Análisis de sentimientos con Hadoop

Sistemas Distribuidos de Procesamiento de Datos

Miguel Ángel Monjas Llorente

Descripción del código

Advertencia: las descripciones de la memoria se refieren, a menos que se especifique lo contrario, a las implementaciones basadas en Map Reduce Streaming.

La funcionalidad principal implementada es la referida a la búsqueda del "sentimiento" medio, por unidad geográfica, en Twitter. Para ello, se han elegido los estados de Estados Unidos como dicha unidad geográfica, por varias razones: la abundancia de *tweets* provenientes de dicho país y la relativa facilidad para encontrar cuál es el estado de procedencia a partir de la información de localización del *tweet* (no se hace consulta en línea de coordenadas, ya que se considera que no aporta demasiado al propósito de la práctica). También se ha abordado la, comparativamente más sencilla, detección de los diez *trending topics* en los tweets del periodo abordado. Se ha descargado a través del API de Twitter un dataset de unos 8 Gbytes (en tres ficheros). Se proporcionan varios *scripts* Python y un fichero diccionario de estados de Estados Unidos.

Se describirá a continuación el código de la solución principal. Se ha utilizado el paradigma MapReduce Streaming implementado con Python 2.7. Todo el material generado puede encontrarse dentro del directorio `MapReduce` del siguiente repositorio: https://github.com/miguel-angel-monjas/datascience-SDPS. Existen variantes que se describirán en la siguiente sección.

La **variante principal** consta de dos *scripts*, `mapper_ng.py` y `reducer_ng.py`, acompañados de dos ficheros de texto, `us_states.txt` (que ayuda a nombrar y localizar cada estado de Estados Unidos de forma unificada) y `AFINN-111.txt` (permite evaluar los sentimientos asociados a unas dos mil palabras).

`mapper_ng.py` implementa de forma simultánea dos funciones de mapeo: (a) una asociada al identificador del *tweet* como clave, que permitirá encontrar el sentimiento asociado a cada unidad territorial; y (b) otra, con el *hashtag* como clave, para encontrar los *trending topics* (otra posibilidad podría ser encadenar dos funciones de mapeo, pero se prefiere el mapeo simultáneo dado que en cualquiera de las dos hay que recorrer todas las palabras del texto del *tweet*). La funcionalidad de mapeo se efectúa siguiendo los siguientes pasos:

1. Los *tweets* se leen de uno en uno, y se cargan con la función `loads` del módulo `json de Python`.
2. Para cada uno de ellos se intenta extraer el valor asociado a las claves `text` (texto del *tweet*), `lang` (idioma), `id_str` (identificador del *tweet*) y `place` (que proporciona la información de localización; se prefiere esa a la contenida en la clave `location`). La existencia de estas claves se controla mediante excepciones. Si alguna de las claves falta, se omite el *tweet* del análisis y se pasa al siguiente.
3. Solo se consideran *tweets* en inglés (idioma: `en`) provenientes de Estados Unidos (clave asociada al valor `country_code` en `place`: `US`), prescindiéndose del resto.
4. Para cada *tweet* se determina cuál es el estado del que proviene, utilizando un fichero de texto compuesto al efecto: `us_states.txt`. La información del estado del que proviene el tweet se extrae del valor asociado a la clave `full_name` del objeto `place`, en el que suele aparecer la abreviatura del estado asociado al *tweet*.
5. Se extraen las palabras contenidas en el texto del *tweet*. Solo se consideran palabras compuestas de caracteres alfanuméricos o de la almohadilla (#).
6. Para todas aquellas palabras identificadas como *hashtags* (comienzan con #), se emite, como salida del *mapper*, un par clave-valor con el *hashtag* (completo) como clave y un uno (`1`) como valor (el valor es realmente irrelevante). Esta función de mapeo es similar a la del omnipresente wordcount. Esta es la primera función de mapeo.
   Por otra parte, para todas las palabras (incluyendo los *hashtags*, a los que se les elimina el carácter almohadilla), se emite como salida del *mapper* un par clave-valor en el que la clave es el identificador

del *tweet* y como valor se incluye una tupla (entidad geográfica, palabra). Esta es la segunda función de mapeo. Puede verse un ejemplo de la salida de la ejecución de `mapper_ng` a continuación (aparecen mezcladas las salidas de ambos *mappers*):

```
9697190427166720 0      ['Oregon', u'all']
796971904271667200      ['Oregon', u'trump']
796971904271667200      ['Oregon', u'supporters']
796971904271667200      ['Oregon', u'in']
796971904271667200      ['Oregon', u'w']
796971904271667200      ['Oregon', u'the']
796971904271667200      ['Oregon', u'clan']
#StopTheViolence        1
796971904271667200      ['Oregon', u'stoptheviolence']
796971912664469504      ['South Dakota', u'everything']
796971912664469504      ['South Dakota', u'about']
796971912664469504      ['South Dakota', u'you']
796971912664469504      ['South Dakota', u'is']
796971912664469504      ['South Dakota', u'so']
796971912664469504      ['South Dakota', u'overwhelming']
```

Las dos funciones de reducción se implementan mediante el *script* `reducer_ng.py`. En este caso, qué función se aplica se controla mediante un parámetro pasado al script, `type`. Con el valor `states` se implementa la función de reducción asociada al cálculo de los sentimientos. El valor `hashtags` implementa la función de reducción asociada a los *trending topics*. La detección de a qué *mapper* corresponde cada línea se hace examinando el primer carácter de la clave. La salida del *script* tiene formato CSV, con un punto y coma como separador. En el caso de los sentimientos, es una tripleta `estado;número_de_tweets;sentimiento_medio`. En el caso de los *hashtags*, se sigue también un formato CSV del tipo `hashtag;número_de_apariciones`. Solo se muestran los diez primeros.

En el caso de los *hashtags*, la función de reducción simplemente va almacenando los *hashtags* y su número de apariciones. En el caso de los sentimientos, se calcula el sentimiento total asociado a cada *tweet* (identificado por su clave). A continuación, se va almacenando este dato en un diccionario cuyas claves son los nombres de los estados. El valor asociado a cada clave de este diccionario es una lista con el número de tweets de cada estado y el sentimiento agregado.

Adicionalmente, con el objeto de extraer indicaciones de tiempo para las ejecuciones locales, se ha previsto la escritura del tiempo de comienzo y de final de *mapper* y *reducer* imprimiendo a `stderr`.

Para la implementación en AWS EMR, se ha requerido un paso *map-reduce* adicional. La razón de esto ha sido la forma en la que AWS EMR implementa la función de reducción. En particular, genera un fichero de salida por cada *reducer*. Para ello, ha sido necesario implementar un *reducer* adicional, que se ha aplicado sobre la salida del *reducer* original. Esta función toma los contenidos de los ficheros generados por el paso anterior y calcula el sentimiento asociado a cada estado (cada fichero incluye sumas parciales para un mismo estado). La función de mapeo asociada a este último paso ha sido el *identity mapper*, que he implementado mediante el comando Linux `/bin/cat`. La implementación en AWS EMR ha requerido, por tanto, un nuevo *reducer* y algunas modificaciones en el *mapper* y el *reducer* originales (por un lado, para poder acceder a los ficheros adicionales, véase la última sección, Comentarios personales; por otro, la salida del primer *reducer* ha tenido que ser adecuada al formato requerido para ejecutar MR Streaming). Se trata, en este caso de tres *scripts*: `mapper_aws.py`, `reducer_aws.py` y `reducer_aws_02.py`.

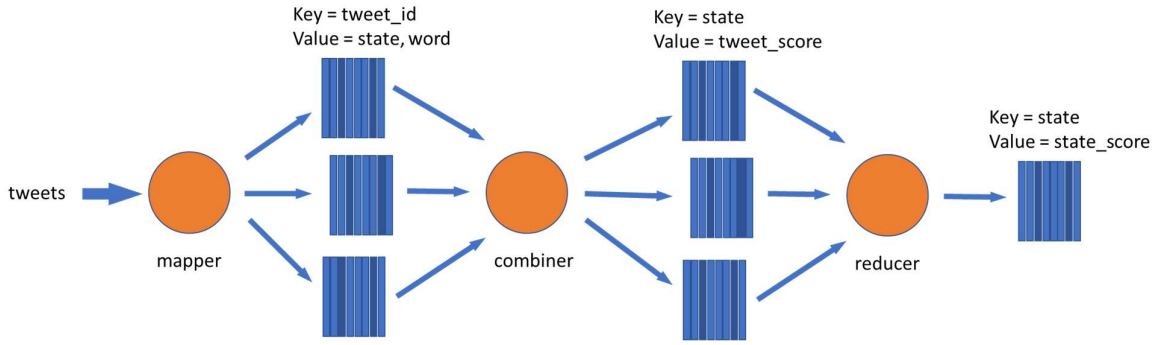
Implementación basada en MRJob

Tal como se menciona en la sección final, la implementación basada en MRJob está menos desarrollada que la basada en Map Reduce Streaming. Se ha abordado únicamente la búsqueda del sentimiento medio por unidad territorial. El código requerido en MRJob es notablemente más limpio y ha utilizado un *mapper*, un *combiner* y un *reducer*, siguiendo la filosofía descrita en la sección anterior:

1. El *mapper* lee los ficheros con tweets línea a línea. Cada línea es un tweet y su contenido se carga con la función `loads` del módulo `json de Python`.
2. Para cada uno de ellos se intenta extraer el valor asociado a las claves `text`, `lang`, `id_str` y `place`. Si alguna de las claves falta, se omite el *tweet* del análisis y se pasa al siguiente. Solo se consideran *tweets* en inglés provenientes de Estados Unidos.
3. Para cada *tweet* se determina cuál es el estado del que proviene.

4. Se extrae cada una de las palabras contenidas en el texto del *tweet*. Solo se consideran palabras compuestas de caracteres alfanuméricos o de la almohadilla (#).
5. El *mapper* emite como salida un par clave-valor en el que la clave es el identificador del *tweet* y el valor es una tupla de la forma (entidad geográfica, palabra).
6. El *composer* toma la entrada del *mapper* y calcula, el sentimiento asociado a cada tweet. Su salida es un par clave valor con el estado como clave y el sentimiento asociado a cada *tweet* por otro.
7. Finalmente, el *reducer* toma la entrada del *composer* y, para cada valor posible de la clave (el estado) cuenta el número de tweets y calcula el sentimiento medio de los *tweets* asociados a ese estado.

La secuencia puede ilustrarse mediante la siguiente figura:



La ejecución basada en MRJob se basa en un único script: `mrjob_twitter.py`.

Diferentes formas de ejecución realizadas

La ejecución se ha hecho de tres formas utilizando MapReduce Streaming: localmente, en Hortonworks Sandbox y en AWS EMR.

A continuación, se muestran dos ejemplos de la ejecución en Hortonworks Sandbox. El primero se refiere a la funcionalidad de extracción de sentimientos por unidad geográfica. El segundo a la determinación de los diez *trending topics*:

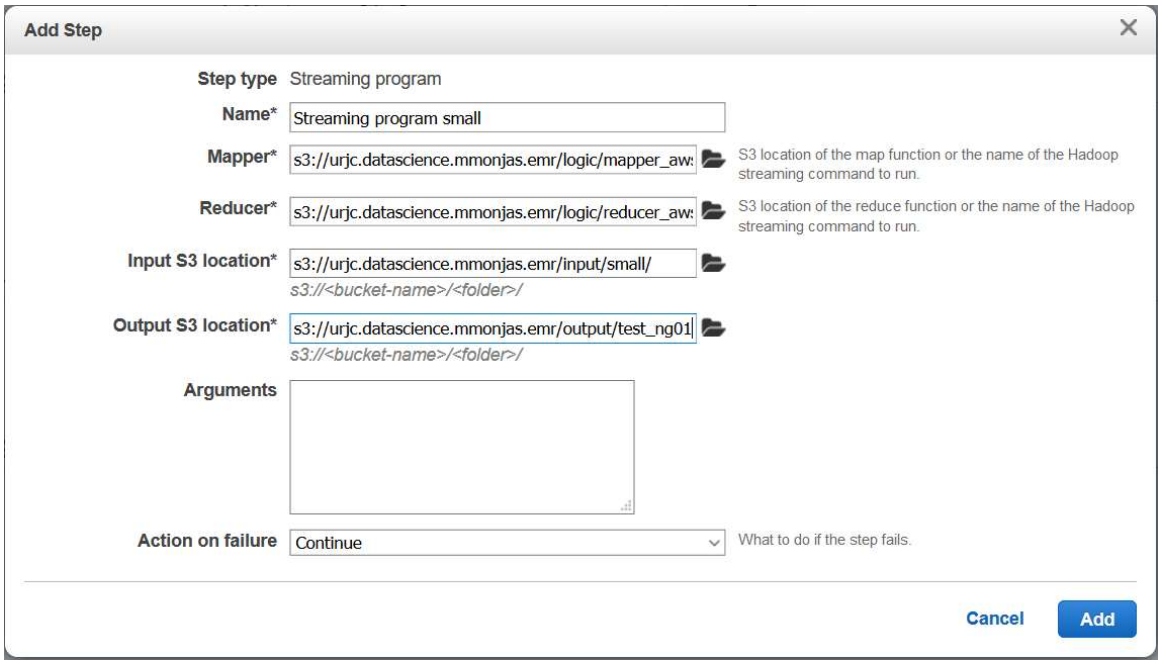
```
[root@sandbox ~]# hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
-files mapper_ng.py,reducer_ng.py,AFINN-111.txt,us_states.txt
-mapper mapper_ng.py -reducer 'reducer_ng.py --type states'
-input hdfs://sandbox.hortonworks.com/user/root/input/streaming/dump2.txt
-output hdfs://sandbox.hortonworks.com/user/root/output/streaming_01

[root@sandbox ~]# hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
-files mapper_ng.py,reducer_ng.py,AFINN-111.txt,us_states.txt
-mapper mapper_ng.py -reducer 'reducer_ng.py --type hashtags'
-input hdfs://sandbox.hortonworks.com/user/root/input/streaming/dump2.txt
-output hdfs://sandbox.hortonworks.com/user/root/output/streaming_02
```

La obtención de los resultados puede hacerse utilizando los comandos habituales de HDFS:

```
[root@sandbox ~]# hdfs dfs -getmerge
hdfs://sandbox.hortonworks.com/user/root/output/streaming_01 ./results_01.txt
```

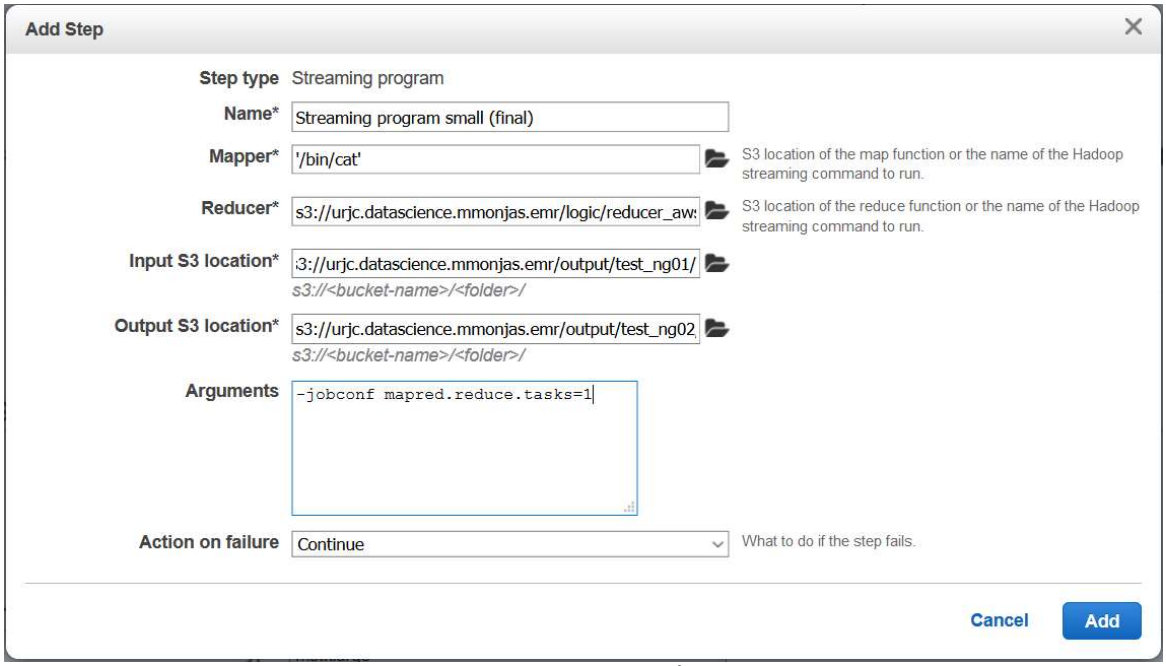
La ejecución en AWS EMR es similar. Se muestran a continuación las pantallas de configuración de los pasos y los comandos generados a través de la consola (con todos los elementos almacenados en S3). El primer paso es como sigue:



El comando generado es el siguiente:

```
hadoop-streaming -files
s3://urjc.datascience.mmonjas.emr/logic/mapper_aws.py,s3://urjc.datascience.mmonjas.emr
/logic/reducer_aws.py
-mapper mapper_aws.py
-reducer reducer_aws.py
-input s3://urjc.datascience.mmonjas.emr/input/small/
-output s3://urjc.datascience.mmonjas.emr/output/test_ng01/
```

El segundo paso es como se muestra a continuación:



Debe observarse que en el segundo paso se ha incluido explícitamente una orden para que solo haya un *reducer* (`-jobconf mapred.reduce.tasks=1`) y que, por tanto, solo se genere un fichero de salida. El comando generado es el siguiente:

```
hadoop-streaming -files
s3://urjc.datascience.mmonjas.emr/logic/mapper_aws.py,s3://urjc.datascience.mmonjas.emr
/logic/reducer_aws.py
-mapper mapper_aws.py
-reducer reducer_aws.py
-input s3://urjc.datascience.mmonjas.emr/output/test_ng01/
-output s3://urjc.datascience.mmonjas.emr/output/test_ng02/
```

Ejecución con MRJob

Se muestran dos ejecuciones de la versión basada en MRJob. La primera es la ejecución local. La segunda la ejecución contra AWS EMR.

```
$ cat dump4.txt | python mrjob_twitter.py --runner inline --file us_states.txt --file
AFINN-111.txt > rs04mrjobs.txt
```

La ejecución contra AWS EMR se ha ejecutado de la siguiente forma:

```
$ python mrjob_twitter.py dump.txt -r emr
s3://urjc.datascience.mmonjas.emr/input/large/
--file AFINN-111.txt --file us_states.txt
--ec2-instance-type c1.medium
--num-core-instances 3
--output-dir=s3://urjc.datascience.mmonjas.emr/output/test_m11
```

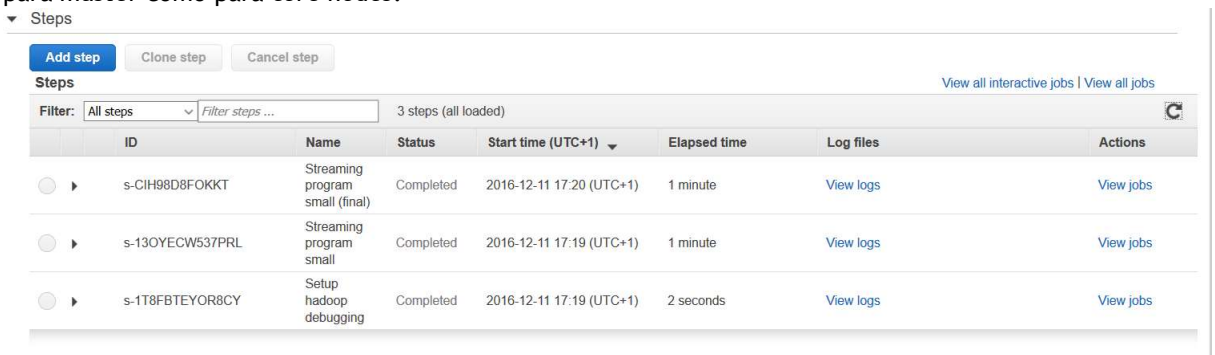
Evaluación de la escalabilidad

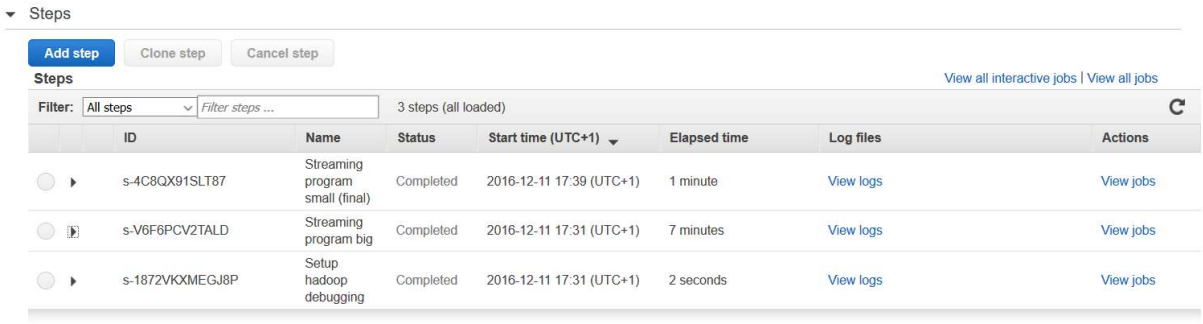
La evaluación de la escalabilidad se discutirá en dos vertientes. En primer lugar, abordando la escalabilidad intrínseca de las funciones de *map* y de *reduce*. Por otra parte, haciendo diversas pruebas, en AWS EMR con distintos tamaños de la entrada y distintas configuraciones del clúster.

En relación con el primer aspecto, la escalabilidad intrínseca de las funciones, en una primera versión, el esquema de *map-reduce* ejecutaba gran parte de las tareas en el *mapper*, concretamente las relativas al cálculo de la felicidad por *tweet*. Aunque este enfoque es abordable considerando que los *tweets* están limitados a 140 caracteres (esto es, no habrá nunca más de unas pocas decenas de palabras), no sería el adecuado en escenarios en los que, en vez de *tweets*, se estuviesen considerando textos de longitud mayor. Por ello, se ha elegido el enfoque descrito en la primera sección, en el que es la función de reduce la que computa el sentimiento asociado a cada *tweet*.

En relación al segundo aspecto, hemos realizado dos comparaciones: por tamaño de entrada y por configuración del clúster. Hemos utilizado los tiempos de ejecución en local como indicador. Para ello, hemos ejecutado las tareas de map reduce sobre una entrada de 150 MBytes y sobre otra de aproximadamente 4 GBytes. En el primer caso, el tiempo de ejecución estaba en el entorno de los diez segundos. En el segundo caso, se encontraba alrededor de 4 minutos y medio.

En el primer caso hemos comparado la ya citada entrada de 150 MBytes frente a una de aproximadamente 8 GBytes. A continuación, se muestran los tiempos de ejecución de ambas ejecuciones (la primera es la efectuada sobre la entrada de menor tamaño; la segunda, la ejecutada sobre la entrada de mayor tamaño). En ambos casos, se usa una configuración pequeña e idéntica para ambos casos, **m3.xlarge** tanto para *master* como para *core nodes*:





Puede observarse que no hay una relación lineal entre las ejecuciones. Prescindiendo de los pasos identificados en la figura como *(final)*, que corresponden al proceso de reduce necesario para consolidar los ficheros producidos por el paso inicial, en el caso del fichero de 150 MBytes el tiempo tardado ha sido un minuto. Un conjunto de ficheros cincuenta veces mayor solo ha requerido siete veces más tiempo. Se presume (a falta de hacer pruebas con *datasets* del orden de TBytes) que existe un umbral mínimo de tiempo de ejecución impuesto por la mecánica Map Reduce. Una vez sobrepasado este umbral, se presume que existe una correlación entre el tamaño de la entrada y el tiempo de ejecución. En comparación con la ejecución local, se puede observar que los tiempos de ejecución para el caso de 8 Gbytes en AWS EMR pueden estimarse menores (si bien no significativamente) comparado con el caso local (suponiendo una relación lineal, para un fichero de 8 GBytes podríamos hablar de unos nueve minutos para la ejecución local).

Como comparación, se puede indicar también los tiempos de ejecución de la versión basada en MRJob, con los parámetros indicados en la sección anterior y sin efectuar la segunda tarea de *reduce*. En el caso del dataset de 150 MBytes, el tiempo consumido ha sido de casi dos minutos (`RUNNING for 113.3s`). La ejecución en el caso del dataset de 8 GBytes ha tardado más de 20 minutos.

A continuación, mostramos los tiempos de ejecución del paso fundamental de Map Reduce en varias configuraciones. Las dos primeras son configuraciones de propósito general; las tres siguientes optimizadas para cómputo y, finalmente, las tres últimas están optimizadas para memoria):[1]

Configuración	CPU	Memoria (GBytes)	Tiempo de ejecución del paso principal	Tiempo total	Horas de instancias normalizadas
m3.xlarge	4	15	7 minutos	15 minutos	24
m3.2xlarge	8	30	4 minutos	11 minutos	48
c4.2xlarge	8	15	3 minutos	9 minutos	48
c4.4xlarge	16	30	2 minutos	8 minutos	96
c4.8xlarge	36	60	1 minuto	9 minutos	192
m4.2xlarge	8	32	3 minutos	11 minutos	48
m4.10xlarge	16	64	1 minuto	9 minutos	240

No me es posible exponer una conclusión clara. Por una parte, el umbral mínimo de ejecución (instanciación de los nodos, provisionamiento y configuración, procesos de depuración y reducción final) imponen un tiempo de ejecución total que nunca baja de 8 minutos. Por otro, intuyo que el tamaño de los *datasets* utilizados (lejos de los TBytes) hace que sea este umbral precisamente el factor limitante a la hora de evaluar las diferencias entre las alternativas. Por ejemplo, si comparamos c4.4xlarge y c4.8xlarge (la segunda con el doble de recursos que la primera), encontramos que incluso si el tiempo de ejecución del paso principal de Map Reduce, el tiempo necesario para instanciar todos los nodos del clúster hace el tiempo de ejecución total en el clúster con más recursos sea mayor que el requerido para el de menor recursos. Con los requisitos de la práctica, recomendaría alguna configuración de la gama C4 (optimizadas para cómputo, concretamente c4.2xlarge, configuraciones mayores no dan una mejora significativa).

[1] AWS Documentation » Amazon Elastic MapReduce Documentation » Developer Guide » Plan an Amazon EMR Cluster » Choose the Number and Type of Instances » Instance Configurations: http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-ec2-instances.html

Comentarios personales

La práctica parecía aparentemente "fácil" (lo es, de hecho), pero ha planteado diversas dificultades fruto, en general, de la falta de experiencia con sistemas HDFS y MapReduce. Como ya se ha comentado en apartados anteriores, se ha abordado la ejecución del código tanto en Hortonworks Sandbox como en AWS EMR con scripts basados en el paradigma Map Reduce Streaming. El uso de MRJob ha sido, comparativamente, muy inferior.

Los problemas encontrados durante la ejecución de la práctica han sido de varios tipos:

- He escrito los *scripts* y efectuado las pruebas locales en un entorno **Windows**. Tanto en Hortonworks Sandbox como en AWS EMR ha sido preciso **adaptar los scripts al formato Unix**. En caso contrario, era imposible ejecutar *mappers* y *reducers*, al no ser reconocidos los *scripts* como tales por el intérprete. Es preciso señalar que, aunque la detección en HwS ha sido sencilla, al efectuarse las pruebas desde línea de comandos y mostrarse de los errores de forma inmediata, en el entorno AWS EMR ha sido mucho más complicado, dada la "oscuridad" de los logs y mensajes de error. Fueron necesarias más de una decena de pruebas en esta última plataforma antes de caer en la cuenta de cuál era el error.
  En el caso de HwS, se ha usado `dos2unix`, tras ser instalado en el nodo máster (CentOS). En el caso de AWS EMR, se ha tenido que cambiar la configuración del IDE antes de subir los scripts a S3.

- En el caso de Hortonworks Sandbox, más allá de las no siempre evidentes formas de interactuar con la máquina virtual (el acceso ha de hacerse a través de un cliente ssh y no a través de la consola; el HDFS File System ha de inicializarse; el *hostname* del clúster es `sandbox.hortonworks.com`, por poner algunos ejemplos), se ha verificado que **los scripts Python que implementan las funcionalidades de *map* y *reduce* deben ser autocontenidos y usar solo los paquetes estándar de Python 2.7**. Aunque es posible instalar nuevos paquetes de Python en el nodo máster, estos son ignorados en la ejecución en los cores, por lo que, como mucho, es posible cargar datos desde ficheros de texto, pero no importar funciones o datos desde otros *scripts* Python (inicialmente, el diccionario de estados de Estados Unidos se planteó como un *script* en el contenido como diccionario nativo de Python). Aunque intuyo que es posible instalar módulos en los cores mediante alguna función de *bootstrapping*, no he encontrado la forma concreta.
  Por otra parte, el uso de MapReduce Streaming desde línea de comandos **no permite el encadenado de funciones de *reduce***, por lo que ha habido que ejecutarlas en dos pasos, utilizado HDFS como almacenamiento intermedio y un *identity mapper* (`/bin/cat`) como *mapper* de la segunda ejecución.

- AWS EMR ha planteado problemas de otro tipo. El primer grupo está relacionado con el **acceso a ficheros auxiliares** (los diccionarios antes citados), los cuales no son encontrados por los *scripts* en el sistema de ficheros. La solución ha sido acceder a los ficheros a través de un acceso web (https://s3-eu-west-1.amazonaws.com/urjc.datascience.mmonjas.emr/logic/AFINN-111.txt). Sin embargo, esta solución tampoco fue inmediata ya que los permisos por defecto no eran suficientes, por lo que hubo que asignarles permisos de lectura universales.
  Un segundo problema es consecuencia de la cantidad por defecto de *reducers* que implementa AWS EMR. En la configuración habitual (m3.xlarge: 1 master y 2 cores), se ejecutan siete *reducers*, lo cual se traduce en la generación de siete objetos distintos. Inicialmente se planteó la creación de un *script* local basado en `boto3` capaz de descargarlos desde S3 y concatenarlos localmente. Sin embargo, una rápida inspección permitió averiguar que los resultados de los *reducers* tienen que pasar de nuevo por un proceso de *reduce*, ya que tienen claves repetidas.

- El uso de MRJob ha sido complicado en el entorno que utilizo. Aunque uso Python 2.7 sobre Windows (la versión de Python es correcta), la librería no encontraba el fichero de configuración, posiblemente por tener espacios en el nombre de usuario (y, consecuentemente, en la ruta del directorio HOME). La implementación en Linux ha solventado esos problemas, pero se ha encontrado con otros problemas inesperados: la ejecución sobre AWS EMR me ha dado problemas de verificación de certificados debidos a la existencia de puntos en el nombre del *bucket*. Al final he tenido que saltarme la verificación de certificados:

```
import ssl
if hasattr(ssl, '_create_unverified_context'):
ssl._create_default_https_context = ssl._create_unverified_context
```

No he podido implementar de forma tan exhaustiva la versión MRJob por falta de tiempo, ya que ha requerido una reimplementación no trivial del código existente. Por ello, versión basada en MRJob es bastante inferior a la basada en Map Reduce Streaming y sufre las siguientes limitaciones:

- Se ha probado únicamente en local y en AWS EMR.
- No implementa la funcionalidad de búsqueda de los *trending topics*.
- No implementa el paso adicional requerido para combinar los ficheros producidos por AWS EMR.
- No se han realizado pruebas de tiempo de ejecución similares al de los programas basados en Map Reduce Streaming.

El tiempo dedicado está en torno a de las 60 horas, contando aquí no solo la programación del código sino la gestión tanto de Hortonworks Sandbox como la, tediosa, ejecución en EMR.

En relación con las sugerencias, tengo las siguientes:

- Abordar antes MRJob. Aunque se mencionó durante la fase "regular", el uso in situ no se hizo hasta el último día, cuando ya llevábamos tiempo abordando la práctica. Personalmente, he encontrado el código de la versión de MRJob más limpio y directo que el basado en Map Reduce Streaming, por lo que, a nivel de programación, me parece más elegante e intuitivo y posiblemente habría comenzado por este de haberlo conocido de primera mano al comenzar la práctica.
- Durante las clases con AWS EMR, abordar el *bootstrapping* de nodos para verificar que es posible instalar nuevos módulos de Python.
- Aclarar qué tipo de experimentación se requería para la sección Ejecución con MRJob
- Se muestran dos ejecuciones de la versión basada en MRJob. La primera es la ejecución local. La segunda la ejecución contra AWS EMR.

```
$ cat dump4.txt | python mrjob_twitter.py --runner inline --file us_states.txt --file
AFINN-111.txt > rs04mrjobs.txt
```

La ejecución contra AWS EMR se ha ejecutado de la siguiente forma:

```
$ python mrjob_twitter.py dump.txt -r emr
s3://urjc.datascience.mmonjas.emr/input/large/
--file AFINN-111.txt --file us_states.txt
--ec2-instance-type c1.medium
--num-core-instances 3
--output-dir=s3://urjc.datascience.mmonjas.emr/output/test_m11
```

- Evaluación de la escalabilidad.

Finalmente incluyo una sección de futuras mejoras que me gustaría abordar, aunque no haya sido capaz de implementar en esta versión del código:

- Utilización de generadores en el código Python.
- Implementación en MRJob.
- Uso de *composers*.