Pontificia Universidad Católica Argentina

Facultad de Ciencias Fisicomatemáticas e Ingeniería

Ingeniería Informática

Trabajo Final

Benchmark de Big Data Utilizando Software Libre y Hardware de Bajo Costo

Autor: Marcelo Gaston Estol

Tutor: Ricardo Di Pasquale

Teléfono de contacto: +5411588467288

Correo electrónico: gastonestol@gmail.com

Correo electrónico del tutor: rdipasquale@uca.edu.ar

Índice de Contenidos

Motivación

El campo de la computación distribuida ha tenido una gran influencia sobre los sistemas actuales. La capacidad de realizar procesamientos complejos bajo el marco de la computación distribuida sobre grandes cantidades de datos, en lo que se llamaría "Big Data", ha dado un nuevo valor a los datos, solucionado algunas problemáticas comúnmente encontradas en sistemas con gran cantidad de flujo de datos online.

El área de la computación denominada "Big Data" está cobrando un gran auge en los sistemas actuales, por esa razón parece oportuno realizar una investigación de las diferente tecnologías comúnmente usadas y sus oportunidades de aplicación.

Este trabajo se concentra en tomar un algoritmo y llevarlo a un programa funcional bajo distintos softwares de Apache, en concreto Hadoop y Spark. Estos sistemas son usados comúnmente por organizaciones con plataformas online para realizar pruebas de aceptación de usuarios, realizar consultas no estructuradas de forma ágil y eficaz, montar motores de bases de datos sobre registros del sistema y otros archivos no estructurados, entre otras áreas de aplicación.

Se investigaron otras tecnologías diferentes a Spark y Hadoop, pero se ha decidido no incluir las mismas en este documento. Entre ellas se puede mencionar Hbase, Titan DB, Amazon MapReduce (entre otros servicios), Spatial and Graph.

Introducción

Los datos forman una parte indispensable en cualquier aspecto del mundo actual. Desde organizaciones sin fines de lucro hasta gobiernos; pasando por empresas, corporaciones e incluso personas individuales, para todos ellos los datos tienen importancia, ya que proveen una fuente de la cual es posible extraer patrones de comportamiento, relaciones, realizar inteligencia de negocio (BI), obtener métricas, reportes; para ser más abarcativos, es posible extraer información.

El mundo cada vez está más conectado a la internet, cada vez más personas tienen uno o más dispositivos que están conectados a la red global, los cuales envían y reciben datos de aplicaciones y otros dispositivos. La recepción de datos ha crecido exponencialmente los últimos años, lo cual revela la ineficacia de las bases de datos tradicionales para obtener información. Incluso el almacenarlos ha sido un desafío de una nueva era.

Una de las primeras organizaciones en toparse con este problema fue Google.inc al implementar su navegador. El navegador de Google utiliza entre otras herramientas un algoritmo de fondo llamado Page Rank [2][7], el cual explora la web y asigna un puntaje en base a determinadas reglas que establece una calidad a cada página web encontrada. Luego al realizarse una búsqueda se utiliza el resultado del algoritmo para retornar los resultados de la búsqueda. Page Rank utiliza una sistema de archivos

distribuido para almacenar los resultados de la exploracion y luego mediante un framework Map/Reduce ejecuta la implementación del algoritmo.

Entre otras organizaciones que han implementado Map/Reduce de forma eficiente podemos mencionar, como casos de éxito, a Facebook, Twitter, MercadoLibre, Linkedin, entre otras muchas más.

Al mencionar el algoritmo de Page Rank, se incursiona en un grupo de algoritmos de la matemática llamados algoritmos de grafos. Estos algoritmos en general han sido pensados para ser implementados de forma secuencial y no distribuida. Este documento incursiona en la implementación de un algoritmo del grupo de algoritmos basados en los ideados por Edsger Dijkstra, específicamente el algoritmo de camino más corto con una sola fuente, un problema lineal el cual tiene una variante bajo el framework Map/Reduce [2].

Existen diversas formas de implementar un algoritmo de grafos utilizando Map/Reduce. El contenido principal de este documento es exponer dos de ellas, ambas basadas en el sistema de archivos distribuidos de Apache: Hadoop.

Una alternativa es utilizando Apache Spark, este es un software para procesamiento de grandes cantidades de datos, en poco tiempo que es montable sobre la arquitectura Hadoop.

La segunda alternativa es utilizar Hadoop solamente. En este contexto es posible crear y ejecutar un trabajo Map/Reduce en lo que se llamaría bajo nivel de Hadoop.

Introducción a Big Data

Cuando se hace referencia a Big Data (“datos masivos” o “gran cantidad de datos” por su traducción al español) se está hablando de una modalidad bastante reciente en lo que sería análisis de datos. La razón de esto es la escalabilidad de los sistemas actuales con la posibilidad de analizar millones de datos para encontrar patrones de comportamiento y actuar acorde a ello. Esto quiere decir ser proactivos antes que reactivos, anticiparse a los eventos que puedan llegar a ocurrir para poder obtener un mayor beneficio.

El tamaño de los datos que entra en la clasificación de Big Data es algo cambiante, pueden ser unos pocos Gigabytes a varios Terabytes. El desafío principal está en procesar datos no estructurados como archivos de log de un servidor, aplicación o sistema y las relaciones entre los datos de una gran cantidad de archivos no estructurados.

No hay una definición precisa de Big Data, las caracteristicas varian entre autores y organizaciones, pero hay una puesta en común. Gartner [13], otorga tres características encontradas a lo largo de proyectos de Big Data. A su vez varios autores han propuesto otras características que tienen que ver más con la implementación de Big Data que con su definición.

Características:

1. Volumen: no toma muestras, solo lleva un registro de los eventos (Logs).
2. Velocidad: comúnmente está disponible en tiempo real.
3. Variedad: texto, imágenes, audio, video, completitud de datos.
4. Variabilidad: inconsistencia en los datos.
5. Veracidad: la calidad de los datos es variable, su precisión depende de la fuente que los provee.
6. Complejidad: puede ser muy compleja, especialmente con gran cantidad de datos. Los datos pueden estar correlacionados entre sí.

A su vez se puede encontrar otras dos propiedades en común:

1. Machine Learning: detección de patrones de comportamiento.
2. Digital Footprint: es algo de costo gratuito ofrecido por el uso de sistemas digitales.

Big Data es distinto a Business Intelligence (BI) en muchos aspectos. Principalmente BI utiliza estadística descriptiva sobre datos de alto grado de densidad para medir enfoques y modalidades. En cambio, Big Data utiliza estadística inferencial y conceptos de identificación no lineal para inferir reglas, regresiones, relaciones no lineales y efectos causales a partir de gran cantidad de datos con información de baja densidad. Esto permite revelar relaciones, dependencias, y predicciones de resultados y comportamientos.

Algoritmo de Dijkstra

Es un algoritmo de grafos ideado por Edsger Wybe Dijkstra (11 de mayo 1930 - 6 de agosto 2002) en 1959 para determinar el camino más corto de un nodo a todos los otros nodos de un grafo. Dijkstra fue un científico matemático y computacional de Holanda que contribuyó mucho a la comunidad científica, y sus descubrimientos son muy utilizados en diversos campos hoy en día.

Caso de estudio: Single Source Shortest Path (SSSP)

El paso más corto de un nodo a los otros nodos de un grafo es un problema común al realizar análisis de datos, un ejemplo podría ser llegar de una dirección a otra.

Existen dos versiones:

- Utilizando una cola de prioridad que provee dos funcionalidades: adicionar() que agrega nodos con prioridad y extraer_mínimo(). Esto provee mayor velocidad de cómputo que una cola común.

La complejidad computacional en el peor caso es ***O(|A|+|V| log(|V|)).***

```
DIJKSTRA (Grafo G, nodo_fuente s)
       para u ∈ V[G] hacer
           distancia[u] = INFINITO
           padre[u] = NULL
           visto[u] = false
       distancia[s] = 0
       adicionar (cola, (s, distancia[s]))
       mientras que cola no es vacía hacer
           u = extraer_mínimo(cola)
           visto[u] = true
           para todos v ∈ adyacencia[u] hacer
               si no visto[v] y distancia[v] > distancia[u] + peso (u, v) hacer
                   distancia[v] = distancia[u] + peso (u, v)
```

```
                padre[v] = u
                adicionar(cola,(v, distancia[v]))
```

- Sin embargo existe una versión que no utiliza una cola de prioridad.

La complejidad computacional en el peor caso es $O(|V|^2)$.

```
DIJKSTRA (Grafo G, nodo_salida s)
  //Usaremos un vector para guardar las distancias del nodo salida al resto
  entero distancia[n]
  //Inicializamos el vector con distancias iniciales
  booleano visto[n]
   //vector de booleanos para controlar los vértices de los que ya tenemos la
distancia mínima
  para cada w ∈ V[G] hacer
     Si (no existe arista entre s y w) entonces
            distancia[w] = Infinito //puedes marcar la casilla con un -1 por
ejemplo
     Si_no
         distancia[w] = peso (s, w)
     fin si
  fin para
  distancia[s] = 0
  visto[s] = cierto
  //n es el número de vértices que tiene el Grafo
  mientras que (no_estén_vistos_todos) hacer
     vértice = obtener_el_mínimo_del_vector distancia y que no esté visto;
     visto[vértice] = cierto;
     para cada w ∈ sucesores (G, vértice) hacer
         si distancia[w]>distancia[vértice]+peso (vértice, w) entonces
            distancia[w] = distancia[vértice]+peso (vértice, w)
         fin si
     fin para
  fin mientras
fin función.
```

Este enfoque es de extrema eficiencia, su desventaja crítica es ser un algoritmo centralizado, pensado para ser ejecutado secuencialmente.

Framework Map/Reduce

Antes de poder dar una versión del algoritmo de Dijkstra bajo Map/Reduce, debemos definir en qué consta el Framework.

El Framework, como su nombre lo indica, se compone de dos partes principales, el Mapper y el Reducer y otras dos complementarias, el Partitioner y el Combiner.

- Mapper

Se encarga de crear pares tipo [clave,valor] y emitirlos.

- Combiner

Toma los valores con la misma clave del mismo mapper y los agrupa para aportar información más resumida, son pequeños reducers.

- Partitioner

Distribuye los datos en particiones para determinar la cantidad de reducers paralelos.

- Reducer

Toma los pares [clave,valor] y realiza alguna acción acorde, generalmente todos los pares con la misma clave llegan a un mismo reducer. Este comportamiento es modificable a través de los Partitiones.

El programa es llevado adelante por un componente principal llamado Driver cuya responsabilidad es la de llevar adelante el trabajo Map/Reduce definiendo: el input, el output, que mapper asigna, qué combiner actúa, partitioner y reducer, entre otras propiedades del trabajo.

El siguiente ejemplo ilustra la arquitectura del framework.

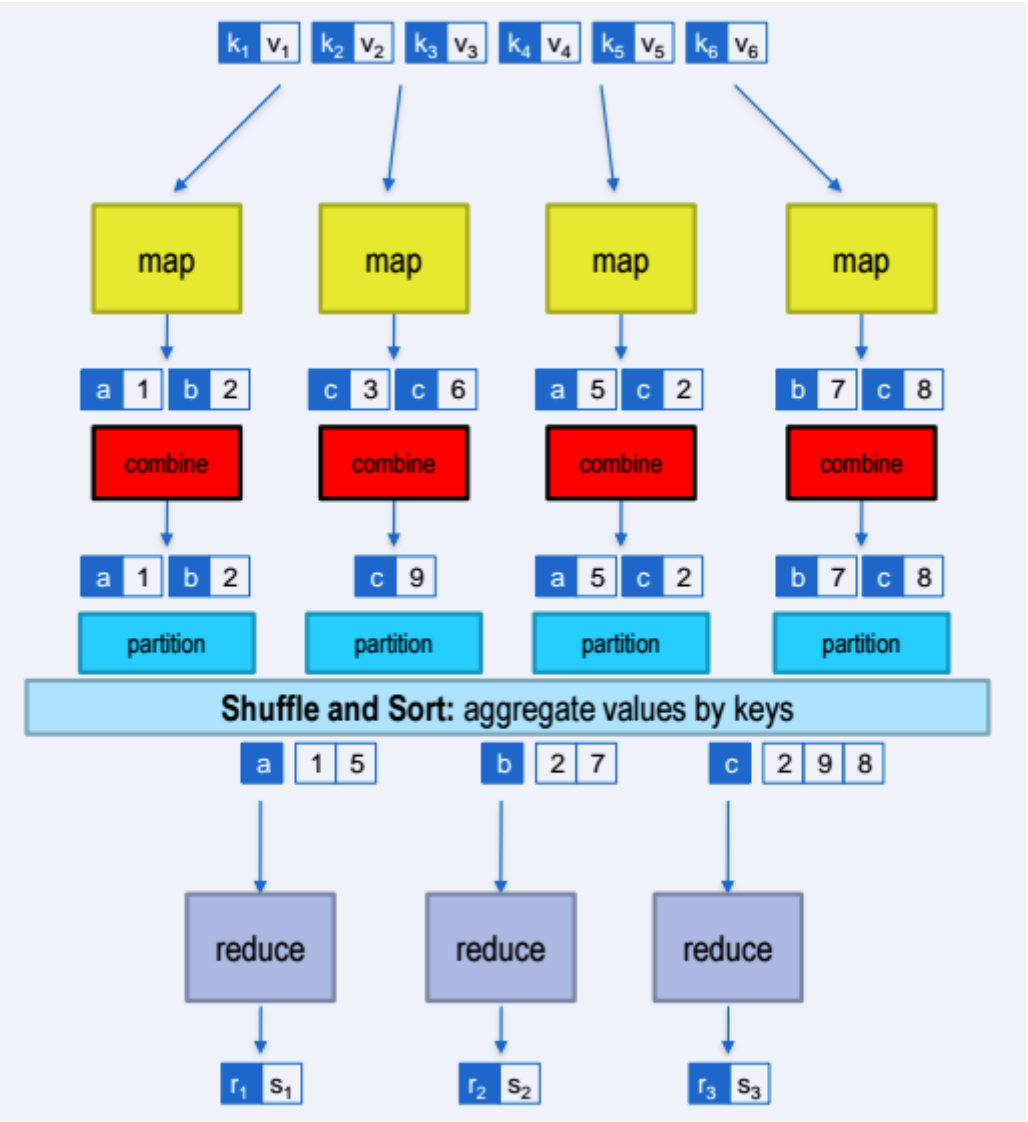

Figura 4: Arquitectura Map/Reduce [14]

Single Source Shortest Path bajo el framework Map/Reduce

Bajo el framework Map/Reduce Dijkstra se expresa de la siguiente manera, se hará omisión a los combiners y partitioners, ya que no entran bajo la lupa de estudio en esta instancia.

```
1. clase Mapper
      a. método Map(nid n,nodo N)
          i.   d  = N.distancia
         ii.   Emit(nid n,N)
          iii. para cada nodeid m ∈ N.ListaDeAdjacencia hacer:
                  1. Emit(nid m, d+1)
         iv.   fin para
      b. fin método
2. fin clase

3. clase Reducer
      a. método Reduce(nid m,[d1 ,d2,d3 ,....])
          i.   distanciaMínima <- infinito
         ii.   M = null
          iii. para cada d ∈ lista [d1 ,d2,d3 ,....] hacer
                  1. Si esNodo(d) entonces
                        a. M = d
                  2. Si_no si d < distanciaMínima entonces
                        a. distanciaMínima = d
                  3. fin si
         iv.   fin para
          v.   M.distancia = distanciaMínima
         vi.   Emit(nid m,node M)
      b. fin método
4. fin clase
```

Este algoritmo es de poca eficiencia comparado al algoritmo lineal original, su fortaleza reside en la posibilidad de realizar procesamiento paralelo. Si a esta característica se suma un data set de gran tamaño, se puede demostrar que la eficiencia para este caso en particular es sumamente superior al original.

La forma de trabajar de este algoritmo es a través de iteraciones sucesivas. Un programa principal es el que lleva adelante las iteraciones, cada iteración contiene un

trabajo Map/Reduce. El significado de cada iteración es expandir la frontera de conocimiento en 1 grado.

De esta forma la frontera se expandirá progresivamente.

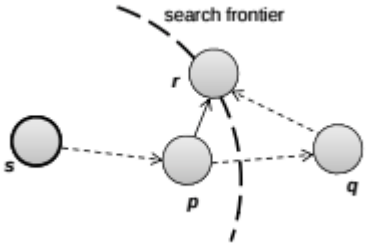

Figura 5

Frontera de conocimiento en expansión, al aumentarla se encuentra un camino más corto de S a Q.

La frontera es de gran importancia ya que, mediante esta técnica, la idea es encontrar el camino más corto de forma progresiva, como demuestra la figura 6, donde se puede ver que en la 5ta iteración se encuentra el camino más corto de **n1** a **n6.**

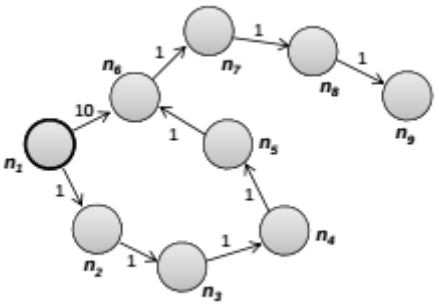

Figura 6, Camino más corto de dos caminos.

La cantidad de iteraciones es otro problema a tratar. Existen distintas versiones que especifican que la mejor cantidad es equivalente al diámetro del grafo [2]. Esto conlleva a otro problema que es encontrar el diámetro del grafo bajo el framework Map/Reduce utilizando el algoritmo HADI [10] en caso de que no se conozca. Para ello

se pueden realizar dos trabajo Map/Reduce consecutivos, uno para encontrar el diámetro y otro para el algoritmo de Dijkstra.

Siendo el diámetro una dimensión que puede variar, se ha adaptado el algoritmo para que tome una estrategia de recorrido mediante el coloreado de sus nodos.

En consecuencia se itera sobre el algoritmo mientras el grafo no se haya recorrido completamente. Por su puesto esto puede elevar cuestiones como grafos infinitos con una estructura muy plana y diámetro de gran valor. Para estos casos se sugiere calcular el diámetro para poder tener una estimativa de la cantidad de procesamiento necesaria.

Arquitecturas a comparar

La primera arquitectura a comparar es la arquitectura base. Sobre esta arquitectura se montarán todos los demás enfoques. La idea es tener a Hadoop de fondo para poder dar un juicio de valor sobre si es preferible tener un enfoque minimalista o utilizar alguna herramienta extra para poder simplificar algunos aspectos.

Hadoop

Es un proyecto open-source de Apache para el desarrollo de software confiable, escalable y distribuido. Este proyecto cuenta con un framework de desarrollo para trabajar con grandes conjuntos de datos a través de clusters de computadoras. Está

diseñado para escalar de un solo servidor a miles de servidores, cada uno ofreciendo capacidad de cómputo y almacenamiento local.

En lugar de utilizar más hardware para obtener la máxima disponibilidad, las librerías del proyecto están diseñadas para detectar y resolver las fallas a nivel de aplicación, pudiendo ofrecer máxima disponibilidad sobre un conjunto de clusters de computadoras cada una de las cuales es propensa a errores.

Dentro de Hadoop se encuentran distintos módulos.

- **Hadoop Common:** Es el módulo base del sistema que provee soporte a los demás módulos y softwares.
- **Hadoop Distributed File System (HDFS™):**Un sistema distribuido que provee acceso de datos a nivel aplicación a un alto rendimiento.
- **Hadoop YARN:** Es un gestionador de trabajos.
- **Hadoop MapReduce:** es un sistema basado para el procesamiento paralelo de grandes conjuntos de datos.

El enfoque se basará en un trabajo Map/Reduce que obtendrá la fuente de datos de HDFS y utiliza Hadoop como herramienta de ejecución.

Spark

Es un software de Apache para procesamiento rápido y general en un cluster de computadoras. Provee APIs en Java, Scala, Python y R, además tiene un motor que soporta ejecuciones en grafos. También contiene librerías de SQL, Machine Learning,

Streaming y por último GraphX, en la cual se implementará la versión distribuida de Dijkstra.

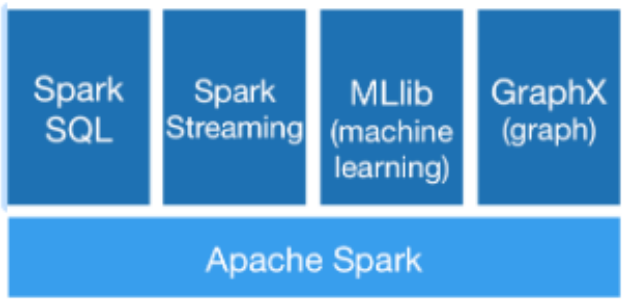

Figura 5 Arquitectura utilizando Spark.

La versatilidad de Spark se encuentra en poder agrupar en una aplicación distintas fuentes de datos provenientes de diferentes softwares y arquitecturas. Esto puede parecer a simple vista muy pesado en cuanto a procesamiento, pero Spark está pensado específicamente para casos complejos en cuanto a manejo de grandes fuentes de datos y procesamiento paralelo.

Spark tiene el contexto de ejecución sobre Hadoop para crear y ejecutar los trabajos, los cuales son monitoreables desde la consola de Spark o si se desea desde la consola de YARN.

Los creador de Spark afirman que su enfoque on memory tiene un Speedup de 100x más que Hadoop Map/Reduce o 10x en disco.

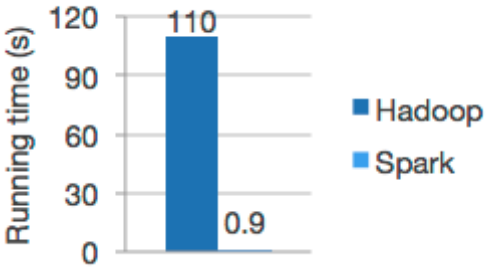

Figura 6 Tiempos de ejecución afirmados por Spark.

Data Sets

Pokec

Es la red social más famosa de Slovakia, su popularidad no se ha visto afectada ante la llegada de Facebook. Por más de 10 años ha conectado a más 1.6 millones de personas. El dataset contiene datos anónimos de la red aunque es posible visualizar las relaciones entre personas, sexo, edad, intereses, educación ,etc. Los datos del perfil están en eslovaco así que no se le dará mucho interés.

En este caso se utilizará la fuente de datos proveída por la **Universidad de Standford** [14].

Las amistades son orientadas. En particular procesaremos un grafo con el formato

```
node_id_1 <tab> node_id_2
```

Donde se señala una relación direccional del node_id_1 al node_id_2.

El diámetro de este grafo según Standford es de 11. Aunque en realidad se encontró que es de 13 enlaces.

El archivo lleva el nombre de **soc-pokec-relationships.txt** y ocupa un espacio de 423,9 MB.

Wikipedia

El trabajo realizado por **Henry Haselgrove** [11], consiste en la creación de un grafo de relación entre documentos de la Wikipedia en su edición en inglés. El trabajo consistió en tomar la base de datos en SQL y crear dos archivos.

links-simple-sorted.zip (323 MB, 1G descomprimido) que contiene las relaciones entre documentos, el formato es el siguiente:

```
id0: id1 id2 id3
```

Si se especifica los arcos dirigidos de id0 a los demás ids.

titles-sorted.zip (28 MB) contiene los rótulos de cada nodo, la línea N contiene el nombre del nodo N.

Implementaciones de SSSP

En el siguiente punto se incursionará en la programación asociada a la implementación del algoritmo SSSP en las distintas arquitecturas mencionadas.

Hadoop

La siguiente implementación se realiza utilizando Hadoop únicamente. La forma de ejecutar un trabajo Map/Reduce sobre un sistema de archivos HDFS (Hadoop Distributed File System), en este caso, es a través de los scripts proveídos por Hadoop. Para ello se debe crear un programa en Java utilizando las dependencias de Hadoop y codificando tanto el driver como el mapper y el reducer principalmente.

El código mostrado a continuación es una variante del código ubicado en HadoopTutorials [15].

El repositorio con el código completo se puede encontrar en el siguiente link SSSP [6]

El programa recibe uno o más archivos de una fuente de datos en hdfs donde cada línea de los archivos tienen la siguiente estructura:

```
id0: id1 id2 id3
```

Lo cual define que el `id0` tiene un arco con dirección hacia los nodos `id1 id2 id3`

Los códigos del **Driver, Mapper y Reducer** se encuentran en el anexo A, incisos 1.1, 1.2 y 1.3 respectivamente.

Spark

La implementación utilizando Spark sobre la arquitectura HDFS como fuente de datos es una de las más innovadoras, y producto estrella de Apache en la actualidad.

Su “simpleza” en comparación a utilizar Hadoop se observa con sólo ver el algoritmo SSSP escrito en Scala utilizando el framework de Spark. Uno muy similar se puede encontrar en la página oficial de Apache Spark [5].

El programa recibe desde la fuente de datos hdfs, un archivo donde cada línea contiene un arco entre dos nodos donde la dirección está marcada por el orden en que están escritos de izquierda a derecha:

```
node_id_1 <tab> node_id_2
```

El código fuente del programa escrito en Scala se puede encontrar en inciso 1.4 del Anexo A.

Se puede observar que Spark posee una API de Pregel para llevar adelante la ejecución de un algoritmo de grafos. Pregel [16] es un modelo computacional diseñado para procesar grafos de gran tamaño de forma eficiente, sobre un cluster escalable de computadoras de bajo costo y bajo un entorno a prueba de fallas. Su modelo posibilita la representación de la mayoría de los algoritmos de grafos existentes.

Fortalezas y debilidades de las arquitecturas

En esta sección se pasará a enumerar las diferentes características comunes de los enfoques analizados. Se dará un pequeño comentario sobre cada uno como conclusión al final de cada aspecto en particular.

- **Complejidad del entorno de trabajo**

Para llevar a cabo esta prueba se ha creado un proyecto de naturaleza Maven y SBT (Scala Build Tool), lo cual ayuda a resolver dependencias y complejidades de compilación. Ambas herramientas crean un empaquetado con extensión .jar, Maven para la sección de código de Hadoop en Java y SBT para la sección de código de Spark implementado en Scala.

El codificar un trabajo Map/Reduce en el caso de Hadoop implica conocer tanto conceptos generales de la metodología Map/Reduce, como también el framework de Hadoop y su flujo de trabajo. Encontrar recursos con las capacidades de desarrollar aplicaciones Map/Reduce en Hadoop es difícil. Cometer un error en este ambiente es muy fácil y costoso, por lo tanto se debe tener experiencia previa en él.

Tener conocimientos del framework no basta en Hadoop. Conocer su entorno de ejecución, las distintas y variadas configuraciones es algo deseable para poder llevar a cabo una aplicación funcional con un buen rendimiento. Quizás este último aspecto sea incluso más importante que el programa en sí.

Por otro lado, Spark posee un framework muy variado con librerías de streaming, Machine Learning, procesamiento de grafos, SQL, entre otras funcionalidades secundarias. Se pueden realizar aplicaciones para Spark en lenguajes de alto nivel, en el caso de estudio se eligió Scala por su gran compatibilidad con Spark.

Poder entender Spark en su completitud es algo complicado. Un recurso que sepa utilizar Spark correctamente es algo muy difícil de encontrar ya que tanto el lenguaje

como la complejidad de la arquitectura para poder realizar pruebas reales hacen ardua la tarea de investigación.

Como sucede con Hadoop en Spark, existe una configuración, la cual es mucho más importante que los programas en sí. Es probable que un programa en Hadoop se ejecute con una configuración medianamente optimizada. Lo mismo no es tan certero con Spark ya que el fino en los valores de configuración puede llevar a que una tarea se ejecute o no.

- **Nivel de abstracción**

Como se puede observar a lo largo del documento, Spark provee un nivel de abstracción superior sobre Hadoop. Esto quiere decir que ambos se encuentran en distintas escalas de abstracción más allá de poder proveer la misma funcionalidad.

En cuanto a codificación basta decir que lo mismo que se realizó con cuatro clases de Java en Hadoop se pudo realizar en una sola más reducida en Spark.

Esto es debido a dos aspectos:

- Lenguaje: Scala es un lenguaje de mayor nivel que Java.
- Framework: Spark posee herramientas que facilitan el cálculo y análisis de datos además de resolver diversas tareas de procesamiento específicas de por sí.

Scala es un lenguaje de scripting. Esto quiere decir que se puede crear un programa en tiempo de ejecución y realizar un cálculo en vivo sobre Spark. Así mismo Scala posee la

propiedad de poder llevarse a un archivo .jar ejecutable en Java y a su vez poder incorporar librerías de Java para su uso.

- **Puesta en producción**

La puesta en funcionamiento de ambos softwares es similar en complejidad. Hadoop posee una mayor cantidad de archivos de configuración tanto para mejorar su rendimiento como también para la exposición de sus servicios. Spark por su lado sólo contiene un solo archivo de configuración.

Ambos poseen un archivo donde están los nodos esclavos. En Hadoop para determinar los nodos maestros se debe indicar que nodo es el maestro por lo menos en dos lugares. En cambio en Spark se ubica en un solo lugar que contiene la configuración global.

Iniciar los clusters, una vez codificados los archivos de configuración, es una tarea similar en ambos.

- **Pruebas con datos reales**

Las siguientes pruebas dan una idea general de lo importante que es la configuración de ambas herramientas.

En el caso de Wikipedia ha sido imposible obtener un resultado final, llegando a 135 interacciones en Hadoop sin resultado final. Por otro lado con Spark el trabajo se ha

consumido el total de la memoria en todas las arquitecturas probadas, por lo que no se ha obtenido ninguna métrica al respecto.

Utilizando el conjunto de datos de Pokec se ha podido obtener algunos resultados, Spark en este caso ha salido favorecido obteniendo un rendimiento casi 90% superior al de Hadoop.

Las características de los equipos utilizados se detallan a continuación:

Equipos:

- LIR
    - RAM: 4GB
    - Procesador: 64bits 2.8 GHz
    - Disco: 160GB

- Mac
    - Maquina: MacBook Pro (13-inch, Mid 2012)
    - Processor: 2,5 GHz Intel Core i5
    - RAM: 16 GB 1600 MHz DDR3
    - Disco: 250 GB
    - Utilizando Spark los recursos fueron:
        - Cores: 4
        - RAM: 16GB

El cluster del LIR que se construyó a partir de los equipos utilizados en para las prácticas del laboratorio y la red del LIR A. Se escogió el LIR A ya que los switch que

posee tiene mayor capacidad que los del LIR B y la S57. Se prefirió elegir más switches de mucha más capacidad (como es el caso del LIR A) sobre una cantidad más pequeña de switches de menor capacidad, además de un mejor hardware en comparación con el laboratorio de la S57.

La arquitectura de la red del LIR A posee 3 Switch que conectan las VLAN de las computadoras entre sí, luego estos 3 Switch se conectan entre sí dando conectividad total al laboratorio, habiendo un DHCP de por medio. Esto último favorecedor para una red de enseñanza y trabajo, pero no es la adecuada para trabajos de Big Data, ya que contiene muchos saltos de red y Switch que pueden provocar colisiones y hacer más lenta la conexión.

El cluster del LIR tiene las siguientes características para ambos softwares que pueden variar de acuerdo a la cantidad de equipos en funcionamiento:

- Spark
    - Memoria disponible: 116.1 GB
    - Cores: 336
    - Nodos: 37
- Hadoop
    - Nodos: 37
    - Masters: 1
    - Capacidad: 5.57TB

Los conjuntos de datos han dado las siguientes variables:

- Pokec (diámetro = 13 => i.e 13 iteraciones)
- Wikipedia (diámetro > 130)

El resultado de las pruebas sobre el cluster del LIR y el equipo Mac, utilizando ambos softwares con los distintos conjuntos de datos mencionados, se pueden ver en los siguientes gráficos:

La figura número 7 nos muestra las distintas pruebas llevadas a cabo utilizando Spark para ejecutar el Algoritmo sobre Pokec en los distintos ámbitos. En él se puede observar una ligera ventaja del Lir sobre Mac, esto en términos económicos es muy contrario a lo esperado ya que una sola computadora es casi o más eficaz que un cluster de computadoras. Esto también puede deberse a que la red del Lir no es la adecuada para llevar a cabo este tipo de pruebas y por ende existe un bajo rendimiento del sistema en comparación a la versión centralizada del mismo.

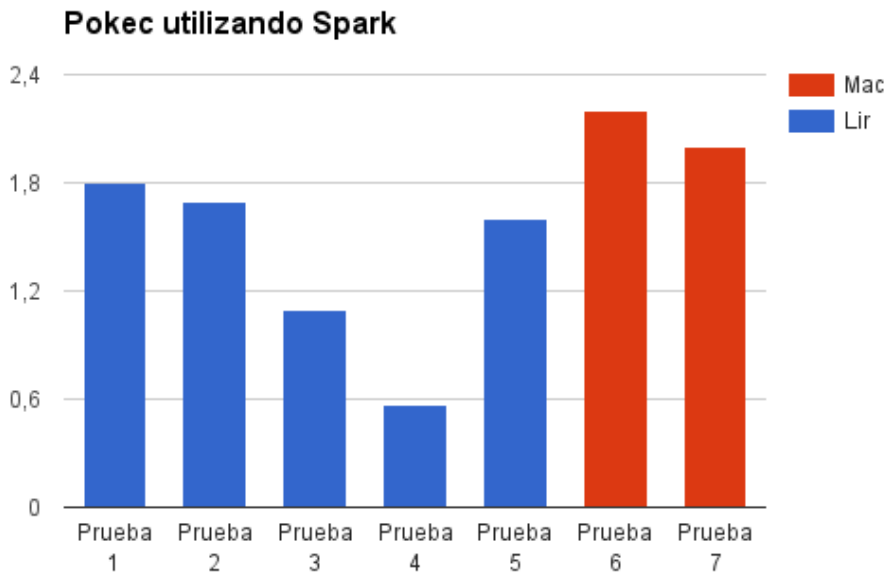

Figura 7 Pruebas de Spark sobre Pokec.

La figura 8 nos muestra con más énfasis el problema de la red del laboratorio ya que se puede ver una clara ventaja de Mac sobre el Lir en cuanto a tiempo de procesamiento.

Es también probable que la configuración de Spark y Hadoop no esté optimizada al máximo, sin embargo esto no debería afectar de forma significativa el rendimiento de las aplicaciones ejecutadas sobre el cluster del Lir.

Otro inconveniente del Lir es que existen dos Switch de 1 GB que redirigen el tráfico de las computadoras del Lir A y que a su vez estos dos Switch se encuentran conectados a otro Switch más que permite una total conexión del Lir A. Esto último es contraproducente ya que esta arquitectura es la que promueve la existencia de colisiones en la red.

El mejor tiempo del Lir se obtuvo con una configuración de 10 reducers y 37 Mappers, en cambio el mejor tiempo de la Mac se obtuvo con un equipo y un reducer.

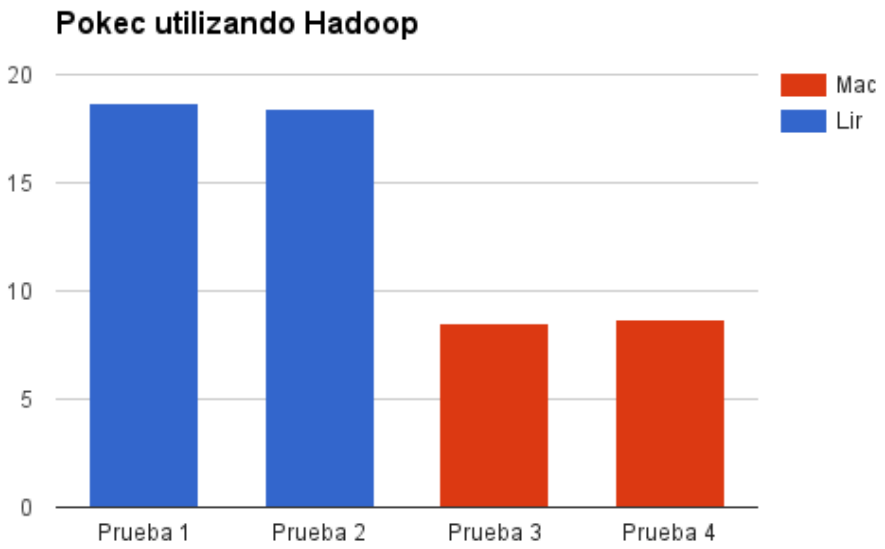

Figura 8 Pruebas de Hadoop sobre Pokec.

La figura 9 nos muestra los resultados finales de los mejores tiempos de ejecución de Spark y Hadoop sobre Pokec, utilizando los entornos de ejecución del Lir y la Mac.

En este gráfico se puede visualizar una clara ventaja de Spark sobre Hadoop en ambos entornos, pudiendo comprobarse lo mencionado por Apache en la figura 6.

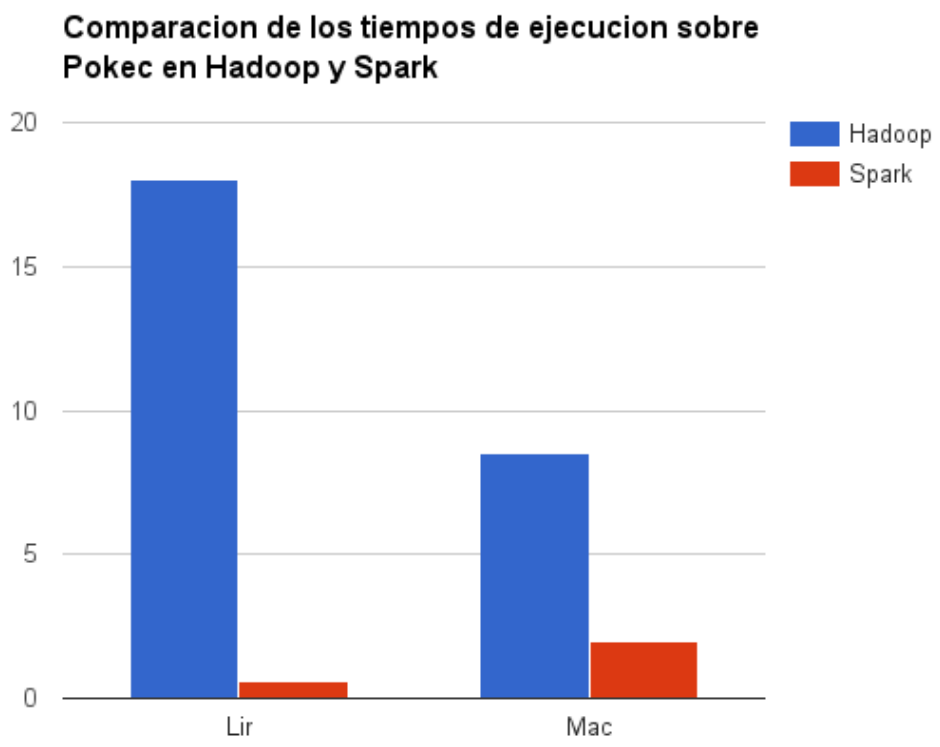

Figura 9 Mejores tiempos de Hadoop y Spark sobre el Lir y la Mac.

La obtención del Speed Up sobre Spark es oportuna en la ocasión, para ello obtuvimos los tiempos de ejecución de Spark sobre Pokec utilizando tanto un paralelismo de datos como de hardware.

Para el almacenamiento de datos, Spark utiliza una estructura de datos llamada RDD (Resilient Distributed Dataset), la cual es una estructura de datos distribuida que permite el manejo de datos sobre todo el cluster de Spark. La figura 10 nos muestra los tiempos de ejecución con distintas configuraciones de paralelismo sobre los datos.

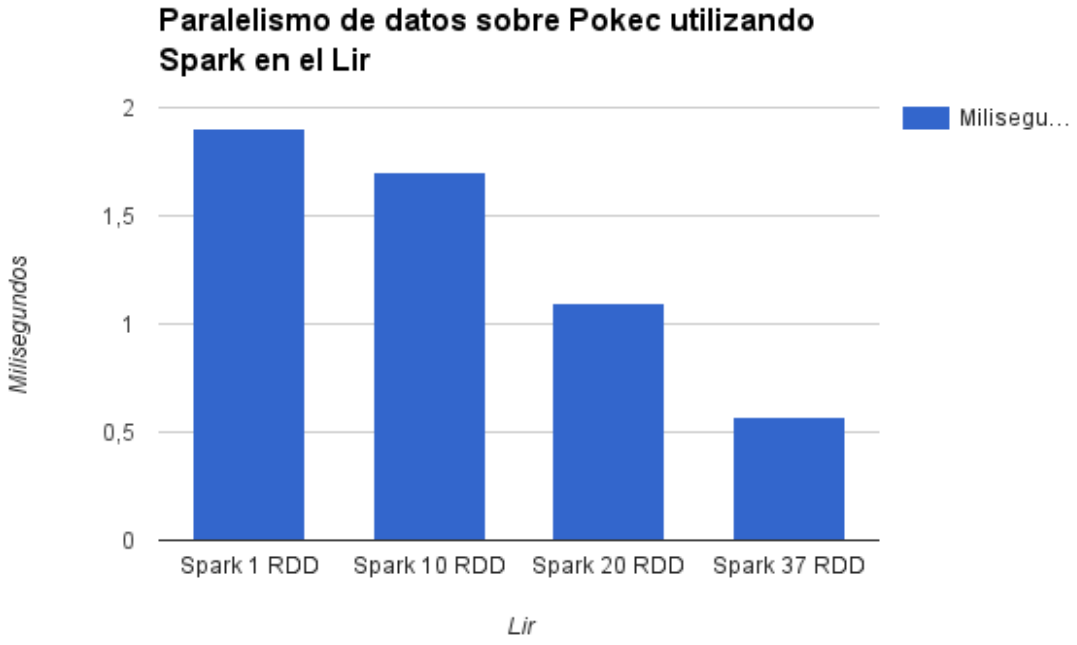

Figura 10 Tiempos de ejecución paralelizando los datos.

Para el cálculo de la Ley de Amdahl se utilizó la siguiente fórmula para estimar la proporción del programa que puede correr en paralelo:

$$P_{\text{estimated}} = \frac{\frac{1}{SU} - 1}{\frac{1}{NP} - 1}$$

Donde NP es la cantidad de nodos utilizados (paralelismo) y SU es el Speed Up asociado.

Tomando los datos de la figura 10 con un NP = 37, el SU se calcula utilizando la fórmula:

$$SU = Tiempo\ Original\ /\ Tiempo\ Mejorado$$

Si tomamos el NP original en 1, luego el tiempo original será de 1,9 minutos y el mejorado de 0,57 minutos dando por valor un SU = **3,33**.

El valor de P = **0,719**.

Para poder ver el Speed Up teórico alcanzado con N RDD se puede utilizar la siguiente fórmula:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Si N tiende a infinito el Speed Up teórico sería de **3,56**.

Para la paralelización del hardware se acotó la cantidad de esclavos utilizados en el cluster, utilizando 1 y 37 esclavos para marcar la diferencia. La figura 11 muestra los resultados de las ejecuciones. Cabe mencionar que se utilizaron 37 RDD tanto en 1 equipo como en 37.

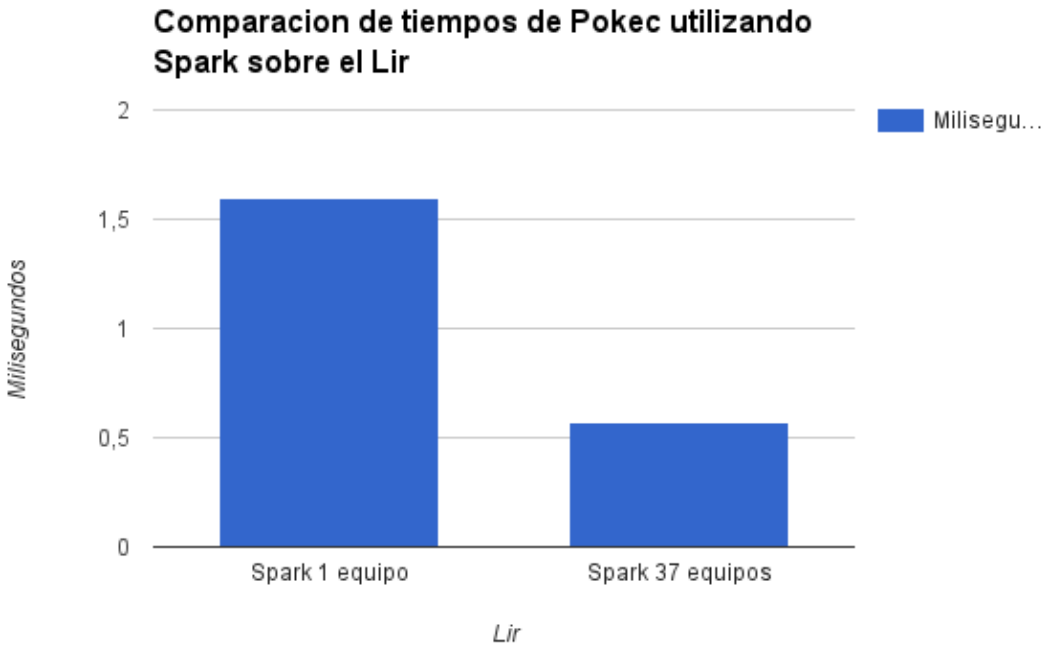

Figura 11 Tiempos de ejecución paralelizando hardware.

Se puede ver que la diferencia de rendimiento no es tan notable en el último gráfico. Esto se debe a que la red del Lir no está preparada para este tipo de trabajos en red, ya mencionado anteriormente.

Utilizando las fórmulas indicadas, procedemos a calcular el Speed Up tomando los tiempos en 1 equipo (1,6 minutos) y en 37 equipos (0,57 minutos). El mismo da por resultado SU = **2,80**.

Luego se procede al cálculo del P estimado, el cual tendrá un valor de P = **0,66**.

Finalmente el Speed Up teórico con N tendiendo a infinito se materializa en S(N)= **2,94**.

La mejora total sobre ambos ambientes y softwares ha sido la siguiente:

- Lir: $18,43\ minutos\ /0,57\ minutos\ = 32,32x$
- Mac: $8,5\ minutos\ /\ 2\ minutos\ = 4.25x$

Spark afirma que se puede obtener una mejora de 100x en memoria y de 10x en disco. Los resultados arrojan algo muy distinto, ya que la mejora del Lir no es mayor a 10x. Esto probablemente se deba a que el cluster no está preparado para manejar el tipo de tráfico requerido por estos softwares. Seguramente con la optimización adecuada de red y configuración de ambos softwares, se pueda obtener un número similar al mencionado por Spark.

Las ejecuciones para Wikipedia han dado muchos problemas. En Spark se ha encontrado problemas de Memoria, muriendo la aplicación en medio de la ejecución en todas las pruebas realizadas. Con Hadoop el problema ha sido distinto, ya que el tiempo de ejecución registrado ha sido 11 horas sin haberse recorrido el gráfico entero. Esto último probablemente se deba a que Wikipedia tiene una estructura muy plana con un diámetro mayor a 135 relaciones. Lo cual no sucede en Pokec, donde el

diámetro es de 13 relaciones. Las diferencias en las estructuras de ambos grafos tienen su raíz en la fuente de datos.

Wikipedia es una relación de documentos, la cual es probable que tenga ramas individuales de gran extensión, en cambio Pokec es una relación entre personas de un país determinado, lo cual reduce la cantidad de relaciones que pueden existir entre personas y por ende comprime el conjunto de datos.

Conclusiones

Trabajar y explotar grandes cantidades de datos siempre ha sido un desafío, en la actualidad más que nunca, ya que con la aparición del fenómeno conocido como “Internet de las cosas (IoT)” se requiere, y se va a requerir, formas más eficientes de procesar grandes cantidades de datos. Muchas organizaciones no gubernamentales, gubernamentales y empresas pequeñas tienen actualmente esta necesidad de extraer información de los datos. Para ellos este documento provee un panorama concreto para poder afrontar rápidamente esta necesidad, mostrando las ventajas y desventajas de dos enfoques muy utilizados en el mercado.

En un principio se puede llegar a la conclusión de que es conveniente utilizar Spark en lugar de Hadoop. Ello puede ser oportuno en algunas situaciones pero vale aclarar que Spark no es un reemplazo de Hadoop. Spark potencia a Hadoop, son simbióticos entre sí.

Se puede observar que las ventajas de Spark provee un software con mayor flexibilidad para afrontar el constante cambio, utilizando diversos lenguajes y API’s que proveen una alternativa con gran valor y rápida implementación.

Por otro lado Hadoop existe hace mucho más tiempo que Spark y las organizaciones lo utilizan como una capa más en los sistemas actuales, montando diversos sistemas como Spark, Hbase, Hive, Hue, entre otros para poder abstraerse, facilitar y poder explotar su funcionalidad.

Es posible que programar un trabajo en Hadoop produzca un mayor rendimiento que uno en Spark ya que se puede trabajar específicamente a bajo nivel sobre una necesidad puntual. Puede llegar a existir un caso puntual en el que el rendimiento de un programa de Spark haya sido muy bajo y por ende haya surgido la necesidad de realizar un trabajo específico en Hadoop, obteniendo un rendimiento mucho mayor en cuanto a tiempo de ejecución del programa y recursos computacionales.

Sin embargo Spark provee un contexto más general y simplificado, lo cual tiene un gran valor a la hora de realizar un desarrollo de corto plazo. Como siempre hay que tener en cuenta el tiempo que se posee, el presupuesto, la necesidad y las capacidades que existen en el personal de la organización o aquellas que se está dispuesto a adquirir.

Los resultados obtenidos demuestran que se puede lograr una mejora mayor al 90 % según los datos sobre las ejecuciones utilizando Pokec. Esto prueba que en este caso particular Spark tiene una ventaja considerable sobre Hadoop. Sin embargo es posible que esto no siempre suceda.

Bibliografía y Referencias

[1] Sandy Ryza, Uri Laserson, Sean Owen, Josh Wills, “Advanced Analytics with Spark: Patterns for Learning from Data at Scale”, 2015

[2] Jimmy Lyn, Chirs Dyler “Data-Intensive Text Processing with Map Reduce”, 2011

[3] Jeffrey Dean, Sanjay Ghemawat MapReduce: “Simplified Data Processing on Large Clusters Google”, Inc, 2004. http://research.google.com/archive/mapreduce.html

[4] https://hadoop.apache.org

[5] http://spark.apache.org

[6] https://github.com/gastonestol/distributedDijkstra

[7] https://es.wikipedia.org/wiki/PageRank

[8] https://es.wikipedia.org/wiki/MapReduce

[9]http://www.personal.kent.edu/%7Ermuhamma/Algorithms/MyAlgorithms/GraphAlgor/dijkstraAlgor.htm

[10] http://people.seas.harvard.edu/~babis/tkdd10.pdf

[11] https://aws.amazon.com/es/elasticmapreduce/

[12] https://github.com/serpi90/hadoop-installer

[13] https://es.wikipedia.org/wiki/Big_data

[14] Adaptado de las diapositivas de UMD Jimmy Lin’s, las cuales están licenciadas bajo Creative Commons Attribution - Sin fines comerciales- Share Alike 3.0 United States. Ver http://creativecommons.org/licenses/by-nc-sa /3.0/us/ para mas detalles

[15] http://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters

[16] http://www.dcs.bbk.ac.uk/~dell/teaching/cc/paper/sigmod10/p135-malewicz.pdf

Anexo A: Fragmentos de Código Relevante

1. Driver

```
@Override
public void ssspJob( String inputPath, String outputPath,String sourceNode)
throws Exception {

    int iterationCount = 0;
    long terminationValue =1;
    Date start = new Date();
    long totalTime = 0;
    Job job;

    while(terminationValue >0){

        Date iterStart = new Date();
        job = getJobSsspConfiguration();
        job.getConfiguration().set("source", sourceNode.trim());
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        String input, output;

        if (iterationCount == 0){
                FileInputFormat.setInputPaths(job, new Path(inputPath));
                FileOutputFormat.setOutputPath(job, new
Path(outputPath+"_"+iterationCount));

        }else{
```

```
                FileInputFormat.setInputPaths(job, new
Path(outputPath+"_"+(iterationCount-1)));

                FileOutputFormat.setOutputPath(job, new
Path(outputPath+"_"+iterationCount));



        }



        job.waitForCompletion(true);

        Date iterEnd = new Date();



        System.out.println("Job number " + iterationCount + " length in ms: " +
(iterEnd.getTime() - iterStart.getTime()));



        Counters jobCntrs = job.getCounters();

        terminationValue =
jobCntrs.findCounter(MoreIterations.numberOfIterations).getValue();







        iterationCount++;

        }

    Date end = new Date();



    System.out.println("Complete Job length in ms:
"+(end.getTime()-start.getTime()));



}
```

2. Mapper

```
public class SearchMapper extends Mapper<Object, Text, Text, Text> {



    public void map(Object key, Text value, Context context, Node inNode)

                throws IOException, InterruptedException {
```

```
        if (inNode.getColor() == Node.Color.GRAY) {
                for (String neighbor : inNode.getEdges()) {
                        Node adjacentNode = new Node();
                        adjacentNode.setId(neighbor);
                        adjacentNode.setDistance(inNode.getDistance() + 1);
                        adjacentNode.setColor(Node.Color.GRAY);
                        adjacentNode.setParent(inNode.getId());
                        context.write(new Text(adjacentNode.getId()), new
Text(adjacentNode.getNodeInfo()));
                }
                inNode.setColor(Node.Color.BLACK);
        }

        context.write(new Text(inNode.getId()), new Text (inNode.getNodeInfo()));

    }
}
```

3. Reducer

```
public class SearchReducer extends Reducer<Text, Text, Text, Text> {

    public Node reduce(Text key, Iterable<Text> values, Context context, Node
outNode)
        throws IOException, InterruptedException {

        outNode.setId(key.toString());
        for (Text value : values) {
```

```
                        Node inNode = new Node(key.toString() + value.toString());
                        if (inNode.getEdges().size() > 0) {
                                outNode.setEdges(inNode.getEdges());
                        }
                        if (inNode.getDistance() < outNode.getDistance()) {
                                outNode.setDistance(inNode.getDistance());
                                outNode.setParent(inNode.getParent());
                        }
                        if (inNode.getColor().ordinal() > outNode.getColor().ordinal()) {
                                outNode.setColor(inNode.getColor());
                        }

                }
                context.write(key, new Text(outNode.getNodeInfo()));
                return outNode;

            }
        }
```

4. Aplicación de Spark

```
import org.apache.spark.graphx._
import org.apache.spark.{SparkConf, SparkContext}
object SimpleSSSP {
    def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val graph = GraphLoader.edgeListFile(sc, args(0), false, 1).mapEdges(e => e.attr.toDouble)
    val sourceId: VertexId =   args(1).toLong
    val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)
    initialGraph.cache()
```

```
    val sssp = initialGraph.pregel(Double.PositiveInfinity)(
     (id, dist, newDist) => math.min(dist, newDist),
     triplet => {


            if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
                    Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
            } else {
                    Iterator.empty
            }
     },
     (a, b) => math.min(a, b)
    )
    println(sssp.vertices.collect.mkString("\n"))
    }
}
```