



Práctica 2. Apache Hadoop y Apache Spark.

INTELIGENCIA COMPUTACIONAL PARA DATOS DE ALTA DIMENSIONALIDAD.

Cristian Morillo Losada | Pedro Sánchez García

MÁSTER UNIVERSITARIO EN BIOINFORMÁTICA PARA CIENCIAS DE LA SALUD.
PROFESOR: DR. CARLOS EIRAS FRANCO.

PARTE 1: ALMACENAMIENTO EN HDFS

INTRODUCCIÓN.

El trabajo en los ordenadores personales conforma la ejecución de aplicaciones en una arquitectura de nodo único que comprende CPU, memoria y disco. En consecuencia, es posible que ciertas aplicaciones empleadas impliquen un mayor esfuerzo por la limitación de dicho hardware. Bajo esta idea, se ubica la implicación de un mayor número de máquinas en la ejecución, es decir, plantear un procesamiento distribuido que mejore el rendimiento. De este modo, podemos apreciar la relevancia del procesamiento distribuido en el entorno de análisis de Big Data.

La primera parte de la práctica 2 se centra en el almacenamiento en HDFS correspondiente a Hadoop. En términos generales, el proyecto Apache Hadoop se trata de un entorno de software diseñado en java para escalar desde servidores únicos a miles de máquinas, donde cada una proporciona computación y almacenamiento local. Con respecto a este último, el almacenamiento distribuido (HDFS: Hadoop Distributed File System) hace referencia a dos componentes principales: *namenode*, que consiste en el nodo maestro encargado de reconstrucción de ficheros, replicación y mantener ubicaciones y, por otro lado, los *datanodes*, donde se almacenan los ficheros y se encargan de crear, eliminar y replicarlos en base a las instrucciones del *namenode*.

En base a lo anterior, esta memoria presenta los pasos realizados para las operaciones con HDFS, muestra de comandos utilizados y las respuestas justificadas a las cuestiones que se plantean.

OBJETIVOS.

- Familiarizarse con el entorno de ejecución de las prácticas de la asignatura.
- Entender el funcionamiento de HDFS y el procesamiento MapReduce.

FUNCIONAMIENTO DE HDFS.

OPERACIONES BÁSICAS CON HDFS.

Se configura HDFS para el almacenamiento de los datos entre las ejecuciones realizadas. Para ello, se reemplaza el fichero `hdfs-site.xml` presente en `$HADOOP_HOME/etc/hdfs-site.xml` por el proporcionado en el material de la práctica. A continuación, se formatea el NameNode para el empleo en HDFS mediante el comando `hdfs namenode -format`.

De este modo, estamos en condiciones de crear un cluster Hadoop en nuestra máquina, el cual estará conformado por varios procesos que ejercerán de las diferentes máquinas. En primer lugar, por el comando `start-dfs.sh` se inicia el cluster con un *namenode* y un *datanode*. Una vez transcurridos unos segundos, obtenemos una salida como la mostrada en la **Figura 1**, con la puesta en marcha correspondiente.

```
bigdata@bigdata-virtualbox:~$ start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/bigdata/Software/hadoop-2.9.2/logs/hadoop-bigdata-n
amenode-bigdata-virtualbox.out
localhost: starting datanode, logging to /home/bigdata/Software/hadoop-2.9.2/logs/hadoop-bigdata-d
atanode-bigdata-virtualbox.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/bigdata/Software/hadoop-2.9.2/logs/hadoop-bi
gdata-secondarynamenode-bigdata-virtualbox.out
```

Figura 1. Salida en terminal del inicio correspondiente al cluster en Hadoop.

Un aspecto muy interesante en lo referente al estado del cluster es la posibilidad de consultar su estado por el comando `hdfs dfsadmin -report` o la dirección <http://localhost:50070>. En el caso de la salida por terminal, se alcanza un listado de información como la capacidad, réplicas y porcentaje empleado del DFS, junto con aspectos similares en el caso de los *datanodes* (Figura 2).

```
bigdata@bigdata-virtualbox:~$ hdfs dfsadmin -report
Configured Capacity: 20995678208 (19.55 GB)
Present Capacity: 6379364352 (5.94 GB)
DFS Remaining: 6379339776 (5.94 GB)
DFS Used: 24576 (24 KB)
DFS Used%: 0.00%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0
Pending deletion blocks: 0

-----
Live datanodes (1):

Name: 127.0.0.1:50010 (localhost)
Hostname: bigdata-virtualbox
Decommission Status : Normal
Configured Capacity: 20995678208 (19.55 GB)
DFS Used: 24576 (24 KB)
Non DFS Used: 13526192128 (12.60 GB)
DFS Remaining: 6379339776 (5.94 GB)
DFS Used%: 0.00%
DFS Remaining%: 30.38%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Wed Oct 05 16:32:28 CEST 2022
Last Block Report: Wed Oct 05 16:30:01 CEST 2022
```

Figura 2. Visualización del estado del clúster en terminal.

Con respecto a la dirección, es preciso destacar que podemos visualizar la misma información en el resumen principal (Figura 3). No obstante, se necesita acceder a diferentes secciones para observar los aspectos restantes (Figura 4).

Summary

Security is off.

Safemode is off.

1 files and directories, 0 blocks = 1 total filesystem object(s).

Heap Memory used 30.45 MB of 60.06 MB Heap Memory. Max Heap Memory is 966.69 MB.

Non Heap Memory used 38.87 MB of 39.56 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	19.55 GB
DFS Used:	24 KB (0%)
Non DFS Used:	12.6 GB
DFS Remaining:	5.94 GB (30.38%)
Block Pool Used:	24 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	Wed Oct 05 16:29:54 +0200 2022
Last Checkpoint Time	Wed Oct 05 16:27:41 +0200 2022

Figura 3. Resumen del estado del cluster por la interfaz web.

NameNode Journal Status

Current transaction ID: 1

Journal Manager	State
FileJournalManager(root=/tmp/hadoop-bigdata/dfs/name)	EditLogOutputStream(/tmp/hadoop-bigdata/dfs/name/current/edits_inprogress_0000000000000000001)

NameNode Storage

Storage Directory	Type	State
/tmp/hadoop-bigdata/dfs/name	IMAGE_AND_EDITS	Active

DFS Storage Types

Storage Type	Configured Capacity	Capacity Used	Capacity Remaining	Block Pool Used	Nodes In Service
DISK	19.55 GB	24 KB (0%)	5.94 GB (30.38%)	24 KB	1

Figura 4. Aspectos restantes sobre el estado del cluster por la interfaz web.

Posteriormente, se procede a la creación de un nuevo directorio en HDFS mediante el comando `hdfs dfs -mkdir /nuevo`, verificándolo con el comando `hdfs dfs -ls /`. Tal y como se puede apreciar en la **Figura 5**, se muestran los permisos concedidos en lectura, escritura y ejecución a la izquierda para dicho directorio, junto con información de fecha y hora en la que fue creado.

```
bigdata@bigdata-virtualbox:~$ hdfs dfs -ls /
Found 1 items
drwxr-xr-x  - bigdata supergroup          0 2022-10-05 16:43 /nuevo
```

Figura 5. Verificación en terminal de la creación del directorio *nuevo* en HDFS.

Tras verificar la creación del directorio, se añade el fichero *sample.vcf* en él a través del comando `hdfs dfs -put /media/sf_PR_ICDAD/Materiales-parte1/sample.vcf /nuevo`, de modo que luego se comprueba la correcta subida del fichero con `hdfs dfs -ls /nuevo` (**Figura 6**).

```
bigdata@bigdata-virtualbox:/media/sf_PR_ICDAD/Materiales-parte1$ hdfs dfs -ls /nuevo
Found 1 items
-rw-r--r--  1 bigdata supergroup      1749 2022-10-05 17:17 /nuevo/sample.vcf
```

Figura 6. Verificación en terminal de la correcta subida del fichero *sample.vcf* en HDFS.

Cabe destacar que resulta muy interesante analizar la forma y ubicación de almacenamiento para el fichero por parte de HDFS. Para ello, se utiliza el comando `hdfs fsck /sample.vcf -files -blocks -locations`, alcanzando el siguiente resultado:

```
bigdata@bigdata-virtualbox:/media/sf_PR_ICDAD/Materiales-parte1$ hdfs fsck /nuevo/sample.vcf -files -blocks -locations
Connecting to namenode via http://localhost:50070/fsck?ugi=bigdata&files=1&blocks=1&locations=1&path=%2Fnuevo%2Fsample.vcf
FSCK started by bigdata (auth:SIMPLE) from /127.0.0.1 for path /nuevo/sample.vcf at Wed Oct 05 17:22:49 CEST 2022
/nuevo/sample.vcf 1749 bytes, 1 block(s): OK
0. BP-1493831614-127.0.0.1-1664980061348:blk_1073741825_1001 len=1749 Live_repl=1 [DatanodeInfoWithStorage[127.0.0.1:50010,DS-72bba140-f28f-42c1-9c5a-1174551c17c3,DISK]]

Status: HEALTHY
Total size:      1749 B
Total dirs:      0
Total files:      1
Total symlinks:    0
Total blocks (validated): 1 (avg. block size 1749 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 1
Average block replication: 1.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 1
Number of racks: 1
FSCK ended at Wed Oct 05 17:22:49 CEST 2022 in 32 milliseconds

The filesystem under path '/nuevo/sample.vcf' is HEALTHY
```

Figura 7. Determinación del almacenamiento para el fichero *sample.vcf* en HDFS.

Tal y como podemos observar en la **Figura 7**, se muestra el tamaño del fichero (1749 B) y la ausencia de réplicas. Ahora nos centramos en lo relativo al nivel de replicación, de modo que atendiendo a un factor de replicación concreto junto con el comando `put`, establecemos la subida del fichero *musicbrainz_track* a HDFS bajo un factor de replicación 4. Para ello, se plantea el siguiente aspecto para el comando: `hdfs dfs -Ddfs.replication=4 -put /media/sf_PR_ICDAD/Materiales-parte1/musicbrainz_track.csv /nuevo`.

```
bigdata@bigdata-virtualbox: /media/sf_PR_ICDAD/Materiales-parte1$ hdfs fsck /nuevo/musicbrainz_track.csv -files -blocks -locations
Connecting to namenode via http://localhost:50070/fsc?ugi=bigdata&files=1&blocks=1&locations=1&path=%2Fnuevo%2Fmusicbrainz_track.csv
FSCK started by bigdata (auth:SIMPLE) from /127.0.0.1 for path /nuevo/musicbrainz_track.csv at Wed Oct 05 20:34:00 CEST 2022
/nuevo/musicbrainz_track.csv 112951768 bytes, 1 block(s): Under replicated BP-138646270-127.0.1.1-1664993595304:blk_1073741826_1002. Target
Replicas is 4 but found 1 live replica(s), 0 decommissioned replica(s), 0 decommissioning replica(s).
0, BP-138646270-127.0.1.1-1664993595304:blk_1073741826_1002 len=112951768 Live_repl=1 [DatanodeInfoWithStorage[127.0.0.1:50010,DS-81c394e1-9c
1f-46df-8739-c9087bdc8758,DISK]]
Status: HEALTHY
Total size: 112951768 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 1 (avg. block size 112951768 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 1 (100.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 1
Average block replication: 1.0
Corrupt blocks: 0
Missing replicas: 3 (75.0 %)
Number of data-nodes: 1
Number of racks: 1
FSCK ended at Wed Oct 05 20:34:00 CEST 2022 in 8 milliseconds
The filesystem under path '/nuevo/musicbrainz_track.csv' is HEALTHY
```

Figura 8. Información del almacenamiento para el fichero *musicbrainz_track.csv* en HDFS.

En la **Figura 8**, se recoge el resultado de la información sobre el almacenamiento del fichero *musicbrainz_track* en HDFS. Se puede apreciar que indica que el factor de replicación objetivo es de 4. Sin embargo, hay 1 réplica, pues al no disponer de nodos suficientes para lograr dicho nivel de replicación, se reflejan 3 *missing replicas*. Con el fin de lograr las réplicas deseadas, se procede a la simulación de procesos que simulan máquinas adicionales en el cluster. En consecuencia, se crean directorios temporales para cada una de ellas en nuestra máquina mediante los siguientes comandos: `sudo mkdir -p /tmpHadoop/slave1 /tmpHadoop/slave2 /tmpHadoop/slave3` y `sudo chown -R bigdata:bigdata /tmpHadoop/`.

La adición del nuevo *datanode* se lleva a cabo con el siguiente comando y se evalúa la información actualizada del almacenamiento para el fichero:

```
hdfs datanode -Ddfs.datanode.address=0.0.0.0:50100 -
```

```
Ddfs.datanode.http.address=0.0.0.0:50101 -
```

```
Ddfs.datanode.ipc.address=0.0.0.0:50102 -
```

```
Dhadoop.tmp.dir=/tmpHadoop/slave1/dfs -
```

```
Ddfs.datanode.data.dir=/tmpHadoop/slave1/dfs
```

```
bigdata@bigdata-virtualbox:~$ hdfs fsck /nuevo/musicbrainz_track.csv
Connecting to namenode via http://localhost:50070/fsc?ugi=bigdata&path=%2Fnuevo%2Fmusicbrainz_track.csv
FSCK started by bigdata (auth:SIMPLE) from /127.0.0.1 for path /nuevo/musicbrainz_track.csv at Mon Oct 10 19:10:44 CEST 2022
/nuevo/musicbrainz_track.csv: Under replicated BP-1065818100-127.0.1.1-1665222566529:blk_1073741826_1002. Target Replicas is 4 but found 2 l
ive replica(s), 0 decommissioned replica(s), 0 decommissioning replica(s).
Status: HEALTHY
Total size: 112951768 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 1 (avg. block size 112951768 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 1 (100.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 1
Average block replication: 2.0
Corrupt blocks: 0
Missing replicas: 2 (50.0 %)
Number of data-nodes: 2
Number of racks: 1
FSCK ended at Mon Oct 10 19:10:44 CEST 2022 in 23 milliseconds
The filesystem under path '/nuevo/musicbrainz_track.csv' is HEALTHY
```

Figura 9. Información actualizada del almacenamiento para el fichero *musicbrainz_track.csv* en HDFS.

Como resultado, por el comando `fsck`, se logra una *missing replica* menos, lo que pone de manifiesto que Hadoop ha replicado el fichero cuando dispuso de nodos (**Figura 9**). Esto es particularmente interesante, pues tal y como se ha mencionado en las clases expositivas, sin indicarlo de manera explícita, el *namenode* gestiona la replicación de bloques de fichero automáticamente en la arquitectura descrita. Además, se debe tener en cuenta que, en caso de posibles fallos en estos procesos, Hadoop cuenta con un sistema de puntos de control por parte del *namenode*. De este modo, para solucionarlos, el clúster no estaría disponible en su tolerancia a los fallos.

Progresivamente, se continua la adición de otros dos *datanodes* aplicando ligeras modificaciones en el comando anterior, correspondientes a los puertos y nombres en el *path*. Tal y como se esperaba, con la adición del segundo *datanode* se produce una pérdida de una *missing replica*:

```
bigdata@bigdata-virtualbox:~$ hdfs fsck /nuevo/musicbrainz_track.csv
Connecting to namenode via http://localhost:50070/fsc?ugi=bigdata&path=%2Fnuevo%2Fmusicbrainz_track.csv
FSCK started by bigdata (auth:SIMPLE) from /127.0.0.1 for path /nuevo/musicbrainz_track.csv at Mon Oct 10 19:13:00 CEST 2022

/nuevo/musicbrainz_track.csv: Under replicated BP-1065818100-127.0.1.1-166522566529:blk_1073741826_1002. Target Replicas is 4 but found 3 l
ive replica(s), 0 decommissioned replica(s), 0 decommissioning replica(s).
Status: HEALTHY
Total size: 112951768 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 1 (avg. block size 112951768 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 1 (100.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 1
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 1 (25.0 %)
Number of data-nodes: 3
Number of racks: 1
FSCK ended at Mon Oct 10 19:13:00 CEST 2022 in 1 milliseconds

The filesystem under path '/nuevo/musicbrainz_track.csv' is HEALTHY
```

Figura 10. Pérdida de *missing replica* para el fichero *musicbrainz_track.csv* en HDFS con adición del segundo *datanode*.

Por último, se añade el tercer *datanode*, lo que implica la ausencia de *missing replicas* como se aprecia a continuación:

```
bigdata@bigdata-virtualbox:~$ hdfs fsck /nuevo/musicbrainz_track.csv
Connecting to namenode via http://localhost:50070/fsc?ugi=bigdata&path=%2Fnuevo%2Fmusicbrainz_track.csv
FSCK started by bigdata (auth:SIMPLE) from /127.0.0.1 for path /nuevo/musicbrainz_track.csv at Mon Oct 10 20:00:09 CEST 2022
Status: HEALTHY
Total size: 112951768 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 1 (avg. block size 112951768 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 1
Average block replication: 4.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
FSCK ended at Mon Oct 10 20:00:09 CEST 2022 in 1 milliseconds

The filesystem under path '/nuevo/musicbrainz_track.csv' is HEALTHY
```

Figura 11. Ausencia de *missing replica* para el fichero *musicbrainz_track.csv* en HDFS con adición del tercer *datanode*.

En resumen, se logra el factor de replicación de 4 una vez que es posible por la presencia de nodos necesarios para ello.

Veamos ahora lo que sucede al subir una copia del fichero *musicbrainz_track* con un factor de replicación de 1 al directorio HDFS creado. Una vez que verificamos que se ha almacenado en uno de los *datanodes* añadidos, se procede a la finalización progresiva de los procesos de dichos *datanodes*. Tras ello, se analiza si es posible la descarga de los ficheros con los que se ha trabajado en HDFS y el estado del cluster.

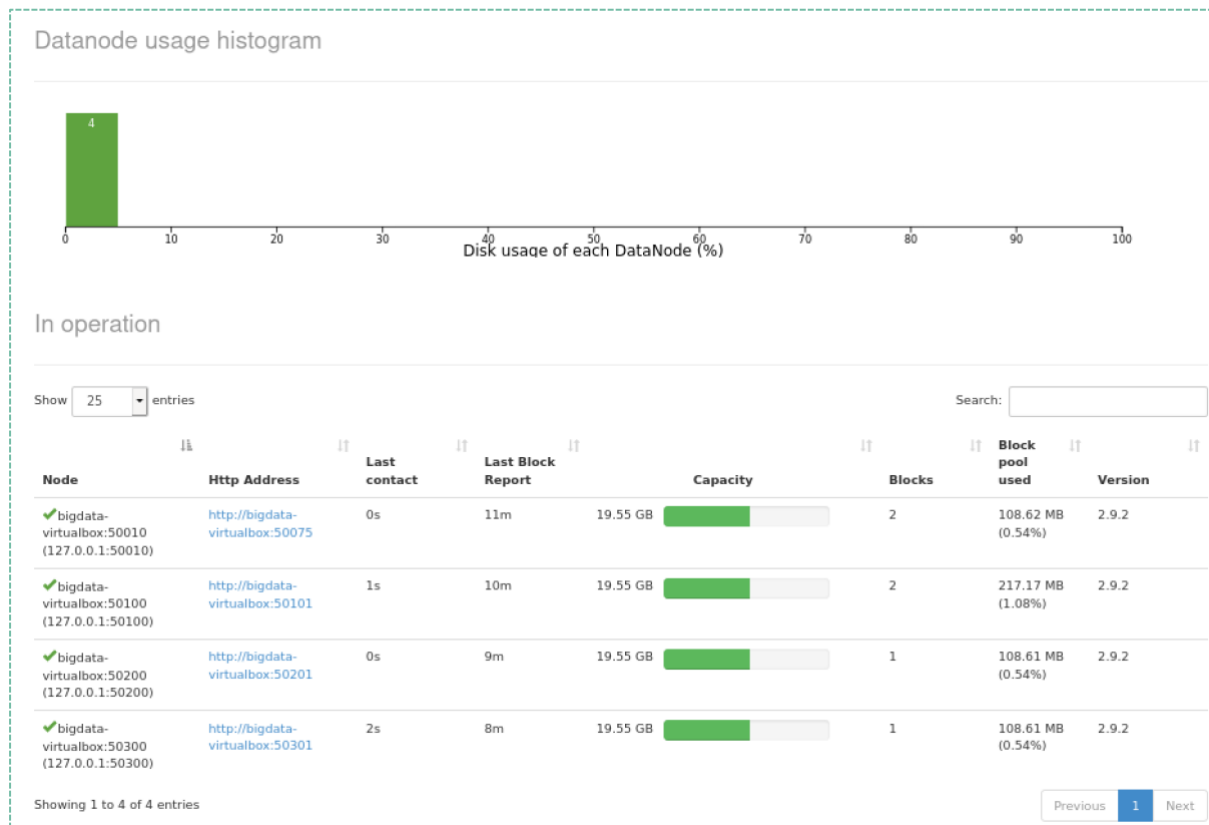


Figura 12. Informe de la interfaz web sobre el estado inmediatamente posterior a la adición de los *datanodes*.

Teniendo en cuenta el anterior esquema mostrado, se produce un cambio en la situación en cuanto se finaliza el proceso del primer *datanode* añadido (50100), ya que la interfaz web ofrece un aviso emergente sobre la pérdida de la copia del fichero *musicbrainz_track*:

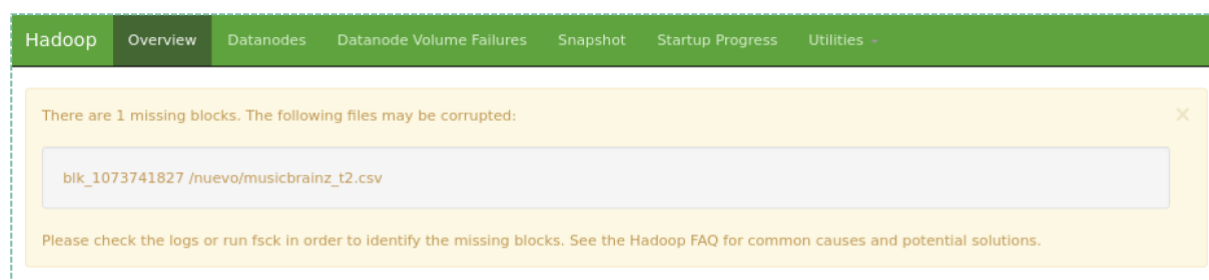


Figura 13. Aviso emergente sobre la pérdida de copia en fichero *musicbrainz_track* tras eliminación del *datanode*.

Este aspecto mostrado en la **Figura 13**, pone de manifiesto que el fichero deja de estar disponible cuando se produce una desconexión del nodo que posee la única copia del bloque. De esta forma, ante un factor de replicación superior a 1, Hadoop replicará dicho bloque cuando sea posible, por lo que, en el transcurso del proceso, solo cuando el nodo original desaparece, se daría una pérdida del acceso al fichero.

Debemos tener en cuenta que resulta crucial comprobar la disponibilidad de nodos con espacio de almacenamiento suficiente para el mantenimiento del nivel de replicación que desea el usuario. En el ámbito de trabajo, un factor de replicación elevado implica una notable reducción en posibilidades de pérdida de acceso al fichero, aunque requiere una mayor demanda de espacio de almacenamiento.

Por último, tal y como se ha mencionado anteriormente, la otra posible pérdida de acceso a ficheros en su totalidad es cuando se pierde el acceso al *namenode*, pues, aunque estén en los *datanodes*, falta esa capacidad de reconstrucción llevada a cabo por el *namenode*.

P1. ES HABITUAL UTILIZAR UN FACTOR DE REPLICACIÓN DE TRES. ¿POR QUÉ CREES QUE ES ESTO? ARGUMENTA EN QUÉ CASOS PODRÍA SER PREFERIBLE UTILIZAR VALORES DISTINTOS.

En base a lo expuesto, el factor de replicación de tres conforma un parámetro por defecto, pues en términos generales se ha valorado el equilibrio entre la accesibilidad a la réplica del fichero y el rendimiento. Para este último, es preciso atender a la lectura y escritura, ya que el incremento en el factor de replicación contribuye a un menor rendimiento en escritura y mejora en el de lectura. En consecuencia, el factor de replicación de tres ofrece el compromiso más adecuado entre accesibilidad y rendimiento para lectura y escritura, permitiendo el acceso a otras copias en caso de daños o falta de accesibilidad.

Por otro lado, de acuerdo con la arquitectura de Hadoop y la tolerancia a fallos, se podría usar un menor factor de replicación para aquellos casos en los que se manejen ficheros temporales que carezcan de importancia para un proyecto concreto. En caso contrario, incrementar dicho factor permitiría ofrecer una alta resistencia a aquellos ficheros más relevantes en una línea de trabajo determinada, valorando la decisión conforme al equilibrio mencionado.

PARTE 2: PROCESAMIENTO MAPREDUCE

INTRODUCCIÓN.

En esta parte, el punto de partida es un sistema de ficheros HDFS sobre el que se lleva a cabo un procesamiento. Este último, se basará en un modelo MapReduce para su gestión en el cluster que se maneja, con el fin de minimizar el trasiego de datos por la red, que proporciona una lentitud notable. Debemos tener en cuenta que MapReduce consta de una función *map*, donde se transforman los datos de entrada devolviendo tuplas (clave, valor) con frecuencia y posteriormente, se agrupan los conjuntos a través de la función *reduce*, de tal forma que se obtiene un resultado por cada clave de los pares (clave, valor) mencionados. Además, es preciso destacar la escalabilidad del modelo, junto con el manejo de posibles fallos y la comunicación asociada en clusters.

En lo que respecta a Hadoop, las aplicaciones MapReduce ejecutadas se encuentran escritas en Java, por lo que, para el presente caso en Python, se emplea la librería Streaming. Con esta, se lleva a cabo el planteamiento del *mapper* y *reducer*, donde el primero lee datos de entrada y proporciona como salida los pares (clave, valor), mientras que el segundo recibe como entrada dichos pares (clave, valor) ordenados por clave y procede a su agrupación de acuerdo con esta.

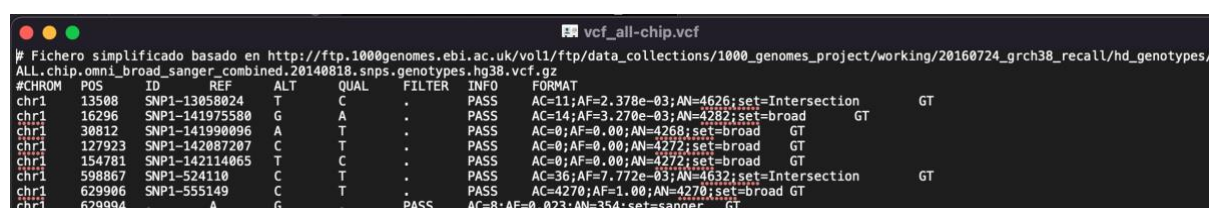
OBJETIVO.

- Escribir programas en Python siguiendo el paradigma MapReduce para su ejecución en Hadoop. De este modo, se plantearán un *mapper* y *reducer* por cada pregunta propuesta.

PREGUNTA 2.

P2. ESCRIBE UN PROGRAMA MAPREDUCE EN PYTHON QUE CUENTE CUÁNTAS VARIANTES APARECEN EN EL FICHERO AGRUPÁNDOLAS POR CROMOSOMA. DESCRIBE EL PROCESO DE EJECUCIÓN Y CÓMO SE HA DISTRIBUIDO EL TRABAJO.

Cabe destacar que se trabaja con el fichero *vcf_all-chip.vcf*, que es grande (203.1 MB), pues contiene información genómica en formato Variant Call Format (VCF). En consecuencia, el fichero contiene las variaciones presentes y su posición en el genoma de un individuo con respecto a un genoma de referencia:



```
# Fichero simplificado basado en http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data_collections/1000_genomes_project/working/20160724_grch38_recall/hd_genotypes/
ALL.chip.omni_broad_sanger_combined.20140818.snps.genotypes.hg38.vcf.gz
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT
chr1 13508 SNP1-13058024 T C . PASS AC=11;AF=2.378e-03;AN=4626;set=Intersection GT
chr1 16296 SNP1-141975580 G A . PASS AC=14;AF=3.270e-03;AN=4282;set=broad GT
chr1 30812 SNP1-141990096 A T . PASS AC=0;AF=0.00;AN=4268;set=broad GT
chr1 127923 SNP1-142087207 C T . PASS AC=0;AF=0.00;AN=4272;set=broad GT
chr1 154781 SNP1-142114065 T C . PASS AC=0;AF=0.00;AN=4272;set=broad GT
chr1 598867 SNP1-524110 C T . PASS AC=36;AF=7.772e-03;AN=4632;set=Intersection GT
chr1 629806 SNP1-555149 C T . PASS AC=4270;AF=1.00;AN=4270;set=broad GT
chr1 629994 . A G . PASS AC=8;AF=0.023;AN=354;set=sanger GT
```

Figura 14. Encabezado del fichero *vcf_all-chip.vcf*, donde se aprecian los diferentes campos de información.

Se sube el fichero en Hadoop bajo un factor de replicación de 1, pues conforma un mayor peso en comparación de otros empleados anteriormente como *sample.vcf* y el *musicbrainz_track.csv*. Además, esta decisión se asocia con el tipo de proyecto realizado, pues se llevan a cabo ejecuciones puntuales tras el planteamiento del *mapper* y *reducer*. No obstante, si se realizase un mayor número de pruebas de forma continua, se optaría por el factor estándar de replicación de 3.

A continuación, se procede a la realización del *mapper* correspondiente:

```
#!/usr/bin/env python
#coding=UTF8
"""mapper.py"""

import sys

# MODIFICA SOLO ESTA FUNCIÓN
def funcion_map(elemento):
    """
    Esta función recibe como entrada una cadena de texto y devuelve una tupla
    La salida de esta función se pasará al agrupador para posteriormente hacer el paso reduce
    """
    linea=elemento
    if linea.startswith('#'):
        return (None, 1)
    else:
        linea_procesada = linea.strip().split()
        clave = linea_procesada[0]
        valor = 1
        return (clave, valor)

#####
## NO HAGAS MODIFICACIONES A PARTIR DE AQUÍ
#####

# Cuerpo principal del script. Lee la entrada, la procesa para separar los elementos, a los que
# la entrada proviene de STDIN (standard input)
for line in sys.stdin:
    # se eliminan los espacios al principio y final
    line = line.strip()

    mapped = funcion_map(line)

    if mapped[0]: # Las tuplas con clave None no se emiten
        # Para emitir la tupla por STDOUT debemos transformarla en string. Usaremos un separador
        print('%s%s' % (str(mapped[0]), str(mapped[1])))
```

Figura 15. *Mapper* desarrollado para la pregunta 2, donde se aprecia el ajuste realizado en la función *map* para lograr las tuplas (clave, valor) correspondientes.

Tal y como se puede apreciar en la **Figura 15**, lo que se ha establecido es la recepción de cada línea del fichero como el elemento de partida en la función. Se omiten aquellas líneas correspondientes al encabezado (#) y se procesa cada una de las líneas restantes, eliminando espacios y convirtiendo la cadena de texto en una lista. De este modo, la clave perteneciente al cromosoma es el primer elemento [0] de dicha lista, mientras que el valor asociado es siempre 1. Finalmente, la función devuelve tuplas (clave, valor) deseadas, las cuales se trasladan al agrupador para la fase *reduce*.

Posteriormente, se procede a la generación del *reducer*, con la función *reduce* que recibe dos valores en forma de cadena de texto y asociados a una misma clave. Por tanto, se opera la suma de ambos, tal y como se muestra a continuación, de modo que, aplicándose a la lista de valores, se alcanza un valor total asociado a dicha clave en cuestión.

```
#!/usr/bin/env python
#coding=UTF8
"""reducer.py"""

from functools import reduce
import sys

# MODIFICA SOLO ESTA FUNCIÓN
def funcion_reduce(elem1, elem2):
    """
    Esta función recibe como entrada una dos
    Esta función se aplicará sucesivamente a
    """
    return int(elem1) + int(elem2)
```

Figura 16. *Reducer* desarrollado para la pregunta 2, donde se muestra la suma de valores aplicada en la función *reduce*.

Una vez que se ha evaluado el funcionamiento adecuado del *mapper* y *reducer* en el terminal de nuestra máquina sobre una versión simplificada del fichero, se lleva a cabo el procesamiento por la distribución de Hadoop. Para ello, se utiliza el comando:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-*streaming* -file
/media/sf_PR_ICDAD/Materiales-parte1/mapper_p2.py -mapper mapper_p2.py -file
/media/sf_PR_ICDAD/Materiales-parte1/reducer_p2.py -reducer reducer_p2.py -input /nuevo/vcf_all-
chip.vcf -output /nuevo/resultado
```

El resultado obtenido (part-00000) se almacena en un directorio de salida, lográndose la siguiente estructura:

```
bigdata@bigdata-virtualbox:~$ hdfs dfs -cat /nuevo/resultado/part-00000
chr10 ----> 123799
chr11 ----> 120969
chr12 ----> 116588
chr13 ----> 86096
chr14 ----> 79406
chr15 ----> 74753
chr16 ----> 80986
chr17 ----> 70372
chr18 ----> 70720
chr19 ----> 52452
chr1 ----> 190988
chr20 ----> 59238
chr21 ----> 33575
chr22 ----> 35924
chr2 ----> 200146
chr3 ----> 168214
chr4 ----> 157635
chr5 ----> 150227
chr6 ----> 157293
chr7 ----> 134012
chr8 ----> 129813
chr9 ----> 107238
chrX ----> 55070
chrY ----> 2031
```

Figura 17. Resultado del programa MapReduce con el recuento de variantes agrupadas por cromosoma.

Para la ejecución efectuada, con la correspondiente salida de cromosomas y variantes determinadas en cada uno de ellos, nos centramos ahora en el proceso que tendría lugar en un cluster real. Tal y como se ha mencionado anteriormente, un aspecto clave de MapReduce es la escalabilidad, de modo que se minimiza el flujo de datos. Para ello, ante un conjunto de datos con un peso considerable, este se divide en bloques, de modo que se repartiría la ejecución del *mapper* y *reducer* entre los *datanodes* disponibles del cluster. Con respecto a esto último, la distribución de los datos se produciría en base a la proximidad de los *datanodes*, permitiendo una eficiencia acorde a la arquitectura empleada y la implicación en el cómputo optimizado sobre el fichero.

PREGUNTA 3.

P3. ESCRIBE OTRO PROGRAMA SIMILAR QUE, USANDO LA SALIDA DEL PROGRAMA ANTERIOR COMO ENTRADA, IDENTIFIQUE EL CROMOSOMA CON MÁS VARIANTES.

Se toma como entrada el resultado alcanzado en la pregunta anterior (part-00000) y se plantea el siguiente *mapper*:

```
#!/usr/bin/env python
#coding=UTF8
"""mapper.py"""

import sys

# MODIFICA SOLO ESTA FUNCIÓN
def funcion_map(elemento):
    """
    Esta función recibe como entrada una cadena de texto y devuelve una tupla
    La salida de esta función se pasará al agrupador para posteriormente hacer el paso reduce
    """
    linea=elemento
    linea_procesada = linea.strip().split()
    clave = 1
    cromosoma = linea_procesada[0] + '-'
    valor = cromosoma+linea_procesada[2]
    if len(cromosoma) < 6:
        crom_mod = cromosoma + '-' # añadir guión para casos donde longitud del nombre de cromosoma sea inferior a 6
        valor_mod = crom_mod+linea_procesada[2]
        return (clave,valor_mod)
    else:
        return (clave,valor)
```

Figura 18. *Mapper* desarrollado para la pregunta 3 con ajustes en la función *map* para lograr las tuplas (clave, valor) o (clave, valor_mod) correspondientes.

Al igual que en la pregunta anterior, se comienza con la recepción de cada línea del fichero como el elemento de partida en la función. Se procesa cada una, eliminando espacios y convirtiendo la cadena de texto en una lista. En este caso, la clave es 1 con el fin de llevar a cabo la agrupación de tuplas (clave, valor) correspondientes. Cabe destacar que la variable denominada *cromosoma* está conformada por el primer elemento de la lista concatenado con un carácter '-' para separación. Por su parte, el valor es dicha variable *cromosoma* concatenada con el número de variaciones, es decir el segundo elemento [2] de la lista. Además, se establece la adición de un carácter '-' extra en aquellos casos con cromosomas inferiores a 10. Para estos casos, se establece un valor modificado que consta de ese carácter extra. Este aspecto es relevante para la posterior realización del *reducer*:

```
#!/usr/bin/env python
#coding=UTF8
"""reducer.py"""

from functools import reduce
import sys

# MODIFICA SOLO ESTA FUNCIÓN
def funcion_reduce(elem1, elem2):
    """
    Esta función recibe como entrada una dos cadenas de texto, que se
    Esta función se aplicará sucesivamente a los elementos la lista de
    """
    num1 = int(elem1[6:])
    num2 = int(elem2[6:])
    maximo = max(num1, num2)
    maximo_str = str(maximo)

    if maximo_str in elem1:
        return elem1
    else:
        return elem2

    return elem2[:6] + str(max(int(elem1[6:]), int(elem2[6:])))
```

Figura 19. *Reducer* planteado para la pregunta 3, donde se muestra la determinación aplicada en la función *reduce* para el cromosoma concreto

Tal y como se refleja en la **Figura 19**, el *reducer* parte de los elementos recibidos como *string*, seleccionando la posición 6 en adelante y convirtiendo a entero. Posteriormente, se determina el máximo, que se convierte a *string*. De este modo, si este se encuentra localizado en un elemento 1, se devuelve este. En caso contrario, se devuelve el elemento 2, por lo que finalmente la función *reduce* proporciona la región del *string* con el nombre del cromosoma [:6] concatenada con el mayor número de variaciones determinado.

De este modo, la clave perteneciente al cromosoma es el primer elemento [0] de dicha lista, mientras que el valor asociado es siempre 1. Finalmente, la función devuelve tuplas (clave, valor) deseadas, las cuales se trasladan al agrupador para la fase *reduce*.

Se evalúa el funcionamiento correcto del *mapper* y *reducer* en el terminal de nuestra máquina sobre el fichero correspondiente, de tal forma que finalmente se lleva a cabo el procesamiento por la distribución de Hadoop. Para ello, se utiliza el comando indicado en la pregunta 2 modificando simplemente el input, output y directorio para conservar el resultado. A continuación, se refleja el aspecto de la solución que devuelve el programa MapReduce al problema propuesto:

```
bigdata@bigdata-virtualbox:~$ hdfs dfs -cat /nuevo/resultado_p3/part-00000
1 ----> chr2--200146
```

Figura 20. Resultado del programa MapReduce con el cromosoma que posee mayor número de variantes.

Se concluye que el cromosoma 2 es el que presenta un mayor número de variantes, en concreto, 200.146.

PARTE 2: APACHE SPARK

INTRODUCCIÓN.

En esta última parte de la práctica, se trabaja con Apache Spark, que conforma un *framework* de procesamiento distribuido de código abierto, donde se pretende mantener un compromiso en la eficiencia de ejecución y la simplicidad de código, el cual puede ser en Python, Java y R. Si tenemos en cuenta lo visto anteriormente con Hadoop, es preciso destacar que este se caracteriza por una complejidad en código y lentitud para ciertos proyectos. Con el fin de superar este tipo de limitaciones, Apache Spark se basa en mantener los datos en memoria durante las ejecuciones, de modo que se mejora notablemente la velocidad. Por tanto, junto con la simplificación del código y la integración con partes de Hadoop (YARN, HDFS...), Apache Spark es un avance al respecto, por lo que se han desarrollado numerosas librerías para ofrecer soluciones rápidas y sencillas en los diversos sectores donde se emplea.

Debemos tener en cuenta que el fundamento de Apache Spark comprende la organización por regresión logística de los datos que se manejan. En consecuencia, aparecen el concepto de *Resilient Distributed Datasets (RDDs)*, que constituyen el núcleo donde se efectúan las operaciones, pues es una colección o conjunto de datos distribuido capaz de soportar fallos en máquina.

El trabajo con Apache Spark presenta un ciclo de vida característico para los RDDs, que son la creación, transformaciones, acciones y el almacenamiento final. No obstante, este ciclo no cumple ese orden siempre, ya que las transformaciones son una forma de creación, que trasladan al comienzo. En lo referente a las transformaciones, se distinguen *map* y *flatMap*, mientras que, para las acciones, hay *reduce*, *reduceByKey*, *groupByKey*... Cada una de estas, presenta unas cuestiones a las que se debe atender de cara al rendimiento por los posibles cálculos parciales. Este aspecto, junto con una serie de trampas comunes mencionadas durante la sesión magistral, implica valorar el código empleado para el proyecto que se esté realizando con Apache Spark.

OBJETIVOS.

- Aprender a programar y ejecutar programas distribuidos con Spark.
- Dominar la consola e interfaz web de Spark.
- Entender los retos a los que nos enfrentamos al paralelizar un código secuencial.

PREVIO.

PREVIO. IMPLEMENTA EL ALGORITMO CON PYTHON (SECUENCIAL, NO DISTRIBUIDO) PARA AJUSTAR DOS CADENAS SEGÚN LO DESCRITO. TAMBIÉN PUEDES BUSCAR UNA IMPLEMENTACIÓN YA HECHA Y VERIFICAR QUE FUNCIONE.

Se toma como punto de partida la problemática del alineamiento de secuencias, que implica un proceso con alto coste computacional asociado a la longitud de las cadenas empleadas. En concreto, esta parte se centra en el uso de Apache Spark para trabajo con el “*Highest-Scoring Fitting Alignment of Two Strings*”, de modo que es preciso establecer el correspondiente algoritmo para ajustar las cadenas según este caso. Para ello, se opta por una implementación simplificada y desarrollada por el usuario ‘egeulgen’, que se obtiene de GitHub:

https://github.com/egeulgen/Bioinformatics_Textbook_Track/blob/master/solutions/BA5H.py

```
def fitting_alignment(v, w):
    v = "-" + v
    w = "-" + w
    indel_penalty=1
    score_mat = [[0 for _ in range(len(w))] for _ in range(len(v))]
    backtrack_mat = [[None for _ in range(len(w))] for _ in range(len(v))]
    for i in range(1, len(v)):
        for j in range(1, len(w)):
            score1 = score_mat[i - 1][j - 1] + (1 if v[i] == w[j] else - 1)
            score2 = score_mat[i - 1][j] - indel_penalty
            score3 = score_mat[i][j - 1] - indel_penalty
            score_mat[i][j] = max(score1, score2, score3)
            if score_mat[i][j] == score1:
                backtrack_mat[i][j] = "d"
            elif score_mat[i][j] == score2:
                backtrack_mat[i][j] = "u"
            elif score_mat[i][j] == score3:
                backtrack_mat[i][j] = "l"
    j = len(w) - 1
    i = max(enumerate([score_mat[row][j] for row in range(len(w) - 1, len(v) - 1)], key=lambda x: x[1])[0] + len(w) - 1)
    max_score = score_mat[i][j]
    aligned_1 = aligned_2 = ""
    while backtrack_mat[i][j] is not None:
        direction = backtrack_mat[i][j]
        if direction == "d":
            aligned_1 = v[i] + aligned_1
            aligned_2 = w[j] + aligned_2
            i -= 1
            j -= 1
        elif direction == "u":
            aligned_1 = v[i] + aligned_1
            aligned_2 = "-" + aligned_2
            i -= 1
        else:
            aligned_1 = "-" + aligned_1
            aligned_2 = w[j] + aligned_2
            j -= 1
    return max_score, aligned_1, aligned_2
```

Figura 21. Algoritmo empleado para la presente parte de la práctica con resolución del alineamiento.

Con respecto a la estructura de la implementación, cabe destacar la construcción de la matriz de puntuaciones junto con los posteriores cálculos para lograr el valor final en puntuación y las cadenas modificadas con *gaps* tras el análisis.

PREGUNTA 1.

P1. UTILIZA SPARK PARA, DADA LA CADENA DE REFERENCIA DEL FICHERO CADENA.TXT, ENCONTRAR EL *HIGHEST-SCORING FITTING ALIGNMENT* DE TODAS LAS CADENAS INCLUIDAS EN EL FICHERO DATASET.TXT. AL ACABAR SE DEBE DEVOLVER, PARA LA CADENA SOBRE LA QUE CONSIGUE MAYOR PUNTUACIÓN Y PARA LA QUE SE CONSIGUE LA MÍNIMA PUNTUACIÓN, EL *SCORE* Y LAS CADENAS MODIFICADAS (CONTENIENDO *GAPS*), TAL COMO SE MUESTRAN EN ROSALIND.

Se plantea el script Spark que lee del fichero dataset.txt. La estructura contiene en primer lugar, el cuerpo de la función para el alineamiento de secuencias tratado. Posteriormente, se encuentra la cadena de referencia del fichero cadena.txt, la lectura de cadena.txt mediante el método `textFile` del objeto `SparkContext` `sc` (que conforma el RDD empleado) y los correspondientes RDDs que dan respuesta a la pregunta formulada:

```
def fitting_alignment(v, w):
    v = "-" + v
    w = "-" + w
    indel_penalty=1
    score_mat = [[0 for _ in range(len(w))] for _ in range(len(v))]
    backtrack_mat = [[None for _ in range(len(w))] for _ in range(len(v))]
    for i in range(1, len(v)):
        for j in range(1, len(w)):
            score1 = score_mat[i - 1][j - 1] + (1 if v[i] == w[j] else - 1)
            score2 = score_mat[i - 1][j] - indel_penalty
            score3 = score_mat[i][j - 1] - indel_penalty
            score_mat[i][j] = max(score1, score2, score3)
            if score_mat[i][j] == score1:
                backtrack_mat[i][j] = "d"
            elif score_mat[i][j] == score2:
                backtrack_mat[i][j] = "u"
            elif score_mat[i][j] == score3:
                backtrack_mat[i][j] = "l"
    j = len(w) - 1
    i = max(enumerate([score_mat[row][j] for row in range(len(w) - 1, len(v) - 1)]), key=lambda x: x[1])[0] + len(w) - 1
    max_score = score_mat[i][j]
    aligned_1 = aligned_2 = ""
    while backtrack_mat[i][j] is not None:
        direction = backtrack_mat[i][j]
        if direction == "d":
            aligned_1 = v[i] + aligned_1
            aligned_2 = w[j] + aligned_2
            i -= 1
            j -= 1
        elif direction == "u":
            aligned_1 = v[i] + aligned_1
            aligned_2 = "-" + aligned_2
            i -= 1
        else:
            aligned_1 = "-" + aligned_1
            aligned_2 = w[j] + aligned_2
            j -= 1
    return max_score, aligned_1, aligned_2

cadena = 'TAATCCGGACAATATTAGTGATATAGGGGCTCGTCTCGTGTCAAAATCGGGGTCGGGAGTCATAATGGTGTACGGGTGTATTGACCGAACCTCCGTCCTATCCCCCTCCC'
rdddataset=sc.textFile('/media/sf_PR_ICDAD/Materiales_parte2/dataset.txt')
rddmax=rdddataset.map(lambda w: fitting_alignment(w, cadena)).max(lambda s: s[0])
rddmin=rdddataset.map(lambda w: fitting_alignment(w, cadena)).min(lambda s: s[0])
```

Figura 22. Script Spark planteado para determinar *Highest-Scoring Fitting Alignment* de cadenas propuestas.

Con respecto al `rddmax` y `rddmin` reflejados en la Figura 22, se establece la conversión del `rdddataset` en otro que contiene el *score* alcanzado, cadena 1 y cadena 2 logradas con sus correspondientes *gaps*, manteniendo la salida de Rosalind. Para ello, se utiliza `map`, con función `lambda`, donde se aplica la función del alineamiento para cada elemento del dataset con la cadena de referencia, seleccionando posteriormente el máximo/mínimo con indicación del elemento correspondiente, es decir, el *score*.

A continuación, se muestra una captura con los resultados alcanzados empleando el script:

```
El score más alto es 81 con cadena 1: TGACTTTGGAAAACATGGGTGATGTTCTACGCAAGGTCGAATCTAGACCG-GCTAGTCCATGTAC-AGG-CAGCGG-GTCACTAGTGG-GTTA--G-GTCC
CCG--CGCAAGTCGCAAGAT--TAT-TCGCTGCGGCACGCCAGCGCATTTTG-G--TGTA--GGGACTTTTCGAGGTGCGTC-CA-CA-AGT-TCC-ACAA-CCGG-GCCTA-CCT-GACGAGAACGAC-CCCGA-G-GGG
TGGGAGATATTTCTGGGCTTAGCTACACGCTACTGGGACC-TAGGGGCGGAGAGGAATCAATACCGCTGTATAAAGGAGGAAATGCAAACTAGTGTCTCTCATTCGAGTCCAGATG-TACGGAACACGCTCCGT
CCCAT-ACCCTCGCAAGGTAGTGGGAGG-TACAACCTTTA-GGTT--CGCG-CA-TAAGAAACCGAGGAGT-GTCCATTACTAGACACC---GATGAG-C--TCGCGCAATTGCTAAC--GTCATGCTCCTGG-ATGT
C-GAAGCC-AG--CTCGA-CGC-TCTTG-TGTGTAA-ATTAACACTTCTGT--AGGCGCT-CCATG

y cadena 2: T-AATCCGGACAATATTAGTGA--T-ATA-G--GGGGTCG--TCT---CGTGCT-GTCAAAAT-CGGGGTC-G-GGAGTCA-TAATGGTG-TACGGTGTGATTGACCG-AA-CCTCC--G-TCCTA
TCCCCCTCCCAAG-GAGCGCACTTTGCGAATCTACCAGC-CGTTAGA-GCGCTCGCACCACA-TCACCGACAACCGCGCGCGAGCCTAGA-GAAAACGGCTCTAGACCGAGG-ACGA-CTA-ATC--GAATTAG--A
A----AAAT-GTCCCATATGAGC-GAGTTTAA-C--T--GTGGTGT-T----A-GAAGTCT-C--AAC-AGAGACTCATCA-TC---TCCACATGCT-C-CAATACG-CACCG----ATGA---T-C-ATG-AG-GCG
TGGCTTACCCTTTACACTTGGCGTGTATTGAG-ATA-GG-GGA-TAG--CATAACCGGAC-CCTAAGGATG-GCCTATAGACG-AATTGAT-CCTTTTAATGTTAGGGTACGTCGGGTGCCGAGTTCTCTATC-CATG
GAGCTG-GCGAACA-GATCACTTCTGTAATTGCGTACC-TG

El score más bajo es 56 con cadena 3: TTTTCC--ATGAT-TT-GTCGGCGTAGGAAGT-G-CGCGATAGC-G-CTGAATCGGACGCAATGAAATGTAACCTCATAATTGGCTCTTACCGTGTGTTTG
GCTTTAGTGGCTCATTTCATTTCAATCGTACAGAAAATCCCGCT-CAACAAGAATTGCACTGAGCCGGAATGC-AGGCTGCTAAC-CGGATGGATGATCTCATGTGGGCTGGTTTCATTACCGG-CGGCCCGCG-CGCAA
CGAAG-GATAACCTCTTTTGAAGTACAGTAGCGGAAGCTAATAACAATCTATTACCTGAGA-GACTC-TTTTA-CG-GTCTTCTCTAG-GGCGG-GAGGAAATCATC--CGAGAGAC-GA-C--CT-CACATGCCCC
CAT--GACATTGGTGAAGTGGTATGCGGTTT-TAGTTTC-CCTGACTCTAGGGCAGTCAGCC--GCTCCTG-G--GGGGACAGCAGCGTCA-CGAACGGTCAAGG-TAGGAAGACAGGACGCAA--G--CC-AGCAA-G
-AC-GCGG-AAAT-GGGACGCCCCG--CTGTTTCCA-GG-CCTTTCGTCTCA-ATCAAC--CTG-ACCA-GCGGAGGTG

y cadena 4: TAATCCGGACAATATTAGT-GATATAGGGGTCGTCTCG-T-GCTGTCAAAATCGG-GGTC---G---G-GAGT-CATAA-TGG-T-GTA-CG-GTG--T-G-ATT-GACCG--A---ACCTC---C
GT-C---CTA-TCCCCCTCCCAAGGAGCGCACTTTG--CGAAT-CTA--C--C--ACGC-G-T-TA-GAGCGC--GT-CGC-ACCACATCACCAGAAC--CCGGCGCGG-AGCCTAGAGAAAACGGCTCTAGA-C-GC
AG--GAGG-A-CTAAT-C--GAAT-TA-GA--AAAATGTCCTATGAGCGAGT-TT-AACT-GTGG-TGTTA-GAAGTC-TCAAC-ACAGACTCATCTCTCCACATG-CTCCAATACG-CACCGATGA-T-C--ATGA
GG--CGTGGCTTACC--CTTTA--CACT-TGCCGTGTTATTGAGATAGGGGATAGCA---TAACCGGACCCT-AAGGAT-GGCCTATA-GACG-AATTGATCCTTTAATGTTTCAGG-GTACGTGGG--TGCCGAG
TCTCTATCCATGGAGTGGCG-AACAGATC-ACCTCTGTA-AATTGCGTACCTG
```

Figura 23. Resultado del alineamiento empleando el script Spark planteado para la pregunta formulada.

De este modo, se logra una salida que mantiene la estructura de Rosalind, lo que también nos permite analizar la presencia de *gaps* en las cadenas modificadas.

PREGUNTA 2.

P2. DETECTA LOS PRINCIPALES CUELLOS DE BOTELLA DE TU CÓDIGO (LAS INSTRUCCIONES QUE CONSUMEN MÁS TIEMPO DE EJECUCIÓN). ¿CUÁLES SON Y POR QUÉ?

En términos generales, las instrucciones que consumen mayor tiempo de ejecución son la obtención del `rddmax` y `rddmin`, pues debemos tener en cuenta que aplica *map*, la función planteada para el alineamiento y finalmente es seleccionado el máximo/mínimo *score* junto con las cadenas modificadas correspondientes. De este modo, dado que la función del alineamiento es compleja a pesar la estructura simplificada, se produce un cuello de botella en esta por las numerosas secuencias manejadas, junto con la determinación del *score*. Por otro lado, instrucciones más simples de exploración en RDDs como la obtención del primer elemento en `rddmax` mediante *first* o aplicar *count* para determinar número de elementos, no consumen un tiempo de ejecución elevado.

PREGUNTA 3.

P3. EJECUTA EL CÓDIGO PARALELO UTILIZANDO 1, 2, 4 Y 8 WORKERS Y DIBUJA UNA GRÁFICA CON LOS TIEMPOS DE EJECUCIÓN. SI ESTÁ BIEN PARALELIZADO, A MAYOR NÚMERO DE WORKERS EL TIEMPO DE EJECUCIÓN DEBERÍA SER MENOR, AUNQUE SIEMPRE CON UN LÍMITE (LEY DE AMDAHL). ADEMÁS, VA A DEPENDER DEL HARDWARE DISPONIBLE ASÍ QUE ES POSIBLE QUE EN TU MÁQUINA VIRTUAL NO SE OBSERVE MEJORA. COMENTA LA GRÁFICA INDICANDO SI HAY MEJORAS O NO Y POR QUÉ. ¿CUÁNTAS TAREAS SE HAN CREADO EN CADA CASO? ¿CUÁNTAS SE HAN EJECUTADO EN PARALELO? ¿CUÁNTO TIEMPO SE DEDICÓ REALMENTE A EJECUTAR?

Para la resolución de esta pregunta se han planteado scripts específicos para cada ejecución en paralelo. El código tratado anteriormente para la pregunta 1 se inserta en cada uno de ellos junto con la definición del SparkContext, de tal forma que el aspecto es similar al siguiente:

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("p3_1_worker").setMaster("local[1]")

sc = SparkContext(conf=conf)

## CÓDIGO EMPLEADO (EN NUESTRO CASO EL REFLEJADO EN PREGUNTA 1) ##

sc.stop()
```

Tal y como se recoge en el ejemplo anterior, cada script presenta una configuración diferente en lo que respecta al número de *workers* para la ejecución (en este caso es para 1 *worker*), que se ajusta en *setMaster('local[x]')*, junto con una identificación en *setAppName*. Para el lanzamiento de esta aplicación de Spark, se utiliza el comando: `spark-submit <script_planteado>`, obteniendo unos resultados en tiempo de ejecución que se muestran en la siguiente gráfica:

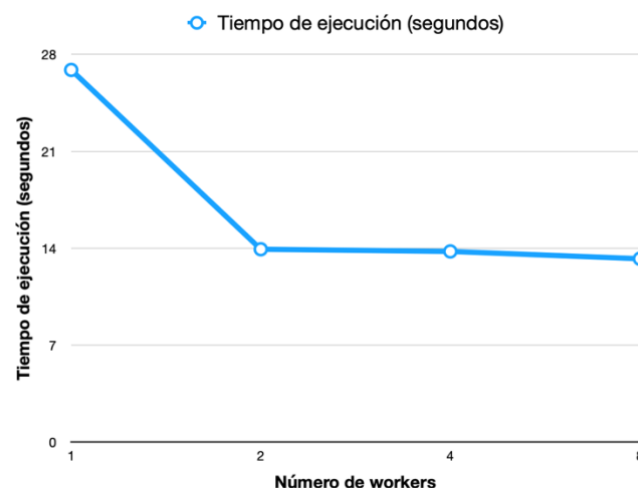


Figura 24. Tiempos de ejecución (segundos) para los *workers* empleados de la ejecución en paralelo.

En términos generales, la ejecución con 1 *worker* se mantiene en torno a los 27 segundos, mientras que con 2 *workers* se produce un drástico descenso en el tiempo hasta aproximadamente 14

segundos. Por otra parte, con 4 y 8 *workers* no se aprecian grandes variaciones, manteniendo una tendencia similar al caso de 2 *workers* (Figura 24).

El acceso a la interfaz web que habilita Spark mientras se está ejecutando, nos permite comprobar información sobre el estado de los trabajos correspondientes. A excepción del caso con 1 *worker* donde se genera 1 tarea, el resto de los casos implican 2 tareas, por lo que debemos centrar la atención en aquellos casos donde estas se han ejecutado en paralelo. Para ello, una forma de analizarlo es si se cumple que el tiempo total destinado a un trabajo es inferior a la suma del tiempo para las tareas que lo conforman. De este modo, las 2 tareas se han ejecutado en paralelo para los casos de 2, 4 y 8 *workers*, pues el tiempo destinado a cada una de las tareas es notablemente superior al total del trabajo, tal y como nos devuelve Spark.

PREGUNTA 4.

P4. MEJORA EL RENDIMIENTO DE TU CÓDIGO SPARK UTILIZANDO CACHES. VALORA EN QUÉ PUNTOS PUEDE SER ÚTIL EMPLEARLAS Y EN CUÁLES NO. JUSTIFICA TU DECISIÓN.

El empleo de cache facilita la reducción del tiempo en ejecuciones que impliquen coste computacional notable como se ha visto anteriormente. En consecuencia, para aquellos casos donde se vaya a reutilizar un RDD con frecuencia, se evita que Spark tenga que proceder a su cálculo desde el inicio calcularlo desde el principio. Por tanto, dicho cálculo se guarda principalmente en memoria, aunque si es preciso, se almacena en disco.

Para el caso de 1 *worker*, se ha obtenido una reducción del tiempo de computación notable, bajando de los 11 segundos a 0,11 segundos al guardarse en memoria. En este caso, necesitábamos utilizar el RDD para calcular el *score* máximo y el mínimo, reutilizándolo dos veces. Si se dispone de la memoria suficiente, para tareas en las que se necesite reutilizar varias veces, es adecuado como un punto de “*check-point*” en lo que respecta a la eficiencia. No obstante, cabe destacar que, si se necesita demasiada memoria, se podría ocasionar el efecto contrario, con una drástica reducción del rendimiento en el proceso.

PREGUNTA 5.

P5. EL FICHERO SCHIZOFRENIA.FASTA SE HA DESCARGADO DE LA WEB DEL NCBI Y CONTIENE SECUENCIAS DE mRNA ASOCIADAS A LA ESQUIZOFRENIA HUMANA. SIN EMBARGO, EL FORMATO DEL FICHERO NO ES ADECUADO PARA PROCESAR DIRECTAMENTE CON SPARK, PORQUE CADA ELEMENTO OCUPA MÁS DE UNA LÍNEA Y SPARK CONSTRUYE LOS RDDs A PARTIR DE FICHEROS PARTIÉNDOLOS POR LÍNEAS. ES, POR TANTO, NECESARIO UN PASO DE PREPROCESADO PARA PODER UTILIZARLO. EL SCRIPT FASTAREADER.PY LEE ESTE FICHERO Y LO TRANSFORMA EN UNA LISTA DE TUPLAS EN LA QUE CADA TUPLA CONTIENE POR UNA PARTE UN TEXTO QUE IDENTIFICA LA CADENA Y POR OTRA PARTE LA PROPIA CADENA. ESTE RDD TRANSFORMADO DESPUÉS SE ALMACENA EN HDFS. COMPLETA EL SCRIPT PARA QUE HAGA LO DESCRITO.

Se parte del `seqsRDD`, que consiste en un RDD de tuplas (línea, número de línea) en base al contenido del archivo 'schizophrenia.fasta'. Este RDD se modifica para alcanzar `numerosCabeceras`, que es una lista de números de línea correspondiente a cabeceras. Por tanto, se lleva a cabo el desarrollo de una función denotada como *filtro* para proceder a un filtrado del `seqsRDD`, con el fin de seleccionar aquellas líneas que son cabecera, junto con sus correspondientes números. Posteriormente, se hace *map* sobre dicho filtrado, seleccionando el elemento correspondiente a cada número de cabecera y recogiendo la salida como una lista mediante *collect()*. A continuación, se ordena la lista de cabeceras por orden creciente y se comprueba si es correcta:

```
seqsRDD=sc.textFile('/media/sf_PR_ICDAD/Materiales_parte2/schizophrenia.fasta').zipWithIndex()
# Se ha cambiado la función filtro para que devuelva un booleano:
def filtro(s):
    if s[0].startswith(">"): return True
    return False

filtrado=seqsRDD.filter(filtro)

numerosCabeceras=filtrado.map(lambda c: c[1]).collect()

# Ordenamos la lista de cabeceras para tenerlas por orden creciente.
numerosCabeceras.sort()

# Comprobamos que se imprime una lista ordenada con los números.
print(numerosCabeceras)
```

Figura 25. Obtención de `numerosCabeceras` a partir de la manipulación de `seqsRDD`.

Una vez que se ha establecido la lista `numerosCabeceras`, se procede a la manipulación de `seqsRDD` para obtener `groupedRDD`, que representa un RDD de tuplas (número de línea de cabecera, lista de líneas asociadas a esa cabecera). En este caso se emplea la función auxiliar *findLastSmallerEqual* con un *map* y una función lambda aplicada en el número de línea, para comparación con elementos de `numerosCabeceras`, de tal forma que se logra `groupedRDD` correctamente:


```
def findLastSmallerEqual(query, elements):
    """
    Dada una lista ordenada elements, devuelve el mayor de sus elementos que
    aún sea menor que query
    """
    previous=-1
    for e in elements:
        # Si encontramos uno mayor, devolvemos el anterior
        if e>query:
            return previous
        previous=e
    # Si no hemos encontrado ninguno mayor, devolvemos el último
    return previous

groupedRDD=seqsRDD.map(lambda c: (findLastSmallerEqual(c[1],numerosCabeceras),c))
```

Figura 26. Obtención de groupedRDD a partir de la manipulación de seqsRDD.

Para poder realizar la parte final, antes es preciso actuar sobre groupedRDD para agrupar por clave, de modo que se alcancen tuplas (número de línea de cabecera, lista de aquellas líneas asociadas a dicha cabecera). Cabe destacar que para lograrlo, se procede a emplear *groupByKey()* sobre groupedRDD para agrupar los resultados por clave, con un *map* donde se aplica la función *list* sobre la segunda parte de la tupla, pues viene dada como un iterable:

```
grouped_por_cabeceras = groupedRDD.groupByKey().map(lambda x: (x[0], list(x[1])))
```

Figura 27. Obtención de grouped_por_cabeceras a partir de la manipulación de seqsRDD.

El resultado 'grouped_por_cabeceras' logrado en la fase anterior se pasa por la función auxiliar *listToSequenceTuple*, que a partir de una lista de tuplas (cadena, número de línea), proporciona una tupla (cabecera, secuencia) gracias a la selección del elemento correspondiente a la cabecera y la concatenación de secuencias con orden en base al número de línea:

```
def listToSequenceTuple(groupElements):
    """
    Dada una lista de tuplas (cadena, número_línea) groupElements devuelve
    una tupla (cabecera, secuencia) donde ambas son cadenas. La cabecera se
    corresponderá con el elemento de groupElements que empiece por '>' (del
    que habrá uno y solo uno) y secuencia consiste en concatenar el resto de
    cadenas habiéndolas ordenado por su número_línea.
    """
    header=None # Aquí guardaremos la cabecera
    seqParts=[] # Aquí guardaremos los elementos que no son cabecera
    for seqPart,lineNumber in groupElements: # Recorremos para separar cabeceras
        if seqPart[0]=='>':
            header=seqPart
        else:
            seqParts.append((lineNumber,seqPart))
    # Ordenamos los elementos que no son cabeceras atendiendo a su número de línea
    seqParts.sort(key=lambda lineNumber_seqPart: lineNumber_seqPart[0])
    # Juntamos las secuencias ya ordenadas
    sequence="".join(map(lambda lineNumber_seqPart: lineNumber_seqPart[1], seqParts))
    return (header,sequence)

sequencesRDD = grouped_por_cabeceras.map(lambda x: listToSequenceTuple(x[1]))
```

Figura 28. Obtención de sequencesRDD a partir de grouped_por_cabeceras.

Por último, el `sequencesRDD` se almacena en HDFS como fichero de texto, de modo que se indica la correspondiente ruta bajo el formato `hdfs://<servidor>:<puerto><ruta_absoluta_HDFS>`:

```
sequencesRDD.saveAsTextFile("hdfs://localhost:9000/p5/sequencesRDD.txt")
```

Figura 29. Almacenamiento de `sequencesRDD` en HDFS.

PREGUNTA 6.

P6. MODIFICA TU CÓDIGO SPARK PARA UTILIZAR COMO ENTRADA EL RDD QUE PRODUCE `FASTAREADER.PY` EN VEZ DEL OBTENIDO A PARTIR DEL FICHERO `DATASET.TXT`. CARGA EL RDD DESDE HDFS. LA RESPUESTA DEBERÁ CONTENER, ADEMÁS DE LO INDICADO EN EL APARTADO 1, EL TEXTO IDENTIFICATIVO DE LA CADENA.

Para esta última pregunta, se parte del `sequencesRDD` generado anteriormente. Para ello, es preciso este RDD desde HDFS. De este modo, se trata de realizar un procedimiento similar al de la pregunta 1, junto con la novedad de incluir el texto identificativo correspondiente a la cadena:

```
cadena = 'TAATCCGGACAATATTAGTGATATAGGGGGTCGTCTCGTGTGCAAAATCGGGGTCGGGAGTCATAATGGTGTACGGTGTGATTGACCGAACCTCCGCTCTATCCCCCTCCCACA'

rdddataset=sc.textFile("hdfs://localhost:9000/p5/sequencesRDD.txt")

### Función auxiliar obtenida de stackoverflow para transformación eficiente de str a tupla:
def conversion(c):
    c=eval(c)
    return c

rddprocesado=rdddataset.map(lambda w: conversion(w))

tupla=rddprocesado.map(lambda w: (w[0], fitting_alignment(w[1], cadena)))

rddmax=tupla.map(lambda x: x).max(lambda s: s[1][0])

rddmin=tupla.map(lambda x: x).min(lambda s: s[1][0])

### Salida con la información correspondiente:
print("El score más alto se logra con {0} es {1} con cadena 1: {2} \n\n y cadena 2: {3} \n\n El score más bajo con {4} es {5}").format(rddmax[0], rddmax[1], rddmax[2], rddmin[2], rddmin[0], rddmin[1])

sc.stop()
exit()
```

Figura 30. Fragmento del script empleado en la resolución de la p6.

Por tanto, nuestro script planteado presenta en primer lugar la función *fitting_alignment* empleada en la pregunta 1, con la cadena de referencia definida posteriormente. Tal y como se puede apreciar en la Figura 30, se empleó una función auxiliar muy eficiente, obtenida en stackoverflow (<https://stackoverflow.com/questions/8494514/convert-string-to-tuple>), pues la resolución manual de la conversión *string* a tupla conducía a errores. Posteriormente, esta función se emplea para procesar el `rdddataset` correspondiente a `sequencesRDD`, con el fin de lograr las tuplas deseadas y poder acceder correctamente a los diferentes índices en estas. A continuación, se utiliza *map*, para obtener tuplas con el aspecto: (elemento identificativo, (elementos del alineamiento)). Sobre este RDD denominado ‘tupla’, se procede a usar *map* de nuevo, destacando el máximo y mínimo sobre el elemento correspondiente al *score* obtenido con la función *fitting-alignment*.

Por último, se planteó una salida de la información en un formato legible y adecuado, que se refleja a continuación:

```
El score más alto se logra con >NM_004032.2 Homo sapiens D-aspartate oxidase (DDO), transcript variant 2, mRNA es 80 con cadena 1: TCAT--GGAC
-ACAGCAGC-GAT-TGCAGTTGTCGGGGCAGGTGGTGGGGCTCTCCA-CGGCTGT-GTGCATCTCCA-AACTGGTCCCCGATGCTCCG-TT-ACC-ATCATTTACAGACAAGT-TTACTCCAGATACCACCAGTGATGT
GGCAGC--CG-GAATGCTTATTCTCACAC-TT--ATC-CAGAT-ACACC-CATTACACAG-C-A--GAAGCAGTGGTTCAGAG-AAAC--CTTTA-A-TC--AC--CTCTTTGCAATT-G-CCAAT-TCTGCAGAAAGC
TGGAGATGCTG-GTGTTCA-TTTGGTATCAG--GGAT-AA-AGGGA--AGTGGAGGCTGGACA--CT-CACT-CG--GCGA--AT-A-GAAGACCTGTGGGAACCTCATCCGCTCTTTGACATCGTGGTCAAC-TGTTCA
GGCCTTG-GA-AGCAGACAGC-TTGCAGGAGACTCAAGATTTTCCCTGTAAGGGGCCAA--G-TCC-TCCAA-GTTCAGGCTCCCT-GGGTG--GAGCATT-T-TATCCGAGATGGCAG-TGGGCTG-ACATAT-ATTTA
TCCTGTACATCCCATGTAACCTAG

y cadena 2: TAATCCGGACAATATTA-GTGATAT--AG--G--GGGTC--GTCTCGTCTG-TCAAAATCGG-GGTCTG-GGA-GT-CATAA-TGGTG-TACGGTG-T--GATTGACCGA--ACCTC--C--GTCCT
A-TCCCCCTCCCACAAG-GA-G-CGCA-CTTTGCGAAT-C-TA--C-CACGCGTTAGAGCGC-G-TCCGACCACA-TCAC-CGACAACCCGGCGCGAGCCT-AGAGAAAACGGCTCTAGACGCAGGACGACTAATCG-A
ATTAGAAAAATGTC-CCA-TA--T--GA-GC-GAGT-TTAACCTGGTGTAGAACTCTCAACAGAGACTCA-T-CA-TCTCCACATGCTCCAATACGCACCGATGATCATG-AG-GC-GT-GG--CTTCA--C--CCTT
T-ACA-C-TTG-C--CGTGT-A--TTGAGATAGGGGATAGCATAACCGACCCT-AAGGA--TGGCCTAT-A--GACGAATTGATCCTTTAATGTTACAGGTACGTCGGTGCCGAG--TTCTCTATCC--ATGG-
AGCT-GGC-GAACAGATCA-CT-T-CT-GTAAAT-TC-GGT-A-CCT-G

El score más bajo con >AF339782.1 Homo sapiens clone IMAGE:1871856, mRNA sequence es 33 con cadena 3: TAAT---G-TAATTTA-TAGAATAATTCTGGGAT--T
-T-GAGAGGATCTAAACTATTTTCTGTA-T-A-AAT-AT-TA----T--TTG-CCAAAAGTTTGT-TTAT---AT-TCAGAA-G-TCTG-ACTATGATGAA--TA--AATC-TTAAATGCTTTGTTTAATTA-AAAA
ACAAAAATCACAATATCCAAGAC-ATGAAGATATCAGTTC-A-A-CA--A--A-TACT-GTAGTTAAGAGACTAATCTCCACTTGTATGGGAACATTTCACTCTTGGTTTTCAGGA-TAT-AACAG-CACT--TC
ACCGAAATATTTCTTACGCCATAC-CACTGGTAACATTTCTACTAAATC-TTCTGTAACACTTAAAGAATT-CCCTCATTGAT-ACCTTA-CAG-T-GTA-AACAGGAGTCTAA--TTTG--TATCAATAC-TATGTTT
TGGTTGTAATATTC-A-GTTCAC-TC---ACCCAATGT-AC-AACCAAT-GAAAT---AAAAGA--AGCAT-T-TAAA--AGG-A---

y cadena 4: TAATCCGGACAATATTAGT-G-AT-A-TAGGGGGTCTGCTCGTCTG-TG-AAAA-TCGGGGTGCGGAGTCAATAGGTGTACGGTGTGATTGACCGAACCTCCGCTCTATCCCCCTCCCACAAGGAG
C-GCACTTTG-CGAATCTACCACGCGTTAGA-GC-GCGTCGCACCATCACCGACAA-C-CC-GGCGCCGAGCCTA-G-AGAAAACGGCTCTAGACGCAGGACGACTAATCG-AATT-AGA-A--AAATGTCC-C-ATA
T--GAGCGA-GTTTAACT-GTGGTGTT-AGAAGTCTCAACAGAGACTCATCTCCACATGC--TC--CAATACGCAC-CG-ATGA--TC-A-TGAGGCGTGGCT-TCACCCTT-TACACTTGCCGT-GTT-ATTGA-GA
TAGGGGATAGCATAACCGACCTAAGGATGGCCTAT--AGACGAAT-TGATCCTTTAATGTTTCAGGGT-ACGTGCGGTGCCGAGT-TCTCTATCC-ATGGAGCTGGCGAACAGATCA-CTTCTGTAATTCGGTACCTG
```

Figura 31. Resultados alcanzados con el script de la p6.

En conclusión, si atendemos al resultado de la **Figura 31**, se determina que para los elementos con el identificativo correspondiente, el mayor y menor *score* viene dado por 80 y 33 respectivamente, complementándose con las cadenas modificadas.