Tarea 1: Hadoop

Paula Navarrete, Sebastián Orellana

I. Interacción con Hadoop Distributed File System

En esta actividad se hace uso del sistema de archivos de Hadoop, HDFS. En particular, es de interés la medición del tiempo de escritura y lectura de archivos sobre el HDFS bajo distintas circunstancia. Para ello, es preciso interactuar con distintas funcionalidades previas del HDFS tales como: la creación de un directorio (comando -mkdir), copiar archivos desde el sistema local (comando -put), mostrar contenido de archivos (comando -cat), verificar información sobre el uso del disco por parte de HDFS (comando -dus), entre otros.

A. Tiempos de transferencia v/s tamaño archivo

Primeramente se escriben y leen archivos de distinto tamaño sobre el HDFS, midiendo el tiempo de transferencia (recuperado desde el archivo *"/var/log/hadoop-hdfs/hadoop-hdfs-namenode-localhost.localdomain.log"*). La Figura 1 presenta los tiempos de escritura al variar el tamaño de archivo. Notar que se consideran archivos de tamaño 10 MB - 140 MB, con diferencias equiespaciadas entre sí de 5 MB. Además, se utiliza el tamaño de bloque por defecto: 64 MB. A grandes rasgos, se observa un crecimiento lineal del tiempo de transferencia, lo cual es razonable dado el carácter pseudo-distribuido de la instalación de Hadoop, donde todos los *daemons* se ejecutan en la misma máquina, implicando que los archivos no pueden ser distribuídos sobre diversos discos que mitiguen la secuencialidad del proceso de escritura. Además, dado que el factor de replicación se mantiene fijo en 1, tampoco se verifica su efecto en la velocidad de transferencia de un proceso efectivamente distribuído. Un aspecto interesante se produce cuando el tamaño de archivo supera el tamaño del bloque establecido, donde se evidencia una leve disminución en el tiempo de transferencia. Esto se interpreta como un primer beneficio de una instalación pseudo-distribuida, donde el balanceo de carga monitoreado por en NameNode exige la partición del archivo en diversos DataNodes, permitiendo fragmentar la transmisión que se traduce en una disminución en el tiempo de la tarea.

La Figura 2 presenta los tiempos de lectura al variar el tamaño de archivo. Notar que se consideran archivos de tamaño 10 MB - 140 MB, con diferencias equiespaciadas entre sí de 10 MB. Nuevamente se utiliza 64 Mb como tamaño de bloque. En términos generales, se sigue un comportamiento similar al observado para la escritura, manteniendo la linealidad en la tarea y los leves decaímientos en tiempos de transferencia al superar el tamaño del bloque, evidenciando que la fragmentación de también constribuye a

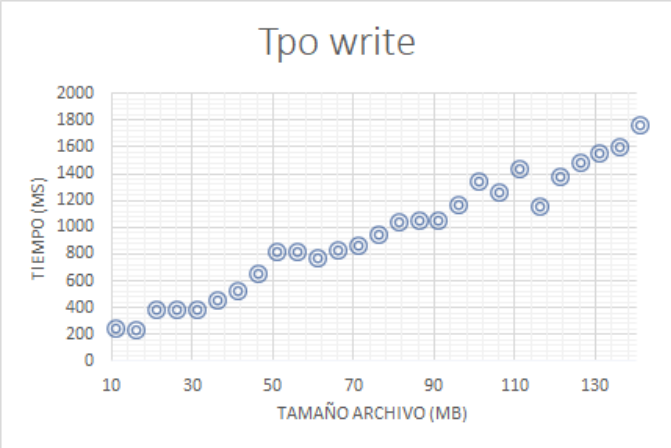

Fig. 1. Tiempos de escritura.

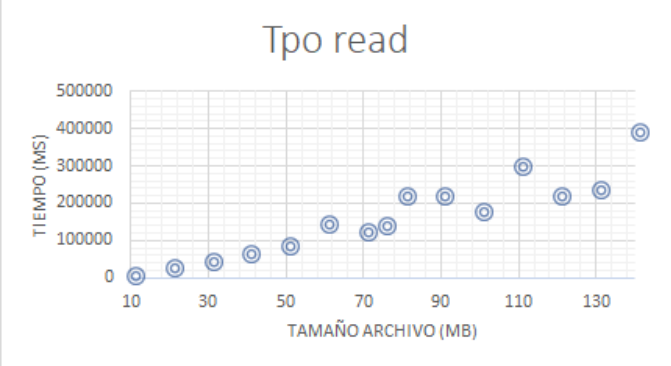

Fig. 2. Tiempos de lectura.

la eficiencia con que se recuperan los datos del HDFS. Cabe destacar que los tiempos de lectura son considerablemente superiores a los de escritura, probablemente influído por la mayor carga asignada al NameNode para ubicar la dirección en los NameNodes de los archivos solicitados.

B. Modificación del tamaño del bloque

A continuación, se repite el ejercicio anterior, pero esta vez utilizando dos tamaños de bloque distinto: 32 MB y 128 MB. La Figuras 3 y 4 presentan los tiempos de escritura y lectura al variar el tamaño de archivo respectivamente, utilizando tamaño de bloque 32 MB. Se observa que los tiempos aumentan levemente con respecto al caso de tamaño de bloque 64 MB, pero el comportamiento se mantiene similar. Notar además que la mayor fragmentación implica mayor inestabilidad en los tiempos de la transferencia, lo cual se evidencia principalmente en la Figura 4. Se decide no incluir

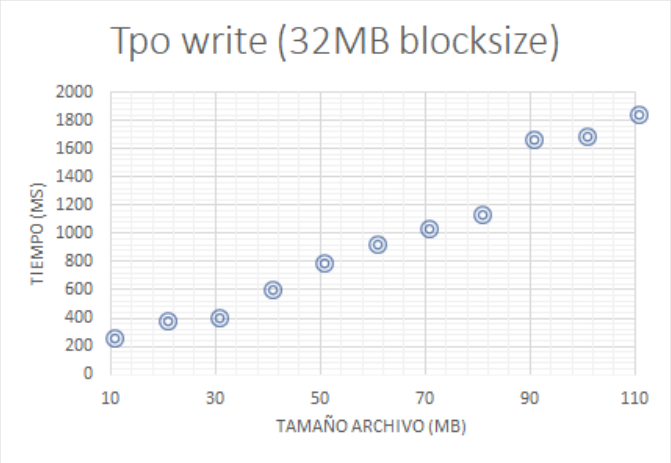

Fig. 3. Tiempos de escritura, tamaño bloque 32 MB.

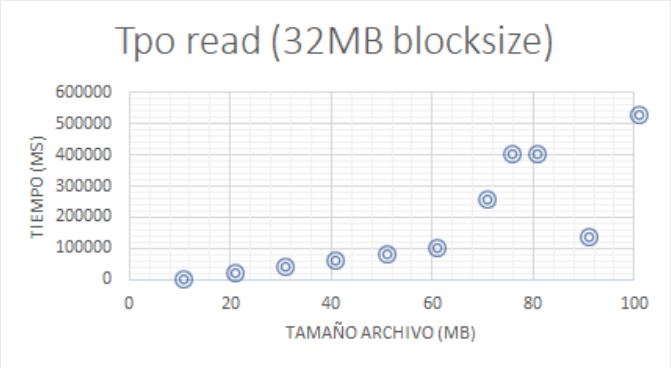

Fig. 4. Tiempos de lectura, tamaño bloque 32 MB.

los gráficos de tiempo de transferencia de lectur y escritura con tamaño de bloque 128 MB pues el comportamiento es similar.

II. Cuenta de Palabras

A continuación, se emplea el paradigma de programación Map Reduce para contar la ocurrencia de cada una de las palabras que aparecen en un conjunto de documentos (*word count*). La solución es implementada en Python, valiéndose para ello de la aplicación HadoopStreaming, que permite ejecutar funciones de Map y Reduce desde la *shell* de la máquina virtual. El código para la función Map es el siguiente:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    # Tratamiento de palabras desde entrada
    estandar
    line = line.strip()
    words = line.split()
    for word in words:
        # Escritura de los elementos (clave,valor)
    en salida estandar
        print '%s\t%s' % (word, 1)
```

wordcount–mapper.py

Y el código para la función Reduce:

```
#!/usr/bin/env python
from operator import itemgetter
import sys

cur_word = None
cur_count = 0
word = None

for line in sys.stdin:
    # Tratamiento de palabras desde entrada
    estandar
    line = line.strip()

    # Parseo de elementos (clave, valor) agrupados
    por clave
    word, count = line.split('\t', 1)

    # Convierte contador a entero
    try:
        count = int(count)
    except ValueError:
        continue

    # Suma de ocurrencias
    if cur_word == word:
        cur_count += count
    else:
        if cur_word:
            # Escritura del resultado en salida
    estandar
            print '%s\t%s' % (cur_word, cur_count)
        cur_count = count
        cur_word = word

# Escritura del ultimo valor en salida estandar
if cur_word == word:
    print '%s\t%s' % (cur_word, cur_count)
```

wordcount–reducer.py

En particular, el conteo se realiza sobre el archivo *palabras.txt*, guardando la salida en el archivo *wordcount-output.txt* adjunto en la entrega. Notar que el delimitador entre palabras es exclusivamente el carácter espacio en blanco, lo cual implica que los pares de palabras del tipo *(“virtual”,“virtual.”)* ó *(“veces”,“veces,”)* sean contabilizadas con frecuencias independientes. Se decide no realizar ningún tratamiento adicional dadas las características del documento utilizado (texto extraído de una plantilla LaTeX), que consta de palabras del tipo *“\texttttA.key=B.Key\\”*, que incluyen signos de puntuación y otros caracteres atípicos que de ser tratados podrían herir la semántica del texto. Situación similar se produce con el manejo de mayúsculas y minúsculas, donde a menudo diferencias en este aspecto se traducen en una amplia disimilitud en el espacio semántico subyacente. Así, de ser ignorado este comporatamiento, pares de palabras que incluyan siglas como *(“siga”,“SIGA”)* ó *(“un”,“UN”)* serían equivalentes, siendo que evidentemente representan distintos conceptos. No obstante, es factible realizar un tratamiento más agudo de este fenómeno que permita robustecer el conteo de palabras sin herir la semántica de los documentos, sin embargo se escapa del ámbito de esta actividad.

III. Unión de datos: Parte 1

En esta sección se implementa una aplicación típica de SQL: join. En particular se utiliza Hadoop para unir el contenido de los archivos *dataCuentaTotal.txt* y *dataCuentaDiaria.txt*. El archivo *dataCuentaTotal.txt* contiene el resultado de la cuenta total de cierto grupo de palabras, empleando el formato <*palabra,cuentaTotal*>. El archivos *dataCuentaDiaria.txt* contiene el resultado de la cuenta diaria de palabras según fecha, utilizando el formato <*fecha palabra,cuentaDiaria*>. El objetivo es unir (join) estos archivos, de tal manera que se genere un archivo de salida con la cuenta diaria y total de cada palabra, utilizando el formato <*fecha palabra,cuentaDiaria cuentaTotal*>.

Como una aplicación de este tipo podría servir para identificar fechas en las que determinadas palabras aparecen más de lo usual, la salida se ordena por defecto en función de las palabras. Luego, como también podría servir para identicar palabras que en determinadas fechas aparecen más de lo usual, la salida se puede ordenar (alfabéticamente por medio del flag *-D mapred.text.key.comparator.options = '-k1,1*) en función del campo fecha. Ambos archivos de salida se adjuntan a la entrega, con nombres *join-output.txt* y *join-output-fecha.txt* respectivamente. El código en Python para la función Map es el siguiente:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    # Setea valores por defecto
  fecha = "-"
  palabra = ""
  cuentaDiaria = "-"
  cuentaTotal = "-"

  # Tratamiento de palabras desde entrada estandar
  line = line.strip()
  splits = line.split(" ")

  # Identifica cuentaDiaria, pues tiene mas
    columnas que cuentaTotal
  if len(splits) == 3:
    # Parsea registros
    fecha = splits[0]
    palabra = splits[1]
    cuentaDiaria = splits[2]
  # Identifica cuentaTotal
  else:
    # Parsea registros
    palabra = splits[0]
    cuentaTotal = splits[1]

  # Escritura de los elementos (clave,valor) en
    salida estandar
  print palabra + ' ' + fecha + ' ' + cuentaDiaria
    + ' ' + cuentaTotal
```

join–mapper.py

Notar que una elección apropiada de *key* por parte del mapper corresponde al campo en común de ambos archivos: *palabra* (sobre el cual se quiere combinar los datos). Luego, (*fecha*, *cuentaDiaria*, *cuentaTotal*) corresponde al *value* de cada palabra. El código Python para la función Reduce es:

```
#!/usr/bin/env python
import sys
import string

last_palabra = None
cur_cuentaTotal = "-"

for line in sys.stdin:
    # Tratamiento de palabras desde entrada
    estandar
    line = line.strip()

    # Parseo de elementos (clave, valor) agrupados
    por clave
    palabra,fecha,cuentaDiaria,cuentaTotal = line.
    split(" ")

    # Identifica si es una nueva palabra
    if not last_palabra or last_palabra != palabra:
        last_palabra = palabra
        cur_cuentaTotal = cuentaTotal
    elif palabra == last_palabra:
        cuentaTotal = cur_cuentaTotal
        # Escritura del resultado en salida
    estandar
        print fecha + ' ' + palabra + ' ' +
    cuentaDiaria + ' ' + cuentaTotal
```

join–reducer.py

Así, teniendo en cuenta que el mapper deja los datos ordenados por *key*, al reducer le basta con recorrerlos secuencialmente para identificar el último par (*key,value*) por palabra, asignar la *cuentaTotal* respectiva e imprimir la salida.

IV. Unión de datos: Parte 2

Por último, se repite la unión de documentos efectuada en la sección anterior, pero en este caso sobre dos tipos de archivos: *i)* Información de programas de televisión que son emitidos por distintos canales, e *ii)* Información del número de televidentes que observan distintas emisiones de cada programa. El objetivo es implementar funciones de Map y Reduce que permitan calcular el total de televidentes que ve cada uno de los programas emitidos por cierto canal de televisión. El código en Python para la función Map es el siguiente:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    # Setea valores por defecto
  programa = ""
  audiencia = "0"
  canal = False

  # Tratamiento de palabras desde entrada estandar
  line = line.strip()
  splits = line.split(",")

  # Se reconoce si segundo campo es un digito (
    programa,audiencia)
  if splits[1].isdigit():
    programa = splits[0]
    audiencia = splits[1]
    canal = True
  # Se reconoce (programa,canal)
  else:
```

```
    programa = splits[0]
    canal = splits[1]=='CAB'
  # Chequea si canal = CAB
  if canal:
    # Escritura de los elementos (clave,valor) en
    salida estandar
    print programa + ' ' + audiencia
```

join2–mapper.py

Nuevamente la elección apropiada de *key* por parte del mapper corresponde al campo en común de ambos archivos: *programa*. Notar que el mapper maneja la posibilidad de que cada combinación (*TvShow, channel*) pueda aparecer múltiples veces, seteando por defecto el campo *audiencia* en cero, por lo que repeticiones de registros no interfieren en la suma de total de audiencia. El código Python para la función Reduce es:

```
#!/usr/bin/env python
import sys
import string

last_programa = None
audiencia_total = 0

for line in sys.stdin:
    # Tratamiento de palabras desde entrada
    estandar
    line = line.strip()

    # Parseo de elementos (clave, valor) agrupados
    por clave
    programa,audiencia = line.split(" ")
    audiencia = int(audiencia)

    # Identifica si es un programa antiguo
    if not last_programa or last_programa ==
    programa:
        # Suma cantidad de audiencia
        audiencia_total += audiencia
        last_programa = programa
    # Identifica si es un programa nuevo
    elif programa != last_programa:
        # Escritura del resultado en salida
    estandar
        print last_programa + ' ' + str(
    audiencia_total)
        last_programa = programa
        audiencia_total = audiencia

# Escritura del ultimo programa
print last_programa + ' ' + str(audiencia_total)
```

join2–reducer.py

Así, teniendo en cuenta que el mapper deja los datos ordenados por *key*, al reducer le basta con recorrerlos secuencialmente para identificar el último par (*key,value*) por programa, y asignar la suma total de audiencia asociada a las múltiples combinaciones (*TvShow, audiencia*).

Notar además, que para este caso particular el reporte obtenido corresponde a la cuenta de los televidentes que vió programas de CAB (archivo de salida *join2-output.txt* adjunto en la entrega), lo cual es filtrado directamente en el mapper con el fin de manejar circunstancias en que un mismo programa de televisión es emitido por varios canales. Para cambiar el canal bajo análisis basta con modificar el valor que se asigna a la variable *canal* en la línea 21 del código del mapper.

References

[1] https://www.cloudera.com/developers/get-started-with-hadoop-tutorial.html

[2] https://wiki.apache.org/hadoop/HadoopStreaming