuido es aquel en el que en una sola máquina conviven diferentes demonios simulando diferentes máquinas en una sola. En nuestro caso, encontraremos los procesos maestro y esclavo de HDFS y YARN.

2.3.5.a. Configuración HDFS

Apache Hadoop, como se ha explicado, posee sus ficheros de configuración en el directorio `hadoop/etc/hadoop/`. Los más importantes para configurar la estructura son:

- `core-site.xml`: fichero que contiene la configuración general del clúster.
- `hdfs-site.xml`: fichero que contiene la configuración del sistema de ficheros HDFS.
- `mapred-site.xml`: fichero que contiene la configuración de los procesos MapReduce.
- `yarn-site.xml`: fichero que contiene la configuración de los procesos YARN.

En el caso del fichero core-site.xml añadiremos la siguiente configuración:

```
<configuration>
      <property>
            <name>fs.defaultFS</name>
            <value>hdfs://nodo1:9000</value>
            <description>NameNode URI.</description>
      </property>
<configuration>
```

Como se puede observar, el fichero debe de comenzar y acabar por la etiqueta `<configuration>` y añadimos la propiedad (`<property>`) `fs.defaultFS` que identifica el sistema de archivos que se utilizará por defecto. Usaremos el que habitualmente se emplea, que no es otro que el propio de Apache Hadoop, el HDFS. Se asigna a la propiedad el valor que indica dónde se encuentra el servidor maestro de HDFS, el NameNode, que se encuentra en el propio nodo1, en el puerto 9000. Para nuestro clúster pseudo distribuido no es necesario añadir ninguna configuración más.

En el caso del fichero hdfs-site.xml añadiremos la siguiente configuración:

```
<configuration>
      <property>
            <name>dfs.replication</name>
            <value>1</value>
      </property>
      <property>
            <name>dfs.namenode.name.dir</name>
            <value>/datos/namenode</value>
      </property>
      <property>
            <name>dfs.datanode.data.dir</name>
            <value>/datos/datanode</value>
      </property>
</configuration>
```

Como se ha explicado anteriormente, HDFS replica los bloques de los ficheros 3 veces por defecto, pero al ser un clúster pseudo distribuido sólo dispondremos de una máquina, por lo que debemos de indicar que HDFS no replique los datos. Vemos que para ello hemos añadido la propiedad `dfs.replication` indicando el valor 1.

Además, la pareja de propiedades `dfs.namenode.name.dir` y `dfs.datanode.data.dir` la incluimos para indicar en dónde se van a encontrar los datos que manejará HDFS, ya sean los metadatos del nodo maestro o los datos en los esclavos. Todo ello se almacenará en los directorios `/datos/namenode` y `/datos/datanode`, respectivamente.

En la carpeta `namenode/`, como su nombre indica, servirá para guardar los metadatos del Namenode. En ella se creará un subdirectorio llamado `current/` donde se encontrarán una serie de ficheros donde se guardan los cambios que se van produciendo dentro de los metadatos (`edits_000xxx`), otro fichero para guardar lo que se están escribiendo los datos en cierto momento (`edits_inprogress_000xxx`) y los ficheros donde se guardan *snapshots* automáticos del sistema de ficheros (`fsimage_000xxx`).

El directorio `datanode/` en los nodos esclavos contendrá todos los datos que introduzcamos en el sistema de ficheros. Contiene, igual que el directorio `namenode/`, un subdirectorio `current/`, en el cuál podremos ver diferentes directorios llamados con los IDs de los bloques pertenecientes a los ficheros dentro de HDFS.

A continuación, debemos de crear los directorios que acabamos de mencionar con el comando `mkdir`[1]. También tenemos que cambiar el propietario de los directorios creados con el comando "`chown -R <usuario>:<usuario> /datos`". Ahora será necesario crear el sistema de archivos dentro de estos directorios, para ello utilizaremos el comando hadoop que se encuentra en la carpeta `bin/` de Apache Hadoop.

```
hadoop namenode -format
```

Lo que hará este comando es obtener el directorio del NameNode del fichero hdfs-site.xml y crear el sistema de ficheros con los metadatos. A medida que se vayan añadiendo ficheros a HDFS irá creando los bloques en el directorio correspondiente. Es por ello por lo que no hace falta formatear el directorio para los datos.

2.3.5.b. Arrancar y parar HDFS

Ya hemos configurado los ficheros necesarios, creado los directorios y formatearlos para HDFS. Ahora podremos arrancar el servicio con el siguiente comando desde el nodo maestro:

```
start-dfs.sh
```

Este comando, que se encuentra en `hadoop/sbin/` arrancará el sistema de datos, el NameNode, el Secundary NameNode y el DataNode, todos ellos en el mismo nodo1.

Para ver si se están ejecutando podemos utilizar los siguientes comandos que mostrarás todos los procesos de Java que se están llevando a cabo:

```
ps -ef | grep "java"
jps -l
```

Cuando arrancamos HDFS, a parte de los servicios mencionados, se arranca un portal web al cuál se accede mediante el puerto 50070 en la versión de Hadoop 2.7 o el 9870 en la versión 3.1. En esta página web se puede consultar toda la información, configuración, estado y datos de HDFS. En el capítulo Entorno web HDFS se hace un recorrido por sus funcionalidades.

[1] Será necesario ser usuario root o tener permisos sudo para crear un directorio en la raíz del sistema.

Para parar todos los servicios que soportan HDFS se debe de usar el siguiente comando desde el nodo maestro:

```
stop-dfs.sh
```

2.3.5.c. Configuración YARN

Disponiendo del sistema de ficheros HDFS a pleno funcionamiento, es el momento de realizar la configuración del servicio de procesamiento de datos YARN.

Como ya sabemos, los archivos de configuración se encuentran en la carpeta `etc/hadoop` de Apache Hadoop. En ella encontraremos dos ficheros importantes para YARN: `yarn-site.xml` y el `mapred-site.xml.template`. Ambos ficheros, como se verá a continuación, tienen una estructura similar a la ya vista en los ficheros de configuración de HDFS, con una etiqueta global `<configuration>` y diferentes propiedades dentro.

En el fichero `yarn-site.xml` deberemos de introducir tres diferentes propiedades:

```
<configuration>
      <property>
            <name>yarn.resourcemanager.hostname</name>
            <value>nodo1</value>
      </property>
      <property>
            <name>yarn.nodemanager.aux-services</name>
            <value>mapreduce_shuffle</value>
      </property>
      <property>
            <name>yarn.nodemanager.aux-
            services.mapreduce.shuffle.class</name>
            <value>org.apache.hadoop.mapred.ShuffleHandler</value>
      </property>
<configuration>
```

En la primera propiedad se indica a YARN cuál de las máquinas va a ser el ResourceManager, es decir, el maestro. En la segunda propiedad, se indica qué servicios auxiliares van a gestionar el Map Reduce, y en la tercera dónde se encuentra dentro de las librerías de Apache Hadoop.

Las propiedades añadidas funcionan correctamente para la versión de Hadoop 2.7, pero para la versión 3.1, en ocasiones se generan algunos errores al correr aplicaciones o arrancar Yarn. Para solucionarlo, debemos añadir la siguiente propiedad que permite a Yarn conocer dónde se encuentran las librerías que necesita (adaptar al directorio donde está instalado):

```
<property>
      <name>yarn.application.classpath</name>
            <value>
                  /.../hadoop/etc/hadoop,
                  /.../hadoop/share/hadoop/common/*,
                  /.../hadoop/share/hadoop/common/lib/*,
                  /.../hadoop/share/hadoop/hdfs/*,
                  /.../hadoop/share/hadoop/hdfs/lib/*,
                  /.../hadoop/share/hadoop/mapreduce/*,
                  /.../hadoop/share/hadoop/mapreduce/lib/*,
                  /.../hadoop/share/hadoop/yarn/*,
                  /.../hadoop/share/hadoop/yarn/lib/*
      </value>
</property>
```

El fichero `mapred-site.xml.template` como se puede ver, es un fichero que sirve de modelo, pero no tiene ningún efecto en la configuración. Para que pueda ser considerado debemos de copiarlo en el mismo directorio y quitarle la última parte de su nombre.

```
cp /…/mapred-site.xml.template /…/mapred-site.xml
```

En el fichero nuevo deberemos de introducir la siguiente propiedad, que va a indicar que el motor de gestión del procesamiento de datos va a ser el YARN:

```
<configuration>
      <property>
            <name>mapreduce.framework.name</name>
            <value>yarn</value>
      </property>
<configuration>
```

2.3.5.d. Arrancar y parar YARN

Una vez hayamos completado la configuración anterior, podremos arrancar el YARN sin ningún problema. Es obligatorio, antes de arrancar YARN, arrancar HDFS:

```
start-dfs.sh

start-yarn.sh
```

Tras esta operación podremos utilizar el comando `jps`, para observar todos los procesos Java corriendo en la máquina[1]. Con él podremos observar que están activos los tres procesos de HDFS: el NameNode, el DataNode y el Secundary NameNode. Y los procesos de YARN: el ResourceManager y el NodeManager.

Cuando arrancamos Yarn, a parte de los servicios mencionados, se arranca un portal web al cuál se accede mediante el puerto 8088 en la versión de Hadoop 2.7. En esta página web se puede consultar toda la información, configuración y estado de los servicios Yarn, así como la información de las aplicaciones que se están ejecutando. En el capítulo Entorno web Yarn se hace un recorrido por sus funcionalidades.

A la hora de parar Yarn utilizaremos el siguiente comando:

```
stop-yarn.sh
```

2.3.5.e. Instalación Spark

En este capítulo veremos cómo instalar Apache Spark en nuestra máquina y conseguir que trabaje con Hadoop, tanto con los datos, HDFS, como sobre Yarn. Comenzaremos accediendo a la sección de descargas de la página oficial de Apache Spark y descargando el paquete que deseemos según nuestras necesidades. Como se ha explicado, Spark es capaz de trabajar sobre múltiples gestores de recursos de un clúster e incluso por separado en podo *standalone*, realizando por sí mismo la gestión de los recursos. En nuestro caso decidiremos descargar la última versión[2] disponible sin Apache Hadoop ya que ya lo tenemos instalado en la máquina (opción: "*Pre-build with user-provided Apache Hadoop*").

Esta descarga dejará un archivo comprimido en nuestra máquina que deberemos de descomprimir el contenido en nuestro directorio de Hadoop.

[1] Aparecen todos los procesos, tanto esclavos como maestros ya que estamos en una configuración pseudo distribuida. En un entorno distribuido aparecerían solo los procesos Maestros o esclavos, dependiendo en qué máquina nos situemos.

[2] En el momento de realización de este trabajo de fin de grado, la versión última de Spark era la versión 2.3.0.

```
tar -xvf spark-2.3.0-bin-without-hadoop.tgz -C /.../hadoop/
```

Cambiamos el nombre de la carpeta para poder trabajar mejor:

```
mv /.../hadoop/spark-2.3.0-bin-without-hadoop /.../hadoop/spark
```

Si nos desplazamos hasta el directorio de Spark podremos observar en su interior los siguientes directorios más importantes:

- `bin/`: contiene diferentes ejecutables para trabajar con Apache Spark.
- `sbin/`: contiene scripts para arrancar o parar los diferentes servicios.
- `conf/`: contiene los ficheros para la configuración de la tecnología.
- `examples/`: contiene diferentes ejecutables de ejemplo.
- `jars/`: contiene librerías necesarias para ejecutar el servicio.
- `yarn/`: contiene el fichero jar necesario para ejecutar una característica llamada Dynamic Resource Allocation que se explica más adelante.

A continuación, prepararemos las variables de entorno necesarias para trabajar con Spark y permitir que funcione correctamente. Añadiremos las siguientes líneas a nuestro fichero de variables de entorno:

```
export SPARK_HOME=/opt/hadoop/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export SPARK_DIST_CLASSPATH=$(hadoop classpath)
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

Vemos que hemos añadido una variable indicando el directorio donde se encuentra Spark, hemos añadido al PATH los directorios de scripts de Spark, hemos creado una variable que contiene todas las librerías asociadas a Hadoop para permitir el funcionamiento de Spark sobre él y, por último, creado una variable para que Spark conozca dónde se encuentran los ficheros de configuración de Hadoop.

Para comprobar que Spark funciona correctamente o probar código podremos utilizar unas herramientas que proporciona Spark que nos permiten tener una consola de Scala, Python, SQL o R. Estas herramientas se encuentran en el directorio `bin/` de Spark y podremos probar comandos en Scala con el script `spark-shell`, en Python con el `pyspark`, en R con el `sparkR`, etc. Estas consolas, cuando se ejecutan generan un SparkContext en la máquina local y en donde podremos ejecutar comandos y probar nuestro código sin necesidad de lanzar una aplicación contra la tecnología.

2.3.6. CLÚSTER VIRTUAL DISTRIBUIDO

Tras haber instalado y configurado una máquina para trabajar en modo pseudo distribuida, nos dispondremos a crear un clúster virtual en base a esta. Los pasos que se redactarán a continuación comenzarán con la configuración de las redes, la clonación de la máquina, la configuración de los servicios y la realización de pruebas de que todo funcione correctamente.

La razón por la que clonamos la máquina virtual es simplemente debido a que en la que tenemos ya están todos los componentes instalados y configurados para que funcionen. Si no las clonamos deberíamos de realizar todos los pasos anteriores de nuevo, con la pérdida de tiempo que ello implica.

2.3.6.a. Red interna

Antes que nada, deberemos de apagar la máquina, porque el software de virtualización VirtualBox no permite la clonación en caliente y tampoco tenemos la necesidad de tenerla encendida durante esta fase en la que realizaremos algunas configuraciones previas sobre ella.

Una vez apagada la máquina abriremos en VirtualBox el menú de configuración e iremos hasta la sección de Red. Podremos observar que dispone de un adaptador de red de tipo NAT, que es el que se asigna por defecto a las máquinas creadas. Crearemos ahora una nueva tarjeta de red en el adaptador 2 clicando en la casilla de “Habilitar adaptador de red” y de tipo “Red interna”. Esta red nueva será una red aislada en la que Hadoop se conectará con las diferentes máquinas virtuales que estén en ella. Aceptamos y quedaría guardada la nueva configuración.

2.3.6.b. Clonación de la máquina virtual

Ya con la nueva red interna creada, clonaremos la máquina virtual que tenemos, la clonación guardará la máquina en el directorio que tenga especificado VirtualBox en sus preferencias, por si se desea cambiarlo. El proceso de clonación se realizará el número de veces que se requiera. En mi caso sólo voy a tener 3 máquinas virtuales por las capacidades de mi ordenador.

Ahora, clicamos con el botón derecho y seleccionamos la opción clonar. Aparecerá una ventana donde indicamos el nuevo nombre de la máquina (nodo2 y nodo3) y seleccionamos la casilla para reinicializar la dirección MAC, porque las máquinas van a trabajar en la misma red y al mismo tiempo y pondría las mismas IPs para ellas. Además, la clonación debe de ser completa.

2.3.6.c. Configuración de las máquinas

Ya con las máquinas clonadas, las arrancamos todas y nos dispondremos a configurar los elementos necesarios. Primero, cambiaremos los nombres de las creadas a "nodo2" o "nodo3", ya que todas se llaman actualmente "nodo1".

```
hostname nodoX
```
[1]

Este comando permite el cambio de nombre de forma no permanente, para que el cambio perdure después de reiniciar la máquina debemos cambiar el fichero `/etc/sysconfig/network`[2]. En la etiqueta "HOSTNAME" cambiaremos el nombre al nuevo según la máquina.

Como las máquinas tienen que conectarse entre sí, es necesario que todas conozcan las direcciones IP de las demás. Para ello añadiremos las siguientes líneas al fichero `/etc/hosts`:

```
192.168.0.101        nodo1
192.168.0.102        nodo2
192.168.0.103        nodo3
```

Ahora deberemos conectar las máquinas a dicha red. Para ello lo podemos hacer mediante la consola y editando los ficheros de configuración, o mediante el entorno gráfico. La interfaz de red a modificar será la correspondiente con la red interna. Al ser la segunda en aparecer, tendrá el nombre de "eth1". Por consola debemos ir al fichero `/etc/sysconfig/network-scripts/ifcfg-eth1` para cambiar su contenido y que disponga de la IP estática que se configuró en el archivo de hosts. Por interfaz gráfica clicamos con el botón derecho en las redes, seleccionamos "editar las conexiones", editamos la red "eth1" y en los parámetros IPv4 seleccionamos la configuración manual añadiendo la IP correspondiente.

[1] Es necesario ser usuario root o disponer de permisos sudo.
[2] Forma válida para CentOS.

Para probar que está bien configurada la red, podremos realizar el comando "`ping nodoX`" desde las diferentes máquinas para ver si conectamos con las unas y las otras.

El siguiente paso en la configuración de las máquinas será que se permitan las conexiones a través de SSH con certificado, es decir, sin que se pida la contraseña, para que Hadoop y Spark trabajen correctamente. Como se han clonado las máquinas, en todas ellas está el certificado de la primera, por lo que realizaremos el siguiente grupo de comandos en cada máquina para generar nuevos certificados y enviar los públicos entre los nodos. Antes que nada, eliminaremos toda la información que exista de cada nodo con respecto a los certificados y crearemos un certificado nuevo para cada máquina:

```
cd ~/.ssh/
rm *
ssh-keygen
```

Por último, se debe de enviar la clave pública de cada nodo a todos los demás, incluido a sí mismo.

```
ssh-copy-id -i id_rsa.pub <usuario>@nodo1
ssh-copy-id -i id_rsa.pub <usuario>@nodo2
ssh-copy-id -i id_rsa.pub <usuario>@nodo3
```

Tras esto, si realizamos una conexión SSH desde cualquier nodo a cualquier otro, no debería de pedirse contraseña, y estaría todo bien configurado.

2.3.6.d. Configuración Apache Hadoop

Como vimos en la arquitectura de clúster pseudo distribuido, HDFS, hace uso de dos directorios para almacenar los datos, en nuestro caso los llamamos: `/datos/datanode` y `/datos/namenode`. En aquella estructura los servicios maestros y esclavos convivían en la misma máquina, pero esto ya no va a ser así, ahora poseeremos un nodo maestro y dos esclavos. Esto quiere decir que deberemos de eliminar el directorio `datanode/` en el nodo1, el maestro, y los directorios `namenode/` en los nodos 2 y 3, los esclavos.

```
nodo1 $: rm -rf /datos/datanode
nodo2/3 $: rm -rf /datos/namenode
```

Otro paso que realizar para evitar problemas sería eliminar el contenido de todos los directorios restantes, ya que realizaremos de nuevo el formateo de los mismos y así evitamos problemas que puedan aparecer.

```
nodo1 $ rm -rf /datos/namenode/*
nodo2/3 $ rm -rf /datos/datanode/*
```

Lo que debemos de hacer ahora será actualizar los ficheros de configuración de HDFS para que la tecnología contemple que dispone de los nuevos nodos para funcionar. Estos ficheros, recordamos que se encuentran en el directorio `/opt/hadoop/etc/hadoop/`. El fichero que habrá que modificar con respecto a la instalación para clúster pseudo distribuido será el fichero `hdfs-site.xml`. En él cambiaremos las propiedades para que ahora los datos se repliquen dos veces, ya que poseemos 2 nodos esclavos y no se podría replicar más que eso. El resto de las propiedades seguirán siendo las mismas:

```
<property>
     <name>dfs.replication</name>
     <value>2</value>
</property>
```

Tras haber cambiado el fichero, debemos de copiarlo a los otros nodos para que estos dispongan de la misma configuración:

```
scp hdfs-site.xml nodo<2,3>:/opt/hadoop/etc/hadoop/
```

El último fichero que modificar será el llamado `slaves`, en la versión 2.7 de Hadoop, o el llamado `workers`, en la versión 3.1. En ellos añadiremos el nombre de cada nodo que vaya a funcionar como DataNode de HDFS y lo copiamos en las otras máquinas:

```
$: cat slaves
     nodo2
     nodo3
$: scp slaves nodo<2,3>:/opt/hadoop/etc/hadoop/
```

El resto de los ficheros que se configuraron en la sección de instalación del clúster pseudo distribuido quedarán igual.

Ahora, que tenemos toda la configuración de HDFS implementada, formatearemos como se hizo anteriormente el NameNode:

```
hdfs namenode -format
```

Por defecto, en las máquinas estará funcionando el Firewall Iptables, por lo que será necesario permitir las conexiones de los nodos entre sí. Ya que todos se comunicarán a través de la red interna, y esta es inaccesible desde fuera, permitiremos todo tipo de conexión en ella. Usaremos el siguiente comando en cada nodo para aceptar todas las conexiones entrantes por la interfaz “eth1”, la perteneciente a la red interna:

```
iptables -A INPUT -i eth1 -j ACCEPT
```

2.3.6.e. Prueba de funcionamiento

Para probar que todo funciona correctamente podremos iniciar todos los servicios y ver que funcionan correctamente:

```
start-dfs.sh
start-yarn.sh
```

Si todo va bien podremos observar como los diferentes procesos se ejecutan en las diferentes máquinas. Usando el comando `jps` en el nodo1 veremos cómo solamente están activos los procesos maestros NameNode, Secundary NameNode y ResourceManager. Y usando el mismo comando en los nodos esclavos veremos cómo están activos los procesos DataNode y NodeManager. Del mismo modo podremos abrir las webs de administración de HDFS y de Yarn para observar la información de esta nueva estructura.

Para probar Spark podremos iniciar una consola de Scala, por ejemplo, y probar con comandos Scala y con lecturas desde HDFS para ver que todo funciona correctamente. Como se ha visto, la configuración para Spark ha sido prácticamente nula en el cambio de arquitectura. Esto se debe a que como se ha explicado, cuando lancemos aplicaciones Spark sobre el clúster Hadoop, va a ser Yarn el que organice todos los recursos, esto implica que para Spark, la estructura sobre la que se ejecute no le va a afectar porque solamente debe de comunicarse con Yarn y Yarn hacer todo el trabajo de gestión.

2.3.6.f. Spark Dynamic Resource Allocation

Existe un mecanismo para, dinámicamente, ajustar los recursos que una aplicación necesita según su carga de trabajo, ya que Yarn utiliza MapReduce y como hemos visto no es tan efectivo y las aplicaciones que se van a lanzar en el futuro son de tipo Spark. Este mecanismo se llama Dynamic Resource Allocation y permite pedir o devolver recursos del clúster cuando sean o no necesarios. Es una opción bastante interesante cuando existen varias aplicaciones compartiendo recursos.

Esta función se encuentra deshabilitada por defecto, y debe de configurar para trabajar con YARN. Entonces, debemos de configurar un servicio Shuffle externo al de Yarn en todos los nodos esclavos, los NodeManagers. Para ello deberemos de acceder al fichero yarn-site.xml y añadir las siguientes propiedades que indican cuál va a ser el servicio Shuffle y dónde se encuentra la librería para ejecutarlo:

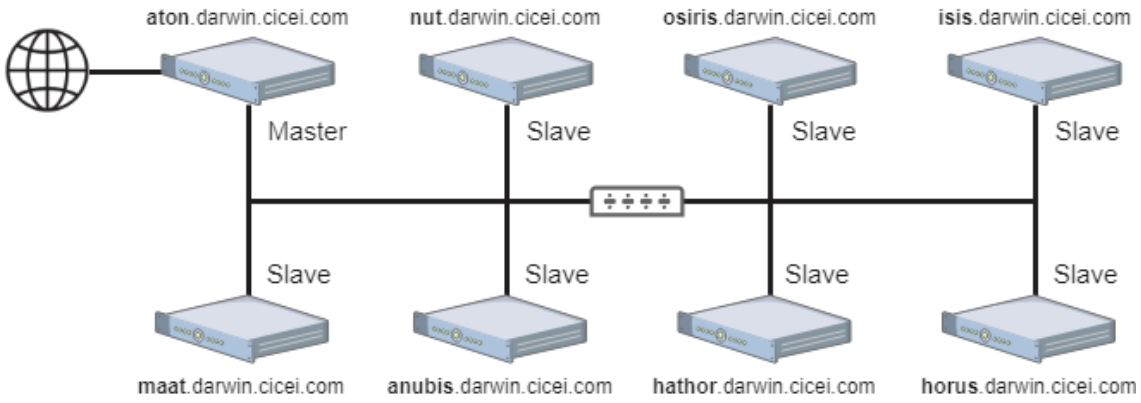
```
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>spark_shuffle</value>
</property>
<property>
    <name>yarn.nodemanager.aux-services.spark_shuffle.classpath</name>
    <value>/opt/hadoop/spark/yarn/*</value>
</property>
<property>
    <name>yarn.nodemanager.aux-services.spark_shuffle.class</name>
    <value>org.apache.spark.network.yarn.YarnShuffleService</value>
</property>
```

Después de guardar estos cambios reiniciamos YARN y ya podríamos ejecutar aplicaciones con esta configuración. A la hora de ejecutarlas con el comando `spark-submit`, añadiremos las configuraciones de `spark.dynamicAllocation.enabled` a `true` y `spark.shuffle.service.enabled` a `true`. Si no las incluimos seguiríamos trabajando como por defecto dejando a YARN utilizar el Shuffle de MapReduce.

2.3.7. CLÚSTER DISTRIBUIDO CICEI

Como último paso de instalación y configuración necesario antes de comenzar a trabajar con los datos será pasar todo lo aprendido al conjunto de máquinas reales situadas en el CICEI. El proceso de realización que se llevó a cabo no fue más distinto que el explicado con antelación, de modo que en este capítulo no se explicará cada paso, sino las características y detalles de la implementación de las tecnologías y su funcionamiento.

El clúster, situado dentro de la red interna del CICEI, posee 8 máquinas con 2 núcleos de procesamiento y 8 GB de RAM cada una (más información aquí). La red tiene la estructura que se muestra en el siguiente diagrama, básicamente están todos los nodos interconectados a través de una red interna propia, siendo el nodo llamado `aton.darwin.cicei.com` el único con acceso al exterior.



6 Red interna del clúster. Fuente: elaboración propia.

Desde el exterior al clúster se le consultará mediante el nombre de `iguana.cicei.com`, que redirigirá la conexión hacia `aton`. El nodo `aton`, como se muestra, es el elegido para correr todos los servicios maestros de Apache Hadoop y Apache Spark, tal y como se hizo con el nodo1 en el clúster virtual montado en secciones anteriores.

A continuación, se resumen los pasos que se llevarán a cabo para configuración del clúster:

1. Instalación del sistema operativo: todas las máquinas se les realizará la instalación del sistema operativo Ubuntu Server. Además, se creará el mismo usuario en todos los nodos que los servicios funcionen correctamente y se declararán los nombres y direcciones IPs que tendrán los demás nodos dentro de la red interna.

2. Configuración de la red: asignaremos las IPs estáticas de la red interna a todas las máquinas y configuraremos la conexión al exterior de aton. Será necesario también, aceptar las conexiones de la red interna y capar toda posible conexión exterior al clúster, salvo si el origen es desde dentro de la red del CICEI, donde aceptaremos las conexiones a los puertos de los servicios web de HDFS, YARN y Spark.

3. Instalación y configuración de Java y SSH: como vimos Hadoop se ejecuta en Java, lo que implica que deberemos instalar los JRE de Java en todas las máquinas. También preparar la configuración de SSH con los certificados para que se pueda acceder desde todos los nodos hacia todos los nodos sin la necesidad de introducir contraseña.

4. Descarga de los paquetes de Apache Hadoop y Apache Spark en aton únicamente.

5. Creación de los directorios donde se instalarán y los que utilizarán, como el del NameNode.

6. Extracción de los paquetes y configuración de los ficheros correspondientes para el funcionamiento, así como las variables de entorno.

7. Copiar el contenido configurado al resto de máquinas. Importante que la distribución de directorios sea la misma en todos los nodos.

Si todos los pasos se han hecho correctamente, podremos iniciar HDFS y YARN y comprobar que todo es correcto. Podremos subir archivos a HDFS, operar con ellos con los comandos y ejemplos de Yarn y también utilizar la consola de Scala de Spark.

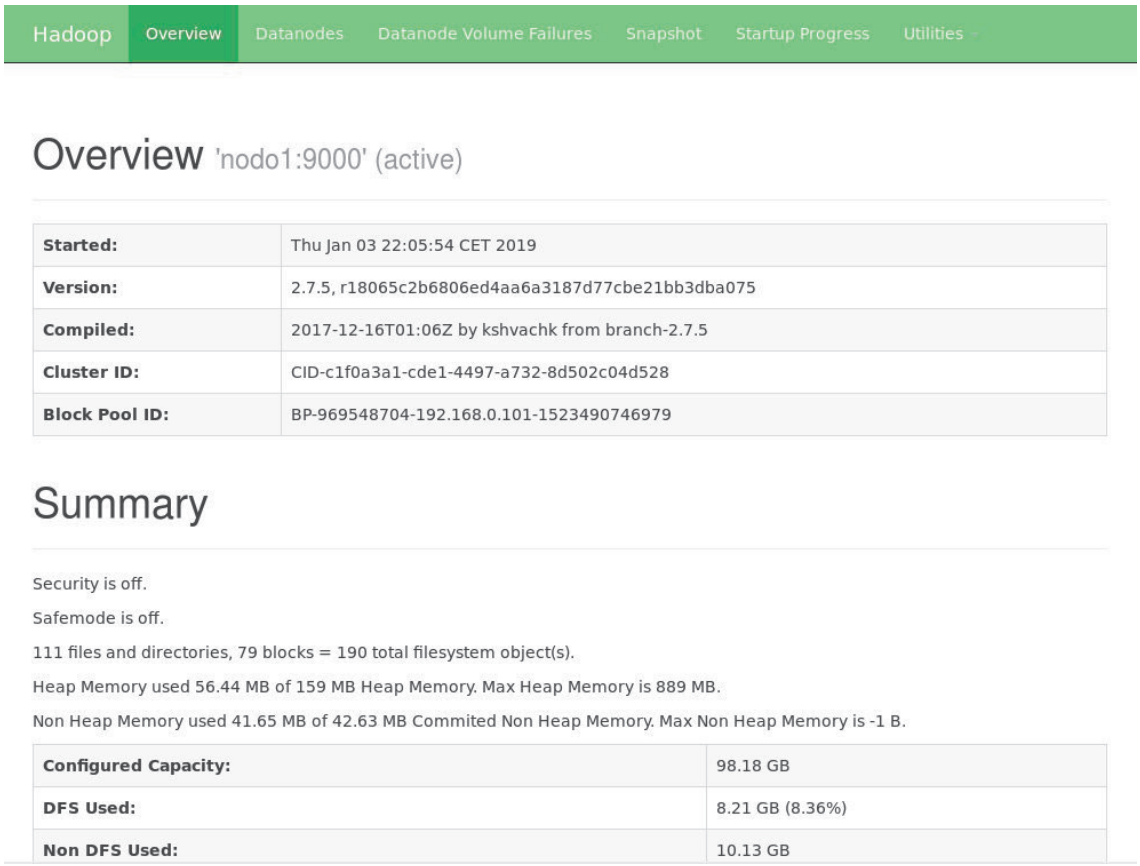
En resumen, la configuración del clúster resulta en un grupo de máquinas idénticas con los mismos usuarios; mismos permisos; misma estructura de directorios; con conexiones sin contraseña por SSH; y misma configuración de Hadoop, Yarn y Spark[1].

[1] Las únicas diferencias serán los directorios de HDFS, el de los metadatos en el nodo maestro y el de los datos en los nodos esclavos; y la configuración extra del Spark Dynamic Allocation que únicamente se añade en los NodeManagers.

2.4. UTILIDADES

2.4.1. ENTORNO WEB HDFS

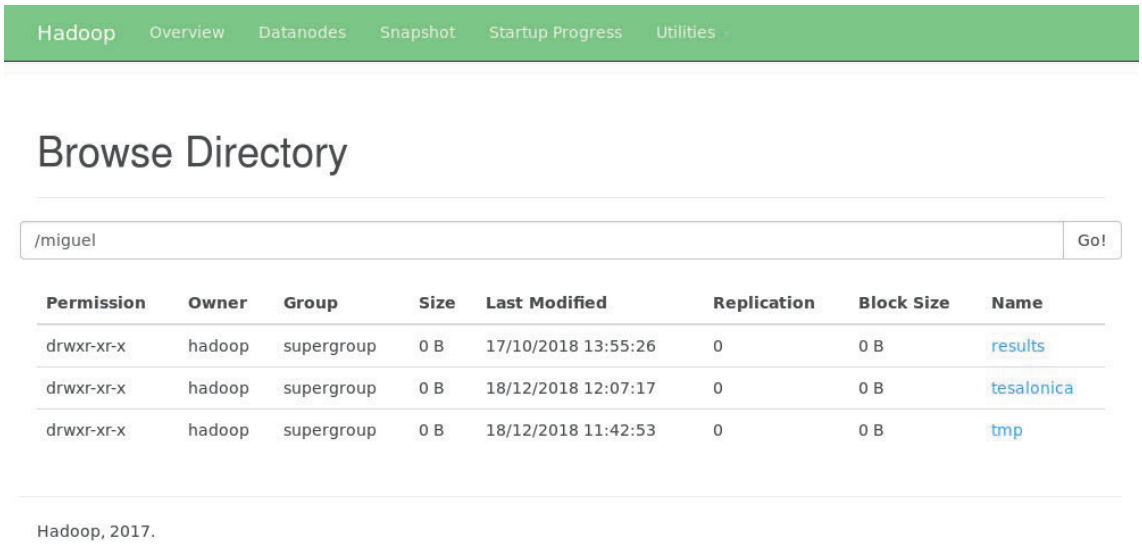
El sistema de ficheros de Hadoop ofrece un portal web al que acceder para consultar todo tipo de información sobre la configuración y estado del clúster. En la página de inicio podremos ver, como se muestra en la imagen, un resumen de la información de la versión actual del sistema, así como el ID o cuándo fue arrancado. La siguiente información que podemos consultar en esta página es el estado de la seguridad y modos de trabajo que posee HDFS. Del mismo modo, aparece el número de ficheros, de directorios y número de bloques guardados, espacio de almacenamiento disponible en la máquina, cuánto es usado por datos que no son de HDFS o qué porcentaje de uso del espacio en los DataNodes. También se añade la cantidad de nodos activos y no activos actualmente, los apartados por errores e información sobre los bloques y su replicación.



7 Portal web HDFS: Página principal. Fuente: elaboración propia

En la parte superior podemos observar un menú con las diferentes secciones siguientes:

- Overview: se trata de la página principal que contiene todo lo explicado anteriormente.
- Datanodes: como el propio nombre indica, en esta sección podremos encontrar información relacionada con los DataNodes. Podremos ver aquellos activos y no activos con información sobre el último contacto que se tuvo con ellos, estado del servicio, datos sobre el almacenamiento y la versión que ejecuta.
- Datanode Volume Failures: es una sección en donde se encontrarán, si hubiera, problemas con los volúmenes de HDFS.
- Snapshot: en este apartado se encontrarán los snapshots [1] tomados del almacenamiento.
- Startup Progress: accediendo a este apartado podremos obtener información sobre el progreso de arranque del sistema, visualizando el estado de todos los subprocesos y pasos que se ejecutan cuando se inicia HDFS.
- Utilities: este menú se trata de un desplegable con varias opciones:
    - Browse the file system: sección donde se pueden ver todos los ficheros y directorios dentro de HDFS, con la información perteneciente a cada uno: permisos, fechas, tamaño, bloques, nombre, etc.
    - Logs: sección donde se muestra una lista para ver los ficheros de log que se generan por los servicios de HDFS.

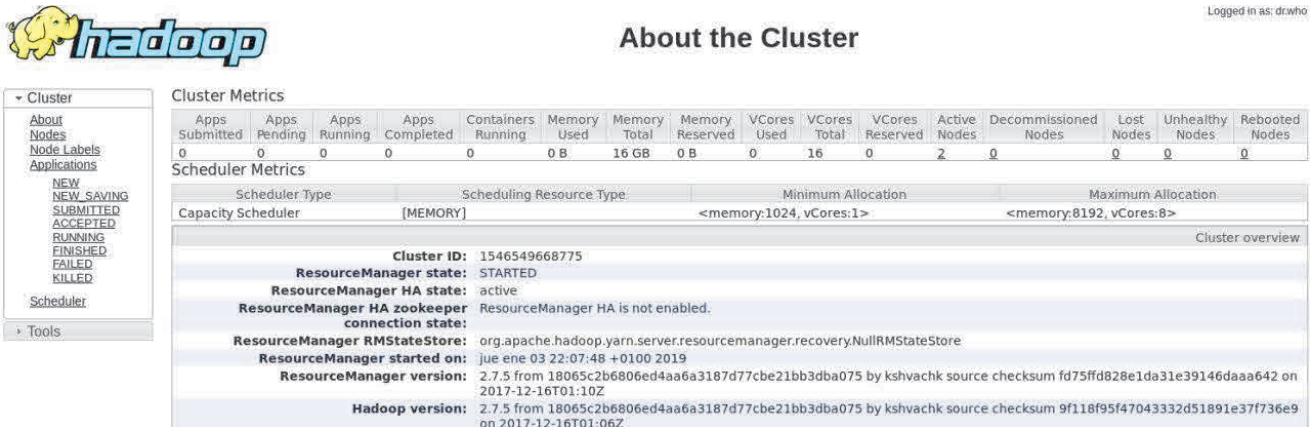


8 Portal web HDFS: directorios. Fuente: elaboración propia

[1] Un snapshot es una copia instantánea del estado del sistema en un momento determinado.

2.4.2. ENTORNO WEB YARN

Como hemos visto, en HDFS existe un portal web donde ver la configuración y estado de este servicio. Del mismo modo, YARN dispone de un portal al que podremos acceder mediante el puerto 8088.



9 Portal web Yarn: página principal. Fuente: elaboración propia

En la página principal podremos observar un menú con diferentes entradas a la izquierda y a la derecha diferentes tablas donde podremos encontrar todas las métricas del clúster y lista de aplicaciones ejecutándose o finalizadas. Entre las métricas más importantes mostradas podemos ver el número de aplicaciones y nodos en sus diferentes estados, datos sobre el uso de memoria y procesadores o número de contenedores activos.

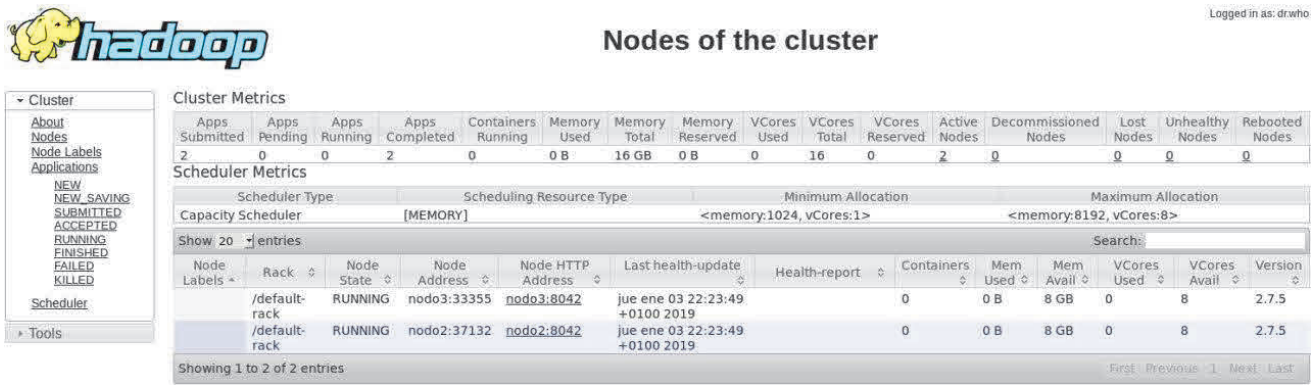
En el menú de la derecha encontramos un grupo de opciones llamado “Cluster” y otro llamado “Tools”. En el grupo “Cluster” encontraremos las siguientes opciones:

- About: sección con información general del clúster como su ID, si está activo el ResourceManager y la versión.
- Nodes: sección donde tenemos un resumen del estado de cada uno de los nodos activos dentro del clúster. Podremos clicar en el enlace que nos ofrece para cada uno de los nodos. Clicando en él podremos ver toda la información de las aplicaciones ejecutando en el propio nodo, sus configuraciones, el estado de los recursos, etc.
- Node Labels: sección dedicada a mostrar información sobre los diferentes grupos de nodos que hayamos configurado.
- Applications: sección con información general de las aplicaciones. Yarn dispone de diferentes subapartados donde podremos filtrar todas las aplicaciones según el estado en el que se encuentren. Podremos clicar en ellas para obtener más

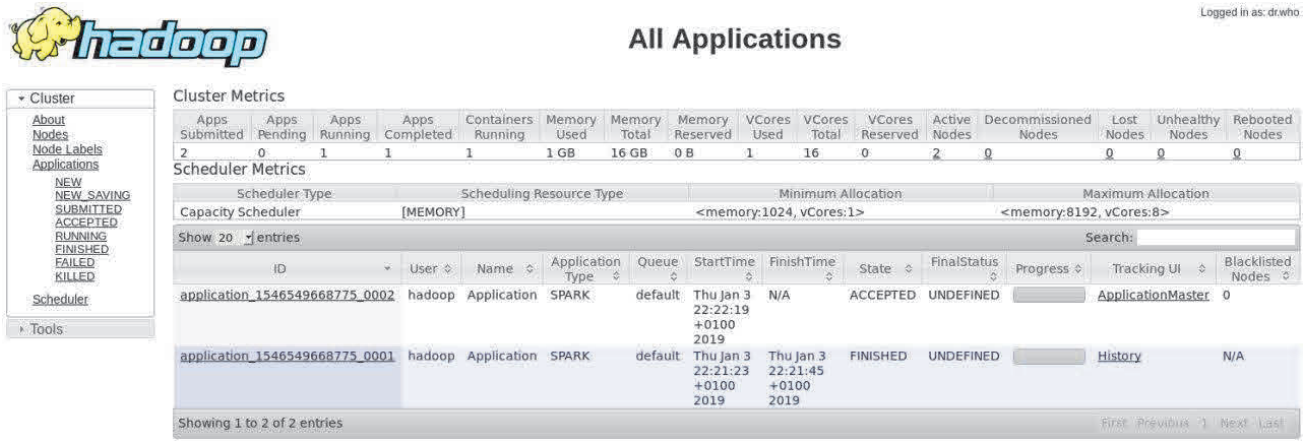
información del estado, número de intentos, logs, número de contenedores lanzados, dónde se está ejecutando dentro del clúster, etc.

- Scheduler: sección donde se muestran todas las métricas del planificador de Yarn.

En el otro grupo de opciones, “Tools”, encontramos enlaces a un XML con toda la configuración, a la lista de archivos de logs o a las métricas del servidor entre otros.



10 Portal web Yarn: Nodos. Fuente: elaboración propia



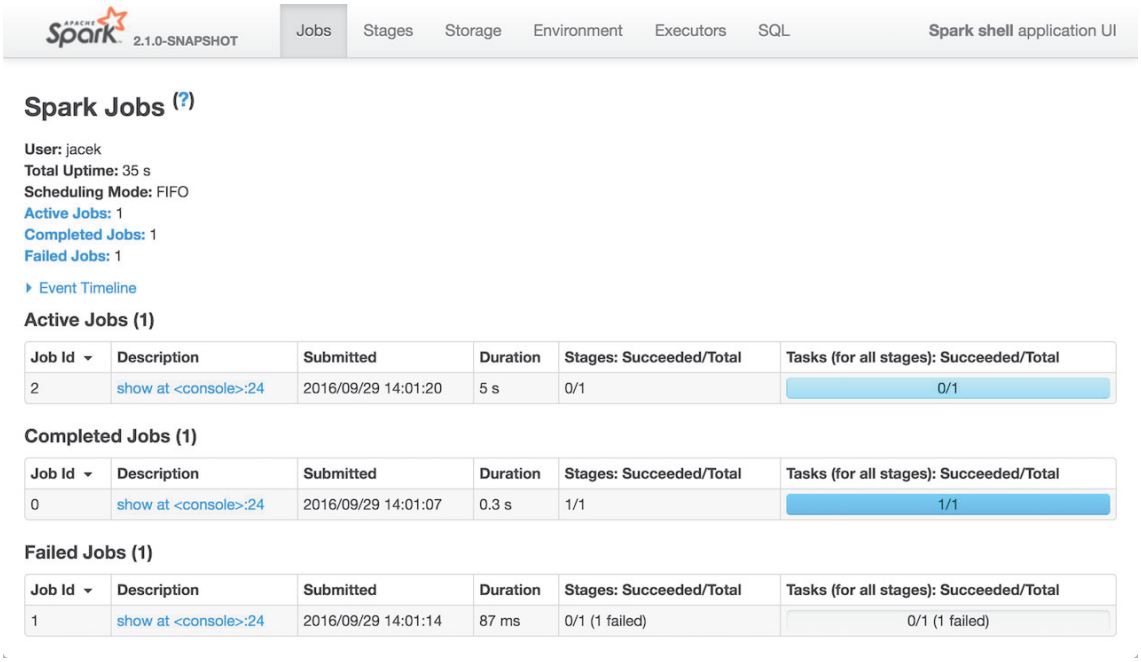
11 Portal web Yarn: Aplicaciones. Fuente: elaboración propia

2.4.3. PORTAL WEB SPARK

Apache Spark también dispone de un servicio web que se inicia cuando se arranca un clúster Spark *standalone*. En el contexto de este trabajo de fin de grado no es el caso de esta arquitectura, por lo que no dispondremos de este servicio activo siempre.

Cuando creemos aplicaciones Spark, estas se ejecutarán sobre el gestor Yarn, pero se crearán los ApplicationMaster para ellas. Al estar estos activos, también se activará la web Spark en el puerto 4040.

En esta web, accediendo desde la web de Yarn, encontraremos toda la información de la aplicación que se está ejecutando. Podremos observar los executors activos o parados, si están realizando tareas, información de las tareas como el tiempo que llevan activas o de qué tipo son, estado del espacio de memoria, información del clúster, etc. Como se ve, es bastante completa y servirá de ayuda a las aplicaciones que se ejecuten para comprobar todo tipo de información y descubrir la forma de trabajo de Spark.



12 Portal web Spark. Fuente: *Web UI – Spark Application's Web Console. jaceklaskowski.gitbooks.io*

2.4.4. COMANDOS IMPORTANTES

Apache Hadoop y Spark ofrecen múltiples comandos para realizar todo tipo de operaciones de configuración, consulta y ejecución. Desde los comandos simples de arrancar o parar procesos a comandos de creación de aplicaciones con múltiples configuraciones o de trabajo con los datos. Dividiremos esta sección en comandos según la tecnología de la que vengan.

2.4.4.a. Comandos de Hadoop

Los comandos de Hadoop podremos encontrarlos en los directorios `bin/` y `sbin/` en donde se encuentre Hadoop. Los más básicos, pero importantes, que podemos encontrar son los encargados del arranque o parada de los servicios:

```
start-dfs.sh
start-yarn.sh
stop-dfs.sh
stop-yarn.sh
```

Como el nombre indica, arrancarán el servicio del sistema de ficheros HDFS y el gestor de recursos del clúster YARN, o los pararán.

Uno de los comandos realmente profundos y con gran cantidad de opciones y parámetros es el comando `hdfs`, con el que podremos, por ejemplo, realizar operaciones en el sistema de ficheros o realizar operaciones de administración de HDFS:

```
hdfs dfs -ls /datos/
hdfs dfsadmin -report
```

A continuación, se detallarán algunos de los subcomandos y operaciones más usadas y útiles del comando `hdfs`:

- `dfs`: es el subcomando para realizar operaciones sobre el sistema de ficheros. En con él podremos introducir muchas de las operaciones básicas de los entornos UNIX como: "`-ls`", "`-cp`", "`-cat`", "`-mkdir`", etc. Dispone también del comando "`-put`", en el que se indica un fichero del sistema de archivos del sistema y se carga en HDFS en el directorio deseado. Del mismo modo, dispone del comando "`-get`", para realizar la operación inversa, descargar de HDFS un archivo al sistema.

```
hdfs dfs -cp /datos/d1 /datos/d2
hdfs dfs -put /home/miguel/tesalonica/* /datos/
hdfs dfs -get /datos/* /home/miguel/tesalonica/
```

- `dfsadmin`: es el subcomando para realizar operaciones de administración como: cambiar su estado a "`safemode`" con "`-safemode enter|leave`"; actualizar la información de los nodos por si a cambiado o a aparecido alguno más con "`-refreshNodes`"; configurar cuotas a los directorios o usuarios con "`-setQuota`" y "`-setSpaceQuota`"; obtener un resumen con toda la información sobre los recursos de HDFS con "`-report`"; o ver los nodos activos y a qué rack pertenecen con el comando "`-printTopology`".

```
hdfs dfsadmin -report
hdfs dfsadmin -safemode enter
```

- `fsck`: es un comando que permite hacer un chequeo del sistema de ficheros HDFS para comprobar si su estado es saludable.

```
hdfs fsck /datos
```

- `--daemon`: este parámetro permite ver el estado, parar o arrancar alguno de los demonios de HDFS.

```
hdfs --daemon start namenode
```

El otro comando importante de Hadoop es el comando "`yarn`" que permite gestionar y visualizar la información de la infraestructura YARN y las aplicaciones que se están ejecutando en el clúster. A continuación, se detallarán algunas de las opciones y parámetros más importantes y usadas de "`yarn`":

- `application | app`: con este comando se pueden realizar operaciones sobre las aplicaciones que se están ejecutando. Con "`-list`" veremos las aplicaciones que se están ejecutando, y añadiendo "`-appStates`" y una etiqueta de estado (`ALL, KILLED, ACEPTED, etc.`) podremos ver otras aplicaciones no aparecían antes. También se puede poner "`-kill <app ID>`" para matar una aplicación que se está ejecutando o con "`-status <app ID>`" ver su estado actual.

```
yarn application -list -appStates ALL
yarn app -kill application_00000000001_0000001
```

- `nodes`: con este comando podremos ver los nodos que se encuentran activos y la información sobre ellos.

```
yarn nodes -list
```

- container: con este comando podremos ver todos los contenedores que posee cierta aplicación que se esté ejecutando.

```
yarn container -list application_00000000001_0000001
```

- `--daemon`: este parámetro permite ver el estado, parar o arrancar alguno de los demonios de YARN.

```
yarn --daemon start resourcemanager
```

2.4.4.b. Comandos Spark

Como ya se ha explicado, Spark es capaz de trabajar sin necesidad de disponer de un gestor de recursos detrás, él se puede encargar de toda la infraestructura. Es por ello por lo que se pueden iniciar los procesos necesarios para ello, el master y los workers. Aunque esto sea ajeno a este proyecto, los comandos para arrancar y parar estos procesos son los siguientes:

```
start-all.sh
stop-all.sh
```

Apache Spark, es capaz de crear aplicaciones implementadas en múltiples lenguajes de programación como Scala, Python, SQL o R. Además, incluye consolas de estos lenguajes donde probar el código, los comandos que abren estas consolas son:

`spark-shell`	(consola Scala)
`pyspark`	(consola Python)
`spark-sql`	(consola SQL)
`sparkR`	(consola R)

A la hora de ejecutar aplicaciones reales en el clúster es necesario utilizar el comando `spark-submit`. Este comando dispone de los siguientes parámetros más importantes y usados:

- --master <master URL>: argumento para indicar cuál será el gestor de recursos del clúster sobre el que se debe de ejecutar la aplicación, en el caso de este TFG, Yarn.
- --deploy-mode: argumento para indicar dónde se ejecutará el programa driver. En el modo “client” se ejecuta localmente y en el modo “cluster” se ejecuta en cualquiera de los nodos trabajadores.
- --class: argumento para aplicaciones Java o Scala en la que se indica la clase principal del programa.
- --packages: argumento que sirve para añadir la lista de paquetes Maven que se deben de incluir para ejecutar el programa.
- --conf: argumento donde se añade o modifican configuraciones de la ejecución.
- --driver-memory: argumento que indica la cantidad de memoria máxima que usará el proceso driver.
- --executor-memory: argumento que indica la cantidad de memoria máxima que usarán los procesos executor.
- --num-executors: únicamente para ejecuciones sobre clúster Yarn, este argumento indica el número de ejecutores que se lanzan.
- <archivo ejecutable>: como último se añade el archivo ejecutable en cuestión para crear la aplicación y sus argumentos.

```
spark-submit --class "App" --master yarn
     --deploy-mode client
     --packages lib.calculations:1.2
     --conf spark.executor.memory=6g
     /project_2.11-1.0.jar -bt 3 15
```

2.5. PRUEBA DEL CLÚSTER Y ANÁLISIS DE LOS RESULTADOS

Ahora que conocemos todos los servicios y se encuentran instalados y configurados el clúster, nos dispondremos a probarlo y comprobar su funcionamiento. Debido a que las características de las máquinas es la que es, demasiado pobre para lo que se recomienda en este tipo de arquitecturas, deberemos de ir adaptándonos a los límites que este soporta.

En este capítulo veremos en detalle cada paso realizado para probar la tecnología y realizar cálculos interesantes sobre los datos.

El proceso comenzará con el análisis y formateo de los datos para obtener grafos de ellos, carga de los datos en HDFS, creación del código para el cálculo de diferentes métricas, descarga de resultados y visualización del comportamiento del clúster y, por último, representación en R en mapas y gráficos.

2.5.1. LOS DATOS

El dataset sobre el que vamos a trabajar se compone de posiciones de taxis en la ciudad griega de Tesalónica a lo largo del día durante prácticamente un año. Esta información se divide en ficheros de texto de 225 MB de media, haciendo un total de 78,4 GB entre 349 archivos. Como queda demostrado, es un dataset lo suficientemente grande como para ser considerado Big Data, ya que hay que tener en cuenta que se trata de 78,4 GB de texto plano únicamente.

El contenido de todos estos ficheros se organiza en columnas separadas por un tabulador. A continuación, se muestra una línea de ejemplo del contenido de este dataset:

```
1  50246 26/11/2015 23:52:26.840  23.0127033333333  40.54788  3.9  0
145.320007324219  26/11/2015 23:52:26.840  1  30  0
```

Ejemplo de los datos aquí.

El dataset no posee ningún tipo de leyenda o explicación, así que, según el sentido común, podemos conocer lo que representa cada columna en el siguiente orden:

- ID: el primer elemento es el ID de la fila, por cada fichero.
- ID del Taxi: es el identificador único de cada taxi.
- Fecha: fecha del momento de la medición.
- Longitud: la longitud en la que se encuentra el taxi en ese momento.
- Latitud: la latitud en la que se encuentra el taxi en ese momento.
- ?: este campo no se sabe qué representa, pero posee normalmente valores bajos.
- Velocidad: es la velocidad en Km/h del taxi en ese momento.
- Orientación: se trata de la orientación geográfica en grados del taxi en ese momento.
- Fecha: se repite de nuevo la fecha en la que se toma la medición.
- Flag: es un valor que siempre se muestra a 1 y podríamos llegar a pensar que se trata de una variable para indicar que la muestra es válida.
- Distrito: variable para indicar en qué distrito se encuentra el taxi en ese momento.

2.5.2. FORMATEO DE LOS DATOS

De estos datos, se desea obtener grafos donde se puedan operar diferentes métricas de centralidad. Para ello, debemos formatear los datos para obtener vértices y aristas que los compongan.

Utilizaremos un programa en R en el cual procesaremos cada fichero obteniendo otro nuevo con las mismas columnas y 3 nuevas con el vértice correspondiente, un ID según la fecha y el taxi en cuestión y el vértice al que se desplaza.

En el código dividiremos el mapa en casillas de un ancho de 0.0001 grados con ID único a cada una de ellas que representa su posición geográfica y que será el ID del vértice para el grafo. También eliminamos posiciones GPS que puedan haber fallado y marcan lugares fuera de la ciudad. Entonces, agruparemos por taxi las medidas para obtener la ruta que ha hecho en el día y así obtener, por cada vértice, el siguiente al cual se dirige. Por último, guardamos la nueva tabla de valores en un fichero nuevo.

Función para obtener el ID del vértice según la casilla en la que se encuentre por su posición geográfica:

```
get_vertex_ID <- function(Lon, Lat, zone, sampling_step){
  grid_ncols <- ceiling((zone[3] - zone[1])/sampling_step)
  col <- ceiling((Lon - zone[1])/sampling_step)
  row <- ceiling((Lat - zone[2])/sampling_step)
  vertex_ID <- ((row-1)*grid_ncols + col)
  return (vertex_ID) #((row-1)*grid_ncols + col)
}
```

Función para realizar el cálculo contrario y obtener la posición GPS de un ID de vértice:

```
get_Lon_Lat <- function(vertex_ID, zone, sampling_step){
  grid_ncols <- ceiling((zone[3] - zone[1])/sampling_step)
  row <- ceiling(vertex_ID/grid_ncols)
  col <- vertex_ID %% grid_ncols
  Lon <- zone[1] + (col-1)*sampling_step + sampling_step/2
  Lat <- zone[2] + (row-1)*sampling_step + sampling_step/2
  return (cbind(Lon, Lat))
}
```

Calculamos la "celda" en la que está cada observación, mirando las coordenadas GPS:

```
table$Vertex <- get_vertex_ID(table$Lon, table$Lat, zone, sampling_step)
```

Agrupamos las muestras por ruta diaria de cada taxi:

```
table<- mutate(table, Trip_ID = paste(paste(
                day(table $Timestamp),
                month(table $Timestamp),sep="_"),
          Taxi_ID,sep="_"))
```

Ponemos juntos en cada fila el nodo (vertex) actual y el anterior de cada ruta, para extraer arcos:

```
table <- table %>% group_by(Trip_ID) %>%
          mutate(Lagged_Vertex = lag(Vertex))
```

Limpieza de filas con huecos en los edges:

```
table <- table[which(!is.na(table[,]$Lagged_Vertex)),]
```

El código completo se puede encontrar aquí.

Tras el formateo completo de todos los datos, obtendremos los nuevos ficheros con los vértices que necesitamos para la creación de grafos con la librería de Spark, GraphX. Aunque parezca que el proceso de formateo ha terminado, fue necesario realizar una última operación, ya que los cálculos con R con números tan grandes para las coordenadas en ocasiones dejaban valores tipo “1.23e+4”. Estos valores serán leídos por el programa en Scala para su procesamiento, y dará problemas ya que no podrá transformarlo a una variable de tipo *Integer* con la que trabajan los algoritmos. Para evitar estas excepciones en la ejecución posterior eliminaremos estas coincidencias con el siguiente comando UNIX:

```
sed -i '/[0-9]\+\.*[0-9]*e+[0-9]\+/d' vertices_2015*
```

El comando irá línea por línea de todos los nuevos ficheros con los vértices eliminando aquellas que contengan la expresión regular.

Ejemplo de los datos finales aquí.

2.5.3. CARGA EN HDFS

Con todos los datos ya procesados correctamente, es el momento de cargarlos en HDFS ya que es desde donde se van a leer para realizar los cálculos. Estando los datos en HDFS aprovechamos las características que ofrece como el particionado de los mismos entre los nodos y gracias al cual se puede trabajar de manera paralela entre los nodos.

En este caso, el formateo de los datos se realizó en una máquina ajena al clúster por lo que es necesario enviar los datos al mismo, ya sea por a través de la red o directamente conectando un dispositivo de almacenamiento al clúster.

Antes de cargar los datos en HDFS creamos una serie de directorios para tener el contenido organizado:

```
hdfs dfs -mkdir /miguel
hdfs dfs -mkdir /miguel/tesalonica
```

Una vez los datos se encuentran en el nodo maestro del clúster, utilizaremos el siguiente comando ya mencionado en anteriores capítulos:

```
hdfs dfs -put /miguel/tesalonica vertices_2015*
```

Esta operación tardará bastante por la cantidad de ficheros y tamaño que tienen. Para ir comprobando el progreso podemos usar la siguiente herramienta UNIX que ejecuta un comando cada cierto tiempo, de manera que vamos viendo qué ficheros se están subiendo:

```
watch hdfs dfs -ls /miguel/tesalonica
```

Una vez la operación termina ya estarían todos los ficheros cargados en HDFS, particionados y replicados por el clúster.

2.5.4. PROGRAMA EN SCALA

A la hora de crear un grafo con los datos de los ficheros y realizar diferentes cálculos se utiliza un programa implementado en Scala. Este programa, para poder ser ejecutado en una aplicación de Hadoop tiene que ser compilado a un archivo ".jar". Haremos uso del compilador sbt, aunque se podrían usar otros sin ningún problema.

Para instalar sbt:

```
sudo apt-get install sbt
```

En este caso, trabajando desde la consola sin un entorno de desarrollo, es necesario crear un directorio para el proyecto. Dentro de este directorio deberemos de crear varios elementos:

- Un archivo llamado build.sbt, donde indicaremos las librerías de las que depende el código Scala.
- La siguiente estructura de directorios básica src/main/scala/ con el fichero .scala en él. En nuestro caso, llamado App.scala.

```
$: tree ScalaApplication/
     src/
     └── main
         └── scala
             └── App.scala
     build.sbt
```

Cuando queramos compilar el programa utilizaremos el comando:

```
sbt package
```

Que creará unas nuevas carpetas target/scala-2.11/ que contendrán el fichero .jar ejecutable con el nombre y versión que le indiquemos en el fichero build.sbt[1].

La librería GraphX de Spark ofrece múltiples funciones para operar con los grafos creados y 3 algoritmos de cálculo de propiedades: "Connected Components", "Triangle Counting" y "Page Rank". Es este último y dos más de otras librerías de la comunidad,

[1] `name := "nombreAplicación"`
`versión := "1.0"`
`scalaVersion := "2.11.8"`

“Betweenness Centrality” y “Eigenvector Centrality”, los que se han elegido para operar sobre los grafos obtenidos de los datos.

2.5.4.a. Programas Spark en Scala

Spark crea un SparkContext para cada aplicación que lanza y para realizar operaciones con los recursos del clúster. Por esto, en toda aplicación ejecutable en el clúster es necesario añadir el siguiente código Scala:

```
val spark = SparkSession
            .builder
            .appName("Application")
            .getOrCreate()
val sc = spark.sparkContext
```

Podemos ver que creamos u obtenemos el SparkSession para luego tener el SparkContext al cual podemos llamar para realizar operaciones con los dispositivos de entrada/salida como la lectura de un fichero, por ejemplo. Además, con el método `conf()` podemos enviar diferentes configuraciones para el funcionamiento de los servicios de Spark o Hadoop.

Es necesario importar en el programa la librería encargada de esto:

```
import org.apache.spark.sql.SparkSession
```

Para finalizar una aplicación y que el gestor del clúster lo sepa se usa el método `stop()` sobre la sesión de Spark:

```
spark.stop()
```

2.5.4.b. Creación de grafos

GraphX, como se ha explicado, es una librería de Spark para el manejo de grafos, para comenzar a crear estos grafos deberemos añadir las dependencias necesarias de esta librería a nuestro fichero build.sbt:

```
libraryDependencies ++= Seq(
      "org.apache.spark" %% "spark-sql" % "2.3.1",
      "org.apache.spark" %% "spark-graphx" % "2.3.1",
      "org.apache.spark" %% "spark-core" % "2.3.1"
}
```

Además de las dependencias debemos de importar los paquetes en el programa:

```
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
```

Como sabemos, los datos se encuentran en los ficheros divididos en columnas, por lo que crearemos una clase que identifique todas esas columnas y un método que permite dividir y obtener todos estos campos para operar con ellos cuando los leamos de HDFS.

```
case class Position(ID:Long, Daily_Sample_ID:Long, Taxi_ID:Long,
      Timestamp:String, Lon:Double, Lat:Double, V6:String,
      Speed:Double, Orientation:Double, Timestamp2:String, V10:String,
      Distrit:Int, V12:String, Vertex:Long, Trip_ID:String,
      Lagged_Vertex:Long)

def parsePosition(str: String): Position = {
      val line = str.split(",")
      Position(line(0).toLong, line(1).toLong, line(2).toLong,
            line(3), line(4).toDouble, line(5).toDouble, line(6),
            line(7).toDouble, line(8).toDouble, line(9), line(10),
            line(11).toInt, line(12), line(13).toLong, line(14),
            line(15).toLong)
}
```

Entonces, leeremos los datos desde el sistema de ficheros de Hadoop a partir de la configuración de Hadoop y el SparkContext. Como son tantos ficheros, no deseamos leer uno en concreto y en el futuro los iremos calculado uno a uno, necesitamos conseguir todos los ficheros que se encuentren dentro del directorio del dataset, /miguel/tesalonica/.

```
val fs=FileSystem.get(sc.hadoopConfiguration)
val files = fs.listStatus(new Path("/miguel/tesalonica"))
```

En la variable files se encontrará una lista con todos los ficheros dentro del directorio, para guardar el contenido de uno de ellos a una variable de tipo RDD utilizamos el método `textfile()` del SparkContext.

```
val textRDD = sc.textFile(files(0).getPath.toString)
```

En esta variable tendremos todo el contenido del fichero en cuestión, pero deberemos de convertirlo a un grupo de elementos de la clase Position, pero posteriormente obtener las columnas deseadas para crear el grafo. Únicamente deseamos los vértices obtenidos en el formateo.

```
val positionsRDD = textRDD.map(parsePosition)
routes = positionsRDD.map(
     pos => (pos.Vertex, pos.Lagged_Vertex)
)
```

A la hora de crear los grafos existen dos constructores básicos, uno al que se le pasan los vértices y las aristas y otro que, a partir de solo las aristas, crea los vértices automáticamente para crear el grafo.

```
val graph = Graph(VertexRDD v, EdgesRDD e, defaultVertex)
val graph = Graph.fromEdges(EdgesRDD e, defaultValue = 0)
```

En la primera opción, los grafos se pueden crear con mucha más información adicional al ID en los nodos que en la segunda opción, donde la creación automática de los vértices los crea con el valor por defecto en todos los nodos. En nuestro caso, no es importante añadir información al grafo, ya que no la necesitamos, únicamente el ID de los vértices, que indican ya de por sí unas posiciones GPS en un mapa.

Antes de crear el grafo será necesario transformar la variable `routes` a la clase específica para las aristas de un grafo, `EdgeRDD`.

```
val edges = routes.map {
     case ((vertex, lagged_Vertex), id) =>Edge(vertex.toLong,
          lagged_Vertex.toLong)
}
```

Creamos ahora el grafo:

```
val graph = Graph.fromEdges(edges, defaultValue = 0)
```

Con el grafo creado ahora se le pueden realizar multitud de operaciones sobre él. Las más básicas son, simplemente, obtener sus vértices, aristas o triadas:

```
graph.vertices          graph.edges          graph.triplets
```

Pero también existen muchas otras que se incluyen en la librería de GraphX y se pueden ver aquí, u otras muchas que la comunidad crea.

2.5.4.c. PageRank

Page Rank es un algoritmo creado por Google que calcula la importancia de cada vértice de un grafo según el número de arcos que apunten a él desde otros vértices. Entonces, si una arista va del nodo A al nodo B, B tendrá un valor de Page Rank mayor. Se decide utilizar este algoritmo para calcular con nuestro grafo ya que es uno de los algoritmos oficiales que ofrece GraphX y con él podremos obtener de los puntos geográficos de la ciudad los cruces de las carreteras, por ejemplo.

Para su implementación utilizamos el método pageRank() de GraphX para objetos de tipo Graph. Se ofrecen dos opciones para el algoritmo, una en la que se especifica el número de iteraciones de la ejecución y otra en la que se ejecuta hasta llegar a cierta convergencia:

```
val pageGraph = graph.pageRank(0.0001)
```

En la nueva variable, tras cierto tiempo de cálculo, que dependerá del tamaño del grafo, obtendremos un nuevo grafo igual que el anterior, pero en cada uno de los vértices se ha añadido el valor calculado de PageRank.

```
$: pageGraph.vertices.take(3)
     (2753634,0.3169677496307769)
     (678438,0.6387060092674801)
     (2503288,1.682575669009857)
```

2.5.4.d. EigenVector Centarliy

Las medidas de centralidad en los grafos son cálculos que determinan la importancia de un punto dentro de este. La medida Eigenvector da más importancia a un vértice si este está unido a otros vértices relevantes del grafo. Este algoritmo es uno de los más usados en grafos, pero en GraphX no se encuentra, por lo que se debe buscar en las librerías de la comunidad.

Spark posee una web donde buscar paquetes y librerías adicionales para todo tipo de funciones que se quiera añadir a nuestro código: https://spark-packages.org/. Indagando en ella y a través de Google, podemos encontrar una librería llamada SparklingGraph que posee múltiples funciones de procesamiento de grafos, entre las cuales se encuentra el cálculo de la centralidad eigenvector.

Para la utilización de esta librería debemos añadir las dependencias y resolutores necesarios al fichero build.sbt:

```
resolvers ++= Seq(
   "Spark Packages Repo" at
     "http://dl.bintray.com/spark-packages/maven",
   "Sonatype OSS Snapshots" at
     "https://oss.sonatype.org/content/repositories/snapshots"
)

libraryDependencies ++= Seq(
   "ml.sparkling" %% "sparkling-graph-api" % "0.0.7",
   "ml.sparkling" %% "sparkling-graph-operators" % "0.0.7"
)
```

Además, importamos la libraría en la clase principal donde se va a usar el cálculo. Para realizar el cálculo, llamamos al método eigenvectorCentrality().

```
import ml.sparkling.graph.operators.OperatorsDSL._
// .. //
val eigenGraph = graph.eigenvectorCentrality()
```

El cálculo finalizará y obtendremos, tal y como el cálculo de Page Rank, el mismo grafo, pero no el valor de la centralidad adjunto a cada nodo.

```
$: eigenGraph.vertices.take(3)
     (2753634,1.427522228497337E-12)
     (678438,3.8188734070730374E-10)
     (2503288,8.067493663127386E-8)
```

2.5.4.e. Betweenness Centrality

Betweenness centrality, o intermediación, es una medida de centralidad que calcula el número de veces que un nodo es de paso a lo largo del camino entre otros 2 nodos de la red. Este algoritmo, como la centralidad eigenvector deberemos de encontrarlo en las librerías de la comunidad.

Utilizaremos la librería llamada "spark-betweenness" que implementa este cálculo para Scala. Añadiremos entonces la siguiente dependencia al fichero build.sbt:

```
libraryDependencies+="com.centrality"%%"spark-betweenness"%"1.0.0"
```

En el método para el cálculo de la centralidad se debe de incluir el tamaño de los subgrafos generados, cuanto mayor sea, el coste del cómputo aumentará. Uso de la librería:

```
import com.centrality.kBC.KBetweenness
// .. //
val kBCGraph = KBetweenness.run(graph, 3)
```

El resultado, como en las otras dos métricas, es el mismo grafo introducido, pero con los valores de la centralidad añadidos a los vértices:

```
$: kBCGraph.vertices.take(3)
     (2753634,0.0)
     (678438,2.2222222222222223)
     (2503288,103.10325796216128)
```

2.5.4.f. Programa final

Con el fin de probar estas 3 métricas en el dataset y ver el funcionamiento del clúster, se crea un programa en Scala. Al programa se le indica qué es lo que debe de ejecutar, las diferentes opciones son:

- Cálculo de Page Rank: se introduce la opción "-pr" junto con el valor de convergencia deseado y el número de ficheros que se quiera procesar. El programa accederá los ficheros en /miguel/tesalonica/ y calculará uno por uno la métrica, guardando los vértices y aristas del grafo resultante en el directorio /miguel/results/PageRank/ de HDFS. Si el programa detecta que el fichero que va a procesar ya ha sido calculado continua con el siguiente.

- Cálculo de la centralidad eigenvector: se introduce la opción “-ei” junto con el número de ficheros que se quiera procesar. El programa accederá los ficheros en /miguel/tesalonica/ y calculará uno por uno la métrica, guardando los vértices y aristas del grafo resultante en el directorio /miguel/results/eigen/ de HDFS. Si el programa detecta que el fichero que va a procesar ya ha sido calculado continua con el siguiente.
- Cálculo de la intermediación: se introduce la opción “-bt” junto con el tamaño de los subgrafos deseados y el número de ficheros que se quiera procesar. El programa accederá los ficheros en /miguel/tesalonica/ y calculará uno por uno la métrica, guardando los vértices y aristas del grafo resultante en el directorio /miguel/results/betweenness/ de HDFS. Si el programa detecta que el fichero que va a procesar ya ha sido calculado continua con el siguiente.
- Test: se introduce la opción “-t” que ejecutará cada una de las centralidades sobre el primer archivo encontrado sin guardar resultados con el fin de ver los tiempos de ejecución de cada una.

A la hora de ejecutar las centralidades en el clúster del CICEI se pudo observar que el cálculo de la intermediación era demasiado costoso para que soportase realizar el proceso sobre el grafo de un archivo del dataset completo. El proceso comenzaba sin problemas, pero al paso de las horas sin tener resultados de una sola ejecución, la aplicación de para por errores relacionados con el espacio de memoria utilizado. De modo que para solucionar este problema se decide dividir los datos de cada fichero en 20 partes iguales.

El problema de dividir un grafo es que pierdes precisión a la hora de calcular la métrica, ya que cortas los enlaces que existen entre esas 20 partes del grafo y no se contemplan. La decisión de este número de partes se tomó en base al tiempo de cálculo según los tamaños de las particiones. Con 20 partes el tiempo de ejecución de la centralidad podía llegar hasta las 3 horas y cuarto en el peor de los casos. De media rondaría los 15 minutos, que es un tiempo relativamente corto. Hay que tener en cuenta que esos 15 minutos es por parte del fichero calculada, es decir, el cálculo de 1 solo fichero se alarga hasta las 5 horas.

A diferencia de la centralidad eigenvector y Page Rank, no se calcula el fichero entero, si no parte por parte. Cada uno de los 20 grafos resultantes se guardan en el directorio /tmp/ con un nombre “<fichero dataset>_<v:vertices | e:edges>_<n parte>” (vertices_20150101_120310_v_3, p.ej.) hasta que finaliza el cálculo de todas las partes. Después, se abre cada fichero de vértices de cada parte para crear 2 colecciones de

vértices: una con la suma de los valores de centralidad que pertenezcan a vértices que aparecen en varios ficheros y otra con un contador por vértice que aumenta cada vez que dicho vértice se repite.

```
val filesV = fs.listStatus(new Path("/miguel/tmp/"+ file + "_v_" + 0))

var vert = sc.textFile(filesV(1).getPath.toString).map{ line =>
    val fields = line.substring(1, line.length-1).split(",")
        ((fields(0).toLong, fields(1).toDouble))
}

var vertCount = sc.textFile(filesV(1).getPath.toString).map{ line =>
    val fields = line.substring(1, line.length-1).split(",")
        ((fields(0).toLong, 1))
}
for(j <- 1 to 19){

    @transient val filesV = fs.listStatus(new Path("/miguel/tmp/" +
             file + "_v_" + j))

    var auxVert = sc.textFile(filesV(1).getPath.toString).map{ line =>
        val fields = line.substring(1, line.length-1).split(",")
            ((fields(0).toLong, fields(1).toDouble))
    }

    vert = vert.fullOuterJoin(auxVert).map {
        case (id:Long, (Some(left), Some(right))) =>
          (id, left + right)
        case (id:Long, (None, Some(right)))=>
          (id, right)
        case (id:Long, (Some(left), None))=>
          (id, left)
    }

    var auxVertCount = sc.textFile(filesV(1).getPath.toString).map{
      line =>
        val fields = line.substring(1, line.length-1).split(",")
            ((fields(0).toLong, 1))
    }

    vertCount = vertCount.fullOuterJoin(auxVertCount).map {
        case (id:Long, (Some(left), Some(right))) =>
          (id, left + 1 )
        case (id:Long, (None, Some(right)))=>
          (id, right)
        case (id:Long, (Some(left), None))=>
          (id, left)
    }
}
```

Finalizada la recogida de todos los valores se unen las dos colecciones realizando la media aritmética de los valores de centralidad de cada vértice.

```
var meanVert = vert.fullOuterJoin(vertCount).map{
    case(id, (Some(cent), Some(count))) =>
        (id, cent/count)
}
```

Por último, se guardan los resultados en el directorio, antes mencionado, /miguel/results/betweenness/.

Hay que recordar que para compilar el programa utilizaremos el comando sbt package desde el directorio del proyecto y así obtendremos el fichero .jar ejecutable.

El código del programa completo se puede ver aquí.

2.5.5. EJECUCIÓN DEL PROGRAMA

Con el programa correctamente implementado, es el momento de ejecutarlo en el clúster del CICEI. Para ello se utilizará el comando spark-submit que se ha explicado en capítulos anteriores con las siguientes opciones.

El comando incluirá las siguientes opciones:

- `--class "App"`
  La clase del código donde está el programa principal.

- `--master yarn`
  La aplicación spark se va a ejecutar sobre el gestor de recursos Yarn de Hadoop.

- `--deploy-mode client`
  El maestro de la aplicación se ejecutará en la máquina aton, que es desde donde se laza.

- `--packages dmarcous:spark-betweenness:1.0-s_2.10,`
  `ml.sparkling:sparkling-graph-examples_2.11:0.0.7`
  Estos paquetes son necesarios ya que se tratan de las librerías externas con los algoritmos de las centralidades a calcular.

- `--conf spark.driver.memory=6g` y `--conf spark.executor.memory=6g`
  Se decide que el máximo de memoria que se le asigne a los procesos driver y executors sea de 6GB. 6GB porque el clúster dispone de 8GB y dejamos 2GB de espacio reservado para el sistema operativo, los servicios de Hadoop y por el "heap overhead" de Java.

- No es necesario incluir configuraciones para los núcleos que se asignan a los executors o al driver, ya que por defecto están configurados para que solo reserven 1 núcleo. Solamente 1 núcleo para dejar el otro de cada máquina al resto de procesos del sistema y Hadoop.

- `--conf spark.dynamicAllocation.enabled=true`
  `--conf spark.shuffle.service.enabled=true`
  Con estas dos configuraciones habilitamos el uso del Spark Dynamic Resource Allocation como se ha explicado.

- `--conf spark.dynamicAllocation.minExecutors=2 | --conf spark.dynamicAllocation.maxExecutors=2`
  Con el Dynamic Resource Allocation activado podremos indicar a Spark cuántos executors queremos que se creen como máximo o mínimo en la aplicación. Por la configuración elegida y las características del clúster, no se podrán crear más que 1 executor por nodo. Así esta opción se puede utilizar para comunicar el número de máquinas que se desean que trabajen.

- `--conf spark.executor.instances=2`
  Indica el número de executors que se crean al comienzo de la aplicación, debe de ser mayor que el mínimo.

Con estas opciones se realizaron 3 tipos de ejecuciones para comprobar los tiempos de ejecución y ver cómo trabaja la tecnología en el clúster.

Ejecución con Dynamic Resource Allocation y permitiendo que Spark cree el número de executors que necesite a lo largo del clúster:

```
spark-submit --class "App" --master yarn --deploy-mode client
      --packages dmarcous:spark-betweenness:1.0-s_2.10,
            ml.sparkling:sparkling-graph-examples_2.11:0.0.7
      --conf spark.dynamicAllocation.enabled=true
      --conf spark.shuffle.service.enabled=true
      --conf spark.dynamicAllocation.minExecutors=2
      --conf spark.executor.instances=2
      --conf spark.driver.memory=6g
      --conf spark.executor.memory=6g
      /home/miguel/project_2.11-1.0.jar -t -bt 3 12
```

Ejecución con Dynamic Resource Allocation e indicando que Spark cree como máximo 2 executors:

```
spark-submit --class "App" --master yarn --deploy-mode client
      --packages dmarcous:spark-betweenness:1.0-s_2.10,
            ml.sparkling:sparkling-graph-examples_2.11:0.0.7
      --conf spark.dynamicAllocation.enabled=true
      --conf spark.shuffle.service.enabled=true
      --conf spark.dynamicAllocation.maxExecutors=2
      --conf spark.driver.memory=6g
      --conf spark.executor.memory=6g
      /home/miguel/project_2.11-1.0.jar -pr 0.0001 12
```

Ejecución sin Dynamic Resource Allocation:

```
spark-submit --class "App" --master yarn --deploy-mode client
      --packages dmarcous:spark-betweenness:1.0-s_2.10,
            ml.sparkling:sparkling-graph-examples_2.11:0.0.7
      --conf spark.driver.memory=6g
      --conf spark.executor.memory=6g
      /home/miguel/project_2.11-1.0.jar -ei 12
```

Todas estas ejecuciones irán mostrando por consola los logs de la ejecución para que sepamos qué es lo que está haciendo la aplicación en ese momento. Para acceder a posteriori a ellos se podrán ver a través de la web de Yarn o en los ficheros de logs. Otra opción podría ser redirigir la salida estándar a un fichero, que podremos ir consultando.

2.5.6. OBTENCIÓN DE RESULTADOS

Se llevaron a cabo ejecuciones para el cálculo de las 3 centralidades para 2 semanas, es decir, 15 ficheros del dataset. Además, se obtuvieron los tiempos de cálculo según diferentes tipos de ejecuciones:

- Cálculos individuales de las métricas con todo el clúster disponible y con Dynamic Resource Allocation.
- Cálculos individuales de las métricas con todo el clúster disponible y con Dynamic Resource Allocation pero sólo 2 executors.
- Cálculo de las métricas para los 15 días con todo el clúster disponible y con Dynamic Resource Allocation.
- Cálculo de las métricas para los 15 días con todo el clúster disponible y con Dynamic Resource Allocation, después de reiniciar Yarn para limpiar todo tipo de logs, cachés, etc.
- Cálculo de las métricas para los 15 días con todo el clúster disponible y sin Dynamic Resource Allocation.

Los resultados de las centralidades se encuentran en el directorio de HDFS /miguel/results/. Para descargarlos y así poder operar con los datos en R utilizaremos el conocido comando:

```
hdfs dfs -get /miguel/results .
```

De todos los datos en sus directorios queremos llegar a tener una tabla los resultados de cada centralidad por cada vértice. Como existirán muchos vértices que se repitan entre ficheros, obtendremos las medias y las desviaciones típicas de cada uno de los nodos para cada métrica, además, añadiremos unas columnas con la latitud y longitud correspondiente a cada vértice para que posteriormente en la representación sea más fácil trabajar.

El programa en R irá fichero por fichero leyendo y cargando todos los nodos y valor de la métrica en una tabla. Dicha tabla contendrá 16 columnas: 1 del ID del vértice y 15 con cada valor de la métrica de cada fichero.

```
file <- allFiles[1]
filePath <- paste(paste(path, file, sep=""), "/part-00000", sep="")
txt <- gsub("[()]", "", readLines(filePath))
table = fread(text=txt, sep = ",")

for (i in 2:16){
  file <- allFiles[i]
  filePath <- paste(paste(path, file, sep=""), "/part-00000", sep="")
  txt <- gsub("[()]", "", readLines(filePath))

  auxTable <- fread(text=txt, sep = ",")
  table <- merge(table,auxTable,by="V1",all=TRUE)

  names(table)[ncol(table)] = paste("V2_", i, sep = "")
}
```

Posteriormente, calculamos la media y desviación típica de las filas de valores, obviando aquellos elementos que no se hayan repetido en algún fichero.

```
table$mean = rowMeans(table[,2:7], na.rm = TRUE)
table$sd = rowSds(data.matrix(table[,2:7]), na.rm = TRUE)
```

Tras tener las 3 tablas de las 3 métricas, las unimos en una con solo los valores de las medias y desviaciones típicas para guardarlas después.

```
table<-merge(eiTable[,c(1,18,19)], prTable[, c(1,18,19)], by="V1", all=TRUE)
table<-merge(table, btTable[, c(1,18,19)], by="V1", all=TRUE)
```

Añadimos las posiciones GPS a través del método `get_Lon__Lat(vertex_id, zone, sampling_step)` y guardamos la tabla en un fichero:

```
table$Lon <- get_Lon_Lat(as.double(table$V1),
                                c(22.82,40.49,23.00,40.71), 1E-4)[,1]
table$Lat <- get_Lon_Lat(as.double(table$V1),
                                c(22.82,40.49,23.00,40.71), 1E-4)[,2]

write.table(table, sep = ",", na = "", col.names = TRUE, row.names = FALSE,
      file = "C:/Users/migue/results/all_means_sds.csv")
```

Puede acceder al código completo aquí.

Los tiempos de ejecución están en los logs de las ejecuciones, ya que los mensajes de impresión en consola de Scala salen junto con el resto de los logs del funcionamiento de la aplicación. Para obtenerlos utilizamos el comando “grep” sobre todos los ficheros de logs:

```
grep -EH “WET.+ENDED” application_logs_* > all_times.txt
```

2.5.7. MAPA

Con los resultados obtenidos se representa en un mapa interactivo las 3 medidas de centralidad. En él podemos ver los valores medios más altos de los vértices en sus posiciones geográficas. No se pueden representar todos porque se trata de millones de puntos, por lo que solo se representan los 10000 puntos con valores de centralidad más alta. Se utiliza para la muestra la librería para R de representación gráfica “leaflet”.

Comenzamos con la lectura del fichero que se creó en la sección anterior y la ordenación y filtrado de los elementos. Se obtienen 3 tablas ordenadas por las medias para las 3 medidas:

```
table <- fread(sep = ",", header = TRUE, showProgress = TRUE,
      file = "C:/Users/migue/Documents/cosaTFG/results/all_means_sds.csv")

# Para Page Rank. Mismo procedimiento para cada métrica
table_pr_mean = tail(table[order(table$pr_mean),], 10000)
```

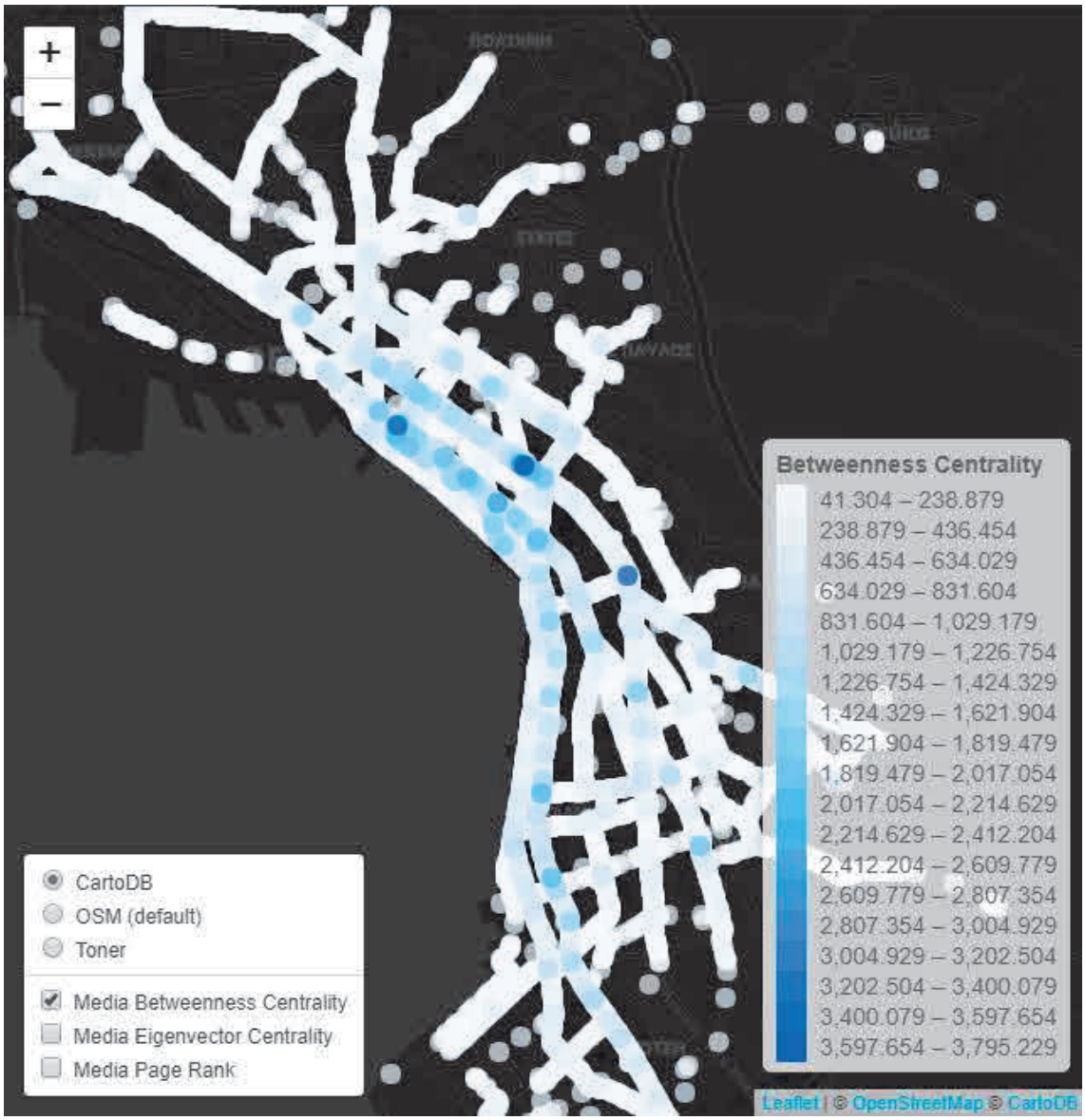
Las medias se representarán en puntos con diferentes grados de un color para indicar visualmente qué puntos tienen un valor más alto. Para ello creamos una paleta de colores y una secuencia para crear 20 rangos de color entre los valores de las medias:

```
bluecols <- brewer.pal(9, 'Blues')
newcol <- colorRampPalette(bluecols)

bins_pr_mean = seq(min(table_pr_mean$pr_mean),
                   max(table_pr_mean$pr_mean), length.out = 20)

palette_pr_mean = colorBin(palette=newcol(20), domain=table_pr_mean,
                           na.color="transparent", bins=bins_pr_mean)
```

Por último, antes de crear el mapa, creamos unas leyendas para que aparezcan cada vez que pasemos el ratón por encima de un puntAo en el mapa. En ellas se mostrará el ID del vértice, su media de valor de centralidad y su desviación típica:

```
text_pr_mean=paste("ID del Vertice: ", table_pr_mean$V1, "<br/>",
                   "Media: ", table_pr_mean$pr_mean, "<br/>",
                   "Desviación Tipica: ", table_pr_mean$pr_sd, sep="") %>%
                    lapply(htmltools::HTML)
```

La librería “leaflet” permite que se cree el mapa con las 3 métricas juntas, y que el usuario pueda tener el control sobre la métrica que quiere ver, además de hacer zoom o cambiar el fondo sobre el que se representan. Para ver el código completo del programa clique aquí. Para descargar el mapa y utilizarlo, pinche aquí.



13 Mapa de métricas de centralidad. Fuente: elaboración propia

2.5.8. ANÁLISIS DE TIEMPOS

En esta sección se realizará la creación y análisis de diferentes gráficos para el análisis de los tiempos de ejecución según los 5 tipos de ejecución que se mencionaron antes:

- Cálculos individuales de las métricas con todo el clúster disponible y con Dynamic Resource Allocation.

- Cálculos individuales de las métricas con todo el clúster disponible y con Dynamic Resource Allocation pero sólo 2 executors.

- Cálculo de las métricas para los 15 días con todo el clúster disponible y con Dynamic Resource Allocation.

- Cálculo de las métricas para los 15 días con todo el clúster disponible y con Dynamic Resource Allocation, después de reiniciar Yarn para limpiar todo tipo de logs, cachés, etc.

- Cálculo de las métricas para los 15 días con todo el clúster disponible y sin Dynamic Resource Allocation.

A la hora de representar en gráficos estas ejecuciones, crearemos múltiples tablas en R para reunir clasificar todos los tiempos. Las tablas son las siguientes:

- bt_table_be: tabla que contiene todos los tiempos para la ejecución de Betweenness Centrality sobre 15 ficheros y con todo el clúster disponible. Estarán en ella todos los tiempos para cada 1/20 de fichero calculado.

- bt_table_ber: tabla que contiene todos los tiempos para la ejecución de Betweenness Centrality sobre 15 ficheros y con todo el clúster disponible, tras un reinicio del proceso Yarn. Estarán en ella todos los tiempos para cada 1/20 de fichero calculado.

- bt_table_bey: tabla que contiene todos los tiempos para la ejecución de Betweenness Centrality sobre 15 ficheros sin el Spark Dynamic Resource Allocation. Estarán en ella todos los tiempos para cada 1/20 de fichero calculado.

- table_bt: tabla que contiene la media de los tiempos de ejecución de Betweenness Centrality para los diferentes tipos de ejecución.

- table_ei: tabla que contiene la media de los tiempos de ejecución de Eigenvector Centrality para los diferentes tipos de ejecución.

- table_pr: tabla que contiene la media de los tiempos de ejecución de Page Rank para los diferentes tipos de ejecución.

Con los datos ya clasificados y preparados, comenzaremos con la creación de los gráficos.

Primeramente, construiremos 3 gráficos lineares para mostrar los 3 modos de ejecuciones largas para Betweenness Centrality: con spark Dynamic Resource Allocation, reiniciando antes y sin Dynamic Resource Allocation.

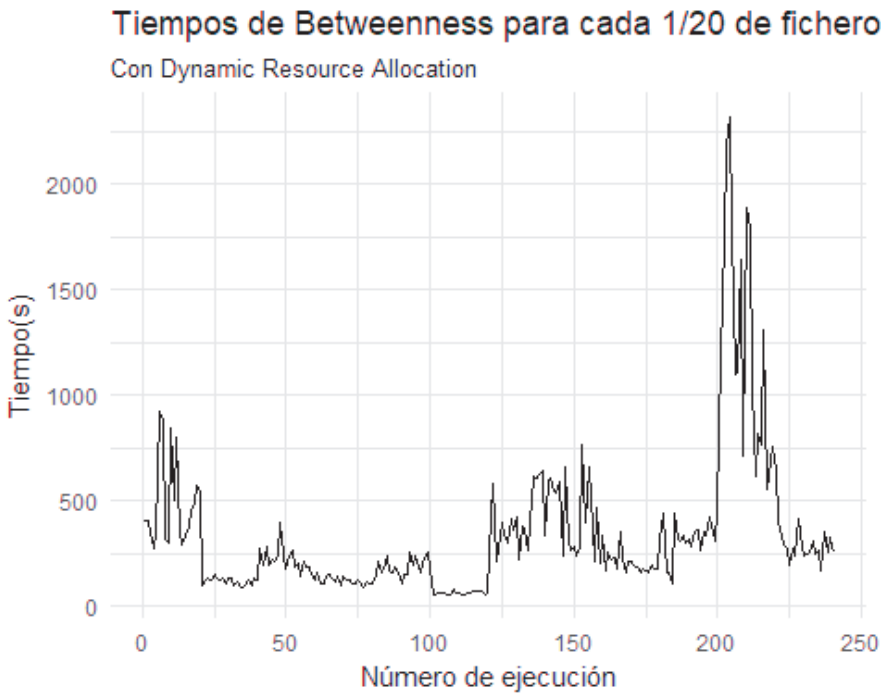
```
linear_g_bt<-ggplot(data=table_bt_be, aes(x=V1, y=V2)) +
               geom_line() +
               labs(title="Tiempos de Betweenness para cada 1/20 de fichero",
                   subtitle="Con Dynamic Resource Allocation",
                   x="Número de ejecución", y="Tiempo(s)") +
               theme_minimal()
```

```
linear_g_btr<-ggplot(data=table_bt_ber, aes(x=V1, y=V2)) +
              geom_line() +
              labs(title="Tiempos de Betweenness para cada 1/20 de fichero",
                   subtitle="Con Dynamic Resource Allocation después de
                             reinicio de Yarn",
                   x="Número de ejecución", y="Tiempo(s)") +
              theme_minimal()
```

```
linear_g_bty<-ggplot(data=table_bt_bey, aes(x=V1, y=V2)) +
              geom_line() +
              labs(title="Tiempos de Betweenness para cada 1/20 de fichero",
                   subtitle="Sin Dynamic Resource Allocation",
                   x="Número de ejecución", y="Tiempo(s)") +
              theme_minimal()
```

Los grafos los podemos ver a continuación:

```
$> linear_g_bt
```

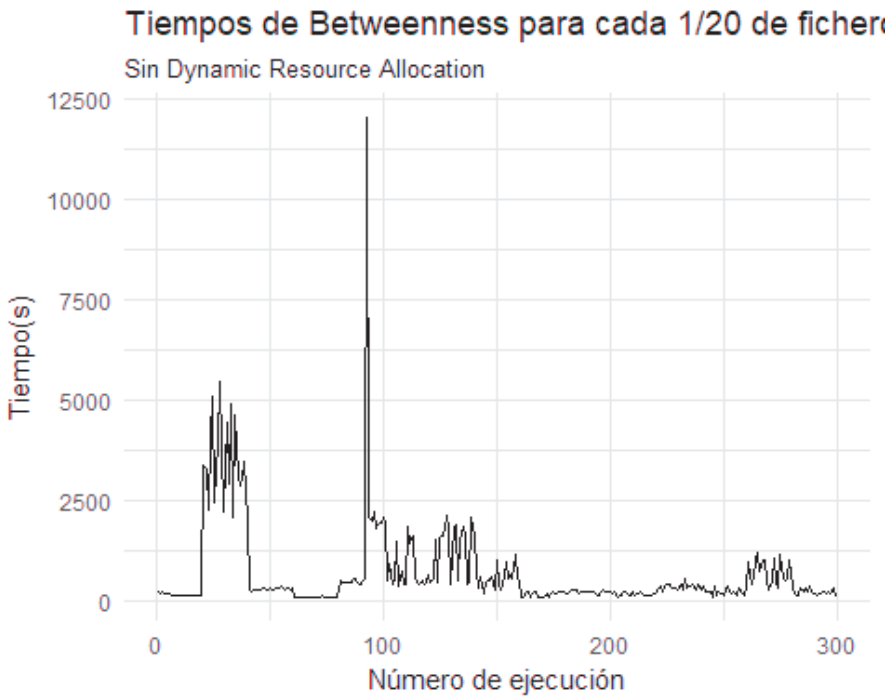


14 Gráfico para Betweenness centrality con DRA. Fuente: elaboración propia

```
$> linear_g_btr
```

15 Gráfico para Betweenness centrality con DRA tras reinicio. Fuente: elaboración propia

```
$> linear_g_bty
```



16 Gráfico para Betweenness centrality sin DRA. Fuente: elaboración propia

Con estos 3 gráficos se pueden sacar múltiples conclusiones de cómo, a lo largo de largas ejecuciones sobre grandes cantidades de datos trabajan los servicios de la arquitectura y la implementación del algoritmo.

Un punto en común entre los tres gráficos es que se pueden distinguir diferentes bloques de tiempos relativamente parejos, cada bloque se corresponde con los cálculos de las 20 partes de un fichero.

También se puede observar como las primeras ejecuciones siempre suelen tardar más que las siguientes. Esto indica que existe un proceso de cacheo en el algoritmo que permite reducir drásticamente los tiempos debido a que ciertos datos ya se encuentran en memoria y no es necesario acceder al lento almacenamiento del disco duro. Como comenta la guía del algoritmo en su página de GitHub, "*[…] We actually hold all k-graphlets in memory for Brandes calculation as they are the core of parallelizing this algorithm. […]*", se van guardando en memoria los subgrafos que se van generando para realizar los cálculos.

Este proceso de almacenaje en memoria permite reducir el tiempo de acceso a datos en las siguientes iteraciones, pero en ocasiones, la memoria de los nodos acaba llenándose y es por ello por lo que vemos como aumenta el tiempo de ejecución gratamente.

Además, durante estas ejecuciones largas se pueden ver en los logs de la aplicación cómo se dedica el tiempo a limpieza y borrado de registros de la memoria:

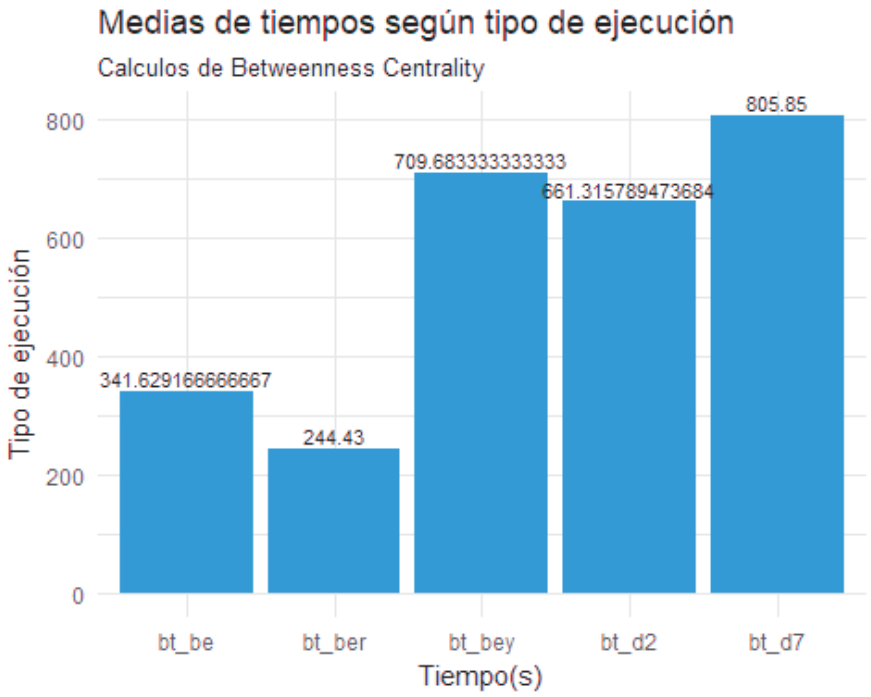
```
[...]
2018-12-25 17:58:28 INFO  ContextCleaner:54 - Cleaned accumulator 48269
2018-12-25 17:58:28 INFO  ContextCleaner:54 - Cleaned accumulator 47926
2018-12-25 17:58:28 INFO  BlockManagerInfo:54 - Removed broadcast_2045_piece0
      on aton.darwin.cicei.com:38701 in memory (size: 3.4 KB, free: 3.0 GB)
2018-12-25 17:58:28 INFO  BlockManagerInfo:54 - Removed broadcast_2045_piece0
      on isis.darwin.cicei.com:33477 in memory (size: 3.4 KB, free: 2.5 GB)
2018-12-25 17:58:28 INFO  BlockManagerInfo:54 - Removed broadcast_2045_piece0
      on nut.darwin.cicei.com:33673 in memory (size: 3.4 KB, free: 2.5 GB)
2018-12-25 17:58:28 INFO  ContextCleaner:54 - Cleaned accumulator 47960
2018-12-25 17:58:28 INFO  ContextCleaner:54 - Cleaned accumulator 47888
[...]
```

Otro dato que arrojan estas gráficas, y que comprobaremos más adelante, es la diferencia entre las tres ejecuciones. La primera la ejecución se realizó después de haber realizado otros cálculos y procesos en el clúster, y como se puede ver, se ralentizan los tiempos llegando al final de la ejecución. Esto puede indicar que la memoria de los nodos se llenó antes de acabar con todos los ficheros, lo que podría indicar que ya contenía datos antes de comenzar la ejecución. Se llega a este razonamiento porque si comprobamos la gráfica en la segunda ejecución; con el mismo tipo de configuración del clúster, pero habiendo reiniciado el proceso Yarn antes; podremos ver que una vez cachea los primeros datos, ya no vuelve a tener problemas y los tiempos de los cálculos son todos bajos hasta completarse el trabajo.

Por último, podemos ver que en la ejecución si Dynamic Resource Allocation hay muchos más cambios en los tiempos, lo que repercutirá en el tiempo medio final de este tipo de ejecución.

A continuación, se construye un gráfico de barras para mostrar las medias de tiempos para los diferentes tipos de ejecución de Betweenness Centrality.

```
g_bt <- ggplot(data=table_bt, aes(x=V1, y=V2)) +
           geom_bar(stat="identity", fill="steelblue") +
           geom_text(aes(label=V2), vjust=-0.3, size=3) +
           labs(title="Medias de tiempos según tipo de ejecución",
                subtitle="Calculos de Betweenness Centrality",
                x="Tiempo(s)", y="Tipo de ejecución") +
           theme_minimal()
```



17 Medias de tiempos para Betweenness Centrality. Fuente: elaboración propia

De entrada, se puede ver una gran diferencia en la media de los tiempos entre las ejecuciones largas con Dynamic Resource Allocation (`bt_be`: ejecución de 15 ficheros. `bt_ber`: ejecución de 15 ficheros con reinicio anterior), y el resto (`bt_bey`: ejecución de 15 ficheros sin Dynamic Resource Allocation. `bt_d2`: ejecución individual de 1/20 de fichero con máximo 2 ejecutores. `bt_d7`: ejecución individual de 1/20 de fichero con mínimo de 2 ejecutores). Esta diferencia en tiempos se debe al uso de la caché en las ejecuciones largas que no llega a ser efectivas en las ejecuciones individuales porque solamente se realza 1 vez el cálculo por aplicación lanzada.

Se puede observar, que la ejecución es más rápida cuando se ha reiniciado el servicio antes de lanzarla. Como se mostró en los gráficos anteriores, no se pierde tiempo vaciando la memoria cuando se queda sin espacio porque no llega a ese punto.

Otra conclusión que se saca, viendo las ejecuciones con Dynamic Resource Allocation contra la ejecución en la que no se utiliza, es que el servicio de *Shuffle* de MapReduce es más lento que el que implementa Spark y que se usa cuando la opción está activada. Además, influye que los tiempos hayan sido, para la ejecución sin Dynamic Resource Allocation, tan variables y poco consistentes.

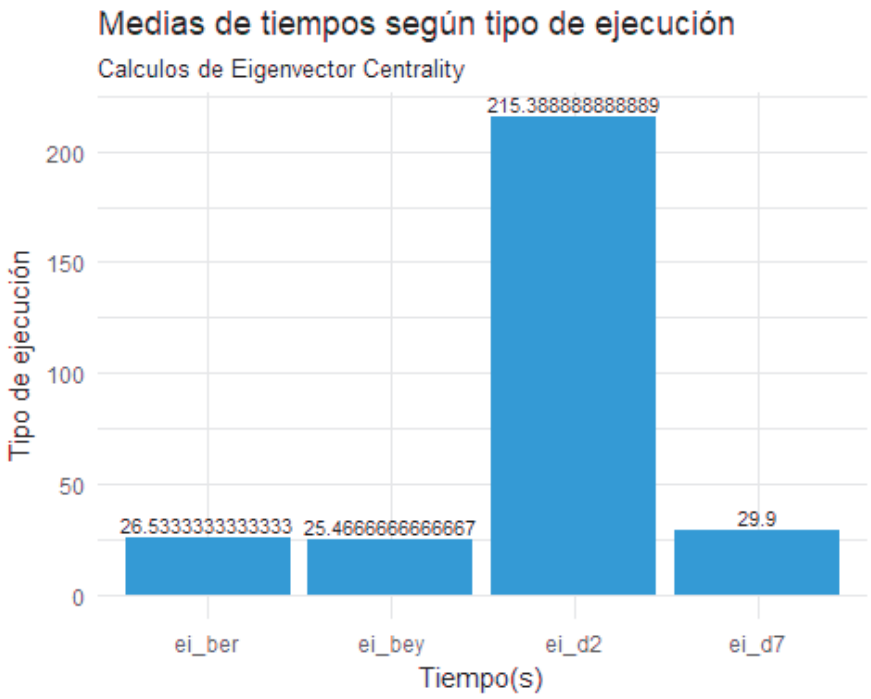
Otro detalle interesante y a priori desconcertante es que la ejecución en la que se dispone de todo el clúster sea más lenta que la que se limita a 2 ejecutores. La razón por la que esto ocurre es porque, aunque se dispongan de todos los nodos, el cálculo se ejecuta como máximo en 2 nodos. Esto hace que se pierda tiempo en crear y eliminar ejecutores que al final no se usan y el trabajo no se paraleliza lo suficiente. Esto es debido a la implementación del algoritmo, y cambiarlo para que funcione mejor ya sería un trabajo de fin de grado por sí solo. En la siguiente imagen se muestra la web de administración de Spark durante la ejecución con todo el clúster disponible y se puede ver como sólo existen 2 executors que han realizado las tareas y muchos otros que se crean luego destruyen:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks
driver	aton.darwin.cicei.com:39005	Active	0	0.0 B / 3.2 GB	0.0 B	0	0	0	0
1	isis.darwin.cicei.com:46393	Active	12	351.6 MB / 3.2 GB	0.0 B	1	1	0	16
2	anubis.darwin.cicei.com:39073	Active	12	352.9 MB / 3.2 GB	0.0 B	1	0	0	16
4	nut.darwin.cicei.com:40691	Dead	0	0.0 B / 3.2 GB	0.0 B	1	0	0	1
5	maat.darwin.cicei.com:33379	Dead	0	0.0 B / 3.2 GB	0.0 B	1	0	0	0
7	horus.darwin.cicei.com:33633	Dead	0	0.0 B / 3.2 GB	0.0 B	1	0	0	0

18 Tabla de executors con Betweenness Centrality. Fuente: elaboración propia

Comprobado el funcionamiento del clúster y el algoritmo de Betweenness Centrality, crearemos un gráfico de barras para mostrar las medias de tiempos para los diferentes tipos de ejecución de Eigenvector Centrality.

```
g_ei <- ggplot(data=table_ei, aes(x=V1, y=V2)) +
           geom_bar(stat="identity", fill="steelblue") +
           geom_text(aes(label=V2), vjust=-0.3, size=3) +
           labs(title="Medias de tiempos según tipo de ejecución",
                subtitle="Calculos de Eigenvector Centrality",
                x="Tiempo(s)", y="Tipo de ejecución") +
           theme_minimal()
```



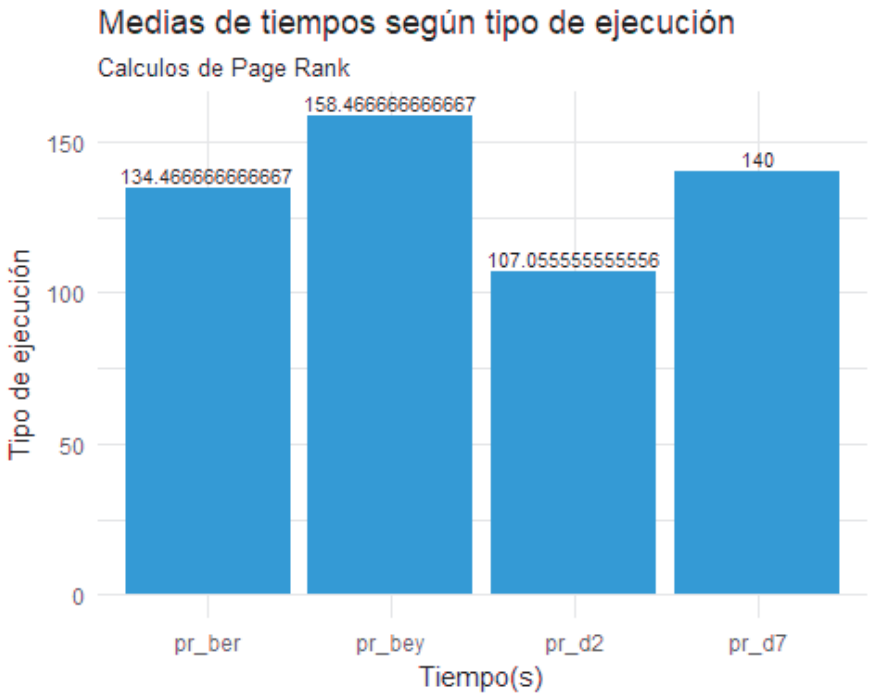
19 Medias de tiempos para Betweenness Centrality. Fuente: elaboración propia

Claramente, desde el primer vistazo se puede concluir que para Eigenvector Centrality si que es importante el número de nodos de los que dispone el clúster. Gracias a cómo está hecho el algoritmo podemos ver un speedup real gracias a la paralelización de los procesos. Los tiempos de ejecución medios para realizar esta métrica sobre un fichero del dataset son 7 veces más rápidos cuando se dispone del clúster completo que cuando se limita a 2 ejecutores.

A diferencia con Betweenness Centrality, podemos ver que el algoritmo parece que no cachea datos de la misma manera. Esto se puede ver en que las medias de tiempos entre una ejecución individual y una con varios ficheros seguidos no cambian prácticamente nada. Ni siquiera hay cambios en los tiempos de cómputo sin Dynamic Resource Allocation.

Por último, crearemos un gráfico de barras para mostrar las medias de tiempos para los diferentes tipos de ejecución del algoritmo Page Rank:

```
g_pr <- ggplot(data=table_pr, aes(x=V1, y=V2)) +
          geom_bar(stat="identity", fill="steelblue") +
          geom_text(aes(label=V2), vjust=-0.3, size=3) +
          labs(title="Medias de tiempos según tipo de ejecución",
               subtitle="Cálculos de Page Rank",
               x="Tiempo(s)", y="Tipo de ejecución") +
          theme_minimal()
```



20 Medias de tiempos para Page Rank. Fuente: elaboración propia

Con este algoritmo parece que pasa algo parecido a lo que ocurría con Betweenness Centrality. El gráfico indica que tarda menos una ejecución individual con 2 ejecutores como máximo que una con 2 ejecutores como mínimo. Además, podemos observar que la ejecución sin Dynamic Resource Allocation tarda más que la ejecución con esta funcionalidad. Aun así, no se ven grandes diferencias entre los tipos para Page Rank.

Para ver el código completo de la creación de estos gráficos puede pinchar aquí. Es importante recalcar que estas conclusiones deben de ser revisadas tras futuras investigaciones en el funcionamiento de los procesos de Apache Hadoop y Spark, así como, un profundo análisis de los algoritmos de las métricas usados en estas pruebas.

3. CONCLUSIONES

Antes de comenzar este proyecto, conocía lo que era el Big Data y la importancia que está adquiriendo día a día. Una de las razones por las que elegí este trabajo fue por esto mismo, por aprender sobre un paradigma de la informática que no se había tocado durante la carrera pero que es de gran importancia hoy en día y mucho más que lo será en el futuro. A medida que han ido pasando los meses he ido observando todas las noticias relacionadas con este mundo e ido comprobando que realmente esta puerta está abierta a personas que sepan sobre cómo tratar los datos, cómo procesarlos, con qué herramientas y cómo sacar información relevante de ellos. El trabajo, al fin y al cabo, me ha introducido en este mundo del cuál puedo llegar a formar parte de manera profesional en el futuro.

Durante el transcurso del tiempo dedicado a realizar este trabajo he comprobado que las tecnologías que hemos usado están realmente en plena fase de desarrollo. Muchas de las compañías que trabajan con Big Data usan Apache Hadoop como base de su arquitectura. Aun así, en mi caso, trabajando con grafos, he encontrado en la red muy pocos trabajos, proyectos, artículos o simplemente foros de consultas sobre el tema. Esto me indica que realmente se está empezando a usar y todavía no hay demasiadas personas trabajando con estas tecnologías. Un ejemplo de esto es que solamente existe una librería para calcular Betweenness Centrality con GraphX y esta misma no parece que haya tenido ningún tipo de soporte o cambio tras su creación e implementación.

Otro problema con el que nos topamos, y que se podría haber visto venir, eran las características del clúster del CICEI en el que íbamos a implementar Hadoop. Al comienzo del proyecto el clúster disponía de únicamente 2GB de memoria y prácticamente cualquier carga de trabajo que se le asignara resultaba en un error por falta de memoria. Con el tiempo se realizó la compra de nuevos módulos de memoria para conseguir llegar al estado actual de tener 8GB de RAM por nodo, pero sin cambiar los procesadores seguiríamos teniendo solamente 2 núcleos de procesamiento por nodo y muy lentos. Pese a la mejora sustancial en la memoria, se tuvo que seguir haciendo cambios y retocando los códigos de los programas para evitar este tipo de errores y que los procesos durasen demasiado tiempo. Esto indica que para montar un clúster para trabajar con Big Data es muy importante no tomar servidores desfasados y poco potentes, porque realmente se pierde mucho tiempo en hacer que funcionen y nunca llegarán a trabajar tan bien como deberían.

Aún así me siento satisfecho con el trabajo realizado y los resultados obtenidos. Pese a las dificultades se han conseguido los objetivos de implementar y probar la tecnología. Añadiendo a estos conocimientos aprendidos muchos otros relacionados con el Big Data, el trabajo en una infraestructura real y la búsqueda de soluciones a problemas.

Por último, este trabajo abre la puerta a quien quiera comenzar con estas tecnologías desde cero, y deja varios frentes en los que avanzar en conocimiento. En trabajos futuros se puede profundizar mucho más en las diferentes opciones de configuración y funcionamiento interno de los servicios, por ejemplo. También, se puede investigar en la forma de crear algoritmos o métodos para aumentar la paralelización de las métricas empleadas en los grafos, o incluso, probar otras librerías y complementos existentes para Hadoop. Como queda demostrado, aquí solamente se ha rascado la superficie de lo que realmente son estas tecnologías Big Data y lo que pueden llegar a conseguir con el tiempo y los medios.

4. BIBLIOGRAFÍA

1	Definición Big Data	https://en.wikipedia.org/wiki/Big_data
2	Características Big Data	https://www.iebschool.com/blog/5-vs-del-big-data/#
3	Definición Eigenvector Centrality	https://en.wikipedia.org/wiki/Eigenvector_centrality
4	Eigenvector y Betweenness	http://www.javierllinares.es/2015/01/07/eigenvector-y-betweenness/
5	Definición de Betweenness Centrality	https://en.wikipedia.org/wiki/Betweenness_centrality
6	Definición de grafo	https://es.wikipedia.org/wiki/Grafo
7	Definición de graphlets	https://en.wikipedia.org/wiki/Graphlets
8	Definición de multigrafo	https://es.wikipedia.org/wiki/Multigrafo
9	Definición de grafo dirigido	https://es.wikipedia.org/wiki/Grafo_dirigido
10	Documentación oficial Apache Hadoop	https://hadoop.apache.org/docs/r3.1.1/
11	Documentación oficial Spark	https://spark.apache.org/docs/latest/
12	Configuración de un servicio Shuffle externo	https://spark.apache.org/docs/latest/running-on-yarn.html#configuring-the-external-shuffle-service
13	Guía de programación de GraphX	https://spark.apache.org/docs/latest/graphx-programming-guide.html
14	Curso Monta un cluster Hadoop Big Data desde cero	https://www.udemy.com/monta-un-cluster-hadoop-big-data-desde-cero/
15	Librería Sparking-graph	https://github.com/sparkling-graph/sparkling-graph

16	Documentación Sparking-graph	https://sparkling-graph.readthedocs.io/en/latest/
17	Librería Betweenness Centrality	https://github.com/dmarcous/spark-betweenness
18	Guía de uso paquete leaflet	https://rstudio.github.io/leaflet/
19	Lista de tipos mapas	http://leaflet-extras.github.io/leaflet-providers/preview/
20	Sistema Operativo CentOS	https://www.centos.org/download/
21	Software de virtualización Virtual Box	https://www.virtualbox.org/
22	JDKs de Java	https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
23	Compilador Scala	https://www.scala-sbt.org/index.html
24	Ejemplo de uso de Apache Spark Graphx en Scala	https://mapr.com/blog/how-get-started-using-apache-spark-graphx-scala/
25	Qué es GraphX y cómo se utiliza	https://www.edureka.co/blog/spark-graphx/
26	Uso del comando spark-submit	https://www.alibabacloud.com/help/doc-detail/28124.htm
27	Puertos de acceso Hadoop	https://blog.cloudera.com/blog/2009/08/hadoop-default-ports-quick-reference/
28	Diferencias entre Hadoop v1 y v2	https://www.hdfstutorial.com/blog/hadoop-1-vs-hadoop-2-differences/
29	Optimización de Yarn	https://www.cloudera.com/documentation/enterprise/5-4-x/topics/cdh_ig_yarn_tuning.html
30	Optimización de recursos para aplicaciones	https://es.slideshare.net/cloudera/top-5-mistakes-to-avoid-when-writing-apache-spark-applications
31	Optimización Yarn	https://dzone.com/articles/apache-spark-on-yarn-resource-planning

32	Mapa en R de ejemplo con leaflet	http://www.r-graph-gallery.com/19-map-leafletr/
33	Colores y paletas en R	https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatsheet.pdf
34	Paletas de más de 9 colores.	https://www.r-graph-gallery.com/40-rcolorbrewer-get-a-longer-palette/