UNIVERSIDAD COMPLUTENSE DE MADRID

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE MATEMÁTICAS

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS
Y COMPUTACIÓN

LA CONSTELACIÓN HADOOP

JUAN MANUEL GÓMEZ DE PARADA LÓPEZ

Dirigido por:
Luis Llana Díaz

Índice general

Capítulo 1

Introducción

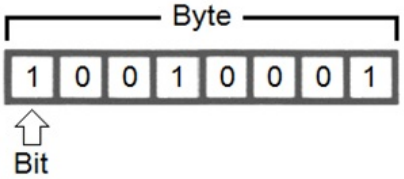
1.1. Comienzos: El Bit

Antes de empezar a hablar de los problemas de procesamiento de grandes cantidades de datos, conviene explicar al lector qué son las grandes cantidades de datos. ¿Cómo se miden los datos?.

Como unidad básica de memoria, tenemos el Bit, la unidad mas pequeña de memoria de la cual podemos hablar. Es una unidad que nos permite expresar un valor booleano: 0 o 1, Verdadero o Falso, Si o No, Blanco o Negro, Abierto o Cerrado... Con bits podríamos escribir en el código binario, sistema numérico usado por los ordenadores para representar información. El código binario solo lo forman unos y ceros, luego para escribir 2 deberemos expresarlo como "10" = $\underline{1}*2 + 0$. Si queremos expresar 5, deberemos escribir "101" = $(\underline{1}*2+\underline{0})*2+\underline{1}$. Y si quisiéramos escribir 6, "110" = $(\underline{1}*2+\underline{1})*2+\underline{0}$.

El reto puede llegar cuando queremos representar algo mas allá de números, pues en los ordenadores podemos hacer más cosas que ver números por pantalla. ¿Cómo representaríamos las letras? Para representarlas, vamos a necesitar una nueva unidad de memoria.

1.2. El Byte



(a) El bit

Figura 1.1: Bit vs Byte
Referencias: http://www.classicaonline.com/musicologia/saggi/15-06-15.html

Un Byte es, básicamente, un conjunto de 8 bits. Si observamos, con un bit se pueden expresar 2 ideas: Abierto y Cerrado, Blanco y Negro, Uno y Cero... Con dos bits podemos expresar 4 ideas: “00”, “01”, “10”, “11”. Con 8 bits podemos expresar hasta 256 posibilidades distintas (2ˆ8).

El Byte es la unidad base de información que se utiliza en las telecomunicaciones y en la computación. Con esta ordenación de 8 bits se desarrolló el código ASCII, código que asocia propiedades a cada uno de los posibles valores de un Byte. Algunos de estos elementos del código ASCII son elementos no imprimibles (bien el retorno de carro, salto de linea, escape, inicio de texto, suprimir...). Otros son elementos imprimibles, tales como “a”, “e”... En la imagen 1.2 podemos ver todos los elementos del código ASCII asignados a las posiciones de la 0 a la 255

Caracteres ASCII de control

00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

ASCII extendido

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Figura 1.2: Código ASCII
Referencias: http://conceptodefinicion.de/ascii/

De este modo, para escribir **“Big Data”**, deberíamos escribir en código binario **34 66 105 103 32 68 97 116 97 34**. Por suerte, este trabajo lo realizan las máquinas. En la actualidad nosotros solo tendremos que pulsar la tecla **a** para que la máquina entienda que le estamos diciendo **97** en código ASCII.

1.3. Cantidades grandes de datos

A partir de este momento, con los Bytes y el Código ASCII en la mano, podemos empezar a escribir cosas mas complejas. Palabras, frases, párrafos, textos... Pero... ¿cuánto ocupa cada cosa?

Cuadro 1.1: Múltiplos de Bytes

Bytes	Unidad	(Corto)	En Bytes	Aproximación
2^{10}	1 KiloByte	(kB)	1024	1000 Bytes
2^{20}	1 MegaByte	(MB)	1024^2	1000^2 Bytes
2^{30}	1 GigaByte	(GB)	1024^3	1000^3 Bytes
2^{40}	1 TeraByte	(TB)	1024^4	1000^4 Bytes
2^{50}	1 PetaByte	(PB)	1024^5	1000^5 Bytes
2^{60}	1 ExaByte	(EB)	1024^6	1000^6 Bytes
2^{70}	1 ZettaByte	(ZB)	1024^7	1000^7 Bytes
2^{80}	1 YottaByte	(YB)	1024^8	1000^8 Bytes

Por ejemplo, los programas que voy a utilizar en mi trabajo (que no llevaría mas de unos minutos escribirlos sabiendo qué hay que escribir) ocupan del orden de 2 kB. "El caballero de la armadura oxidada", una novela infantil tomada de la página web `https://archive.org/stream/librosEDS/caballeroarmaduraoxidada_djvu.txt`, ocupa 100 kB en formato texto plano. El Quijote cabe en 2100 kB en formato texto plano, sacado de la pagina web `https://www.gutenberg.org/cache/epub/2000/pg2000.txt`. Según una transparencia facilitada por mi tutor del Trabajo Fin de Grado, Luis Llana, existen estimaciones de que la Biblioteca del Congreso de los Estados Unidos tiene impresa información en textos y libros del orden de **10 Tb**.

Si intentásemos buscar cuantas veces aparece la palabra "armadura" en el libro de 100 kB, no tardaría mucho (por no decir nada). Podríamos hacerlo con un sencillo programa escrito en Python sin mucha dificultad o tardanza. Si intentamos ver cuantas veces aparece la palabra Kennedy en la Biblioteca del Congreso (una vez convertidos los datos impresos a digitales), la cosa cambia. Seria un proceso que tardaría no solo por tener que revisar 10 Tb de información sino por tener que abrir 147.093.357 archivos como mínimo si asignásemos a cada libro en papel un archivo digital. El número ha sido tomado de Wikipedia `https://es.wikipedia.org/wiki/Biblioteca_del_Congreso_de_Estados_Unidos`.

Más aún, si tratamos de dar acceso a los usuarios de Facebook simultáneamente, tenemos que tener un sistema que nos permita suministrar rápidamente toda la información que precisen sus 1.860 millones de usuarios
`http://www.trecebits.com/2017/02/02/facebook-ya-tiene-1-860-millones-de-usuarios/`.

Otro foco de datos insospechado podrían ser las neveras. Actualmente las neveras están enfriando unas mejor y otras peor. Pero en el futuro compañías como LG o Samsung podrán añadirle a la nevera la capacidad de comunicarse con el usuario `https://www.xataka.com/otros-dispositivos/duelo-de-frigorificos-y-sistemas-operativos-lg-le-pone-windows-10-y-samsung-elige-tizen`, las neveras Liebherr van a poder observar y mandar a nuestro teléfono móvil lo que hay en la nevera, para que lo podamos consultar por ejemplo desde el móvil. Dicho esto, no sería necesario recurrir a las habilidades del Big Data para facilitar el día a día de estas familias. Cuando el usuario lo requiriese, la nevera consultaría los elementos que contiene y lo mandaría al usuario. Ya no sería necesaria esa información y se podría borrar. A esto no podríamos llamarlo "Big Data"

Pero en el mundo en el que vivimos, hay quien ha asociado los datos al petróleo de este siglo `http://www.expansion.com/economia-digital/innovacion/2016/09/20/57e028c6468aeb1e0a8b4620.html`. Las empresas fabricantes de nevera podrán no frenar en su imaginación, y podrían llegar a desarrollar las neveras de tal manera que cuando faltasen determinados productos (que el usuario hubiera comprado frecuentemente), pudiera notificárselo a la empresa correspondiente y esta, a su vez, enviarle una notificación al usuario con algún gancho u oferta para que se lo compre a ellos. Si hablamos de que la empresa que ha comprado ese servicio a la marca de neveras, supongamos,

Neveras de El Corte Ingles , quiere satisfacer las compras de sus clientes, anticipándose para abastecerse de las cantidades justas (tratando de optimizar sus operaciones), y hablamos del volumen de datos de las neveras de toda España, la cosa cambia. Si quisiesen abastecerse para la Campaña de Navidad basado en compras que hizo anteriormente su cliente, la cantidad de datos aumenta considerablemente.

Ahí si podríamos necesitar las herramientas que vamos a ver en este Trabajo Fin de Grado.

Los coches también van a ser capaces de comunicarse tanto con el usuario, como con los talleres, como con los coches que lo rodean por si, por ejemplo, tiene una avería. `http://valencianews.es/motor/bosch-en-el-salon-del-automovil-de-barcelona-el-coche-inteligente-y-conectado/`. Igualmente, en cuanto el coche tenga alguna deficiencia, puede mandar una notificación a consumidor y a taller. En el taller se procesa y se intenta dar hora para llevar el coche al taller. Pero si los talleres de las casas quisieran proveerse de piezas de recambio suficientes (pongamos, para minimizar el tiempo que está un coche en el taller y poder atender a más), necesitaría guardar los datos de cada usuario. Si quisiese guardar la marca esa misma información con una valoración global del coche (pongamos, para ofrecerle al cliente cambiar de coche al finalizar el año, que suele haber rebajas), tendría que guardar más información, hacer estudios sobre ella para decidir cuando es el mejor momento para lanzarle al cliente ese tipo de ofertas. Si ya hablamos de una internacional, por ejemplo Ford, el trafico de datos podría ser ingente.

De nuevo podríamos usar las herramientas que me he propuesto estudiar.

Todos esos datos pronto serán una realidad, y no solo, como podría pensarse en un comienzo, empresas especializadas en Internet (véase Google o Yahoo) van a necesitar manejar grandes cantidades de datos. Esta es una nueva era que está por comenzar

1.4. Tráfico de datos diarios

Según un estudio publicado el 24 de Marzo de 2016, hace apenas un año, Facebook cuenta con mas de 1.860 millones de cuentas
`http://www.trecebits.com/2017/02/02/facebook-ya-tiene-1-860-millones-de-usuarios/`,
Twitter más de 319 millones
`https://www.cnet.com/es/noticias/twitter-319-millones-usuarios-febrero-2017-q4-2016/`,
Google tiene 375 millones
`https://www.multiplicalia.com/redes-sociales-mas-usadas-2017/`.
Los usuarios de Instagram le dan 3.500.000.000 veces a “Me Gusta” al día.
`https://www.brandwatch.com/es/2016/08/96-estadisticas-redes-sociales-2016/`

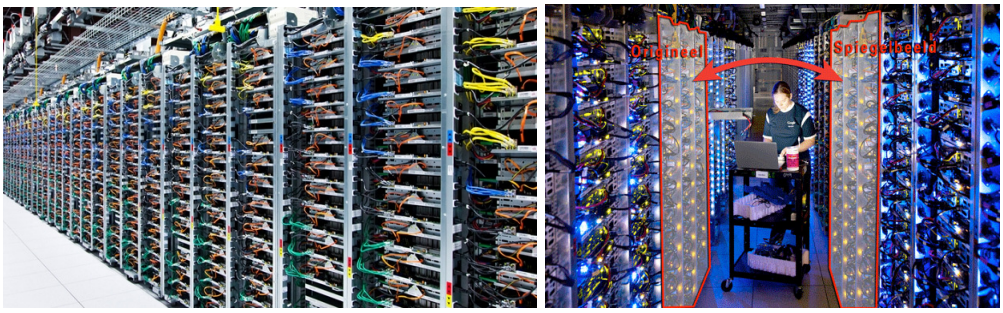
En la barra de buscador de Google se pide información mas de 10.000 millones de veces cada mes, mas de 11.000 veces por segundo se empieza a ver un vídeo en Youtube (referencia link de www.multiplicalia.com)

En cantidad de datos en bytes, se estima que en Internet existían en 2011 5 exabytes de información, 23 exabytes fueron grabados en Internet en el año 2002, y esa cantidad pasó a repetirse cada semana en 2011 `http://www.elmundo.es/elmundo/2011/02/08/navegante/1297179889.html`. Los datos en 2016 eran de velocidad de consumo de internet de 33 Tb/s y 1 763 000 MWh de consumo de electricidad diario `https://es.wikipedia.org/wiki/Internet#Tama.C3.B1o`.

Estas cantidades de datos y de electricidad son ingentes. Resultan difíciles de creer para quien no haya tratado de profundizar en cómo se proporciona los datos que vemos en Facebook, Twitter,

Google... Para ello, diversas empresas cuentan con grandes conjuntos de ordenadores (ubicados frecuentemente en diversos datacenter) que se usan al mismo tiempo, juntos, para proporcionar la información en un tiempo mucho menor que el que usaría una sola maquina. A esta forma de trabajo se le denomina realizar una Programación Paralela, concretamente utilizan Programación Distribuida.

En la siguiente imagen, en ambos lados, podemos ver una parte de un datacenter, un centro de datos usado por empresas como Google, Facebook, Yahoo o Twitter. El de la derecha en cuestión parece estar recibiendo una modificación/reparación/actualización en uno de sus nodos por parte de la persona que aparece en la foto



(a) Clúster (b) Reparación/instalación

Figura 1.3: Granjas de ordenadores
Referencias: http://datacentrum.kostecky.cz/ y
https://dac.com/blog/post/google-datacenter-not-everything-it-seems

Este tipo de máquinas suponen retos en nuestros días. La forma de conectarlos y operar con ellos, la electricidad necesaria para poner a funcionar todos estos ordenadores individuales o la forma de refrigerar estos clusters son los retos de la actualidad. Se han desarrollado varias formas de refrigerar estos ordenadores, entre ellas refrigeración líquida. Sin embargo, en nuestro trabajo vamos a tratar de ver el primer reto.

¿Como podemos conectarlos entre si? ¿Como podemos operar con esta cantidad de datos?

1.5. "Big Data"

Sin embargo, conviene hacerse otra pregunta. ¿Qué es el "Big Data¿ Podriamos estar frente a un cluster con toda la información digital en el mundo y sin embargo no podemos hablar de big data. Entonces, ¿qué es?

En la figura 1.4 podemos observar gráficamente *qué es* eso del Big Data

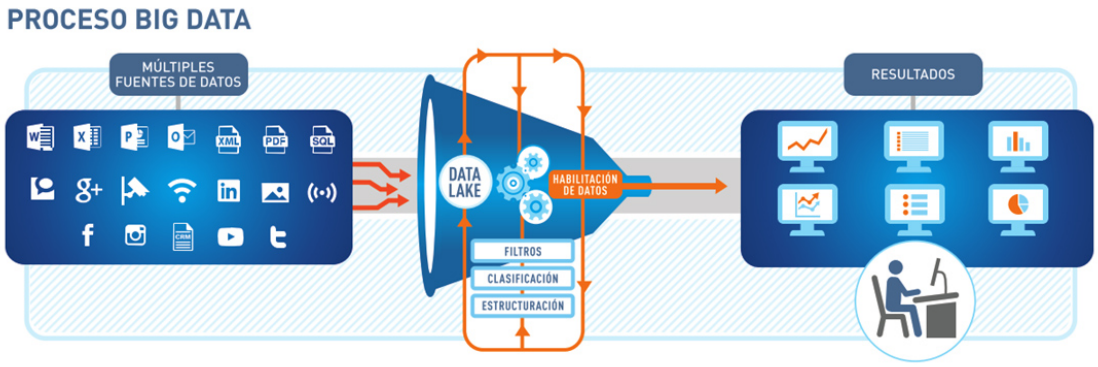


Figura 1.4: Proceso de Big Data
Referenciashttps://www.sonda.com/static/landing/bigdata/img/proceso-big-data.jpg

No consiste en tener una cantidad grande de datos. Eso no tiene ningún valor. Y he usado la palabra clave para definir Big Data. **Valor**. El Big Data consiste en manipular información, que venga de una o varias fuentes, y obtener valor de esa información. Pongo un ejemplo con el caso que nos atañe.

En este trabajo voy a tratar de sacar valor de unos ficheros log determinados. Estos ficheros log representan los procesos que ocurren dentro de una máquina, en mi caso los accesos a la página web de una empresa. Si nos pusiesemos en el caso de que la empresa fuese un banco, y el banco viese que determinadas direcciones IP acceden a la sección de préstamos hipotecarios de su página web, ahi ya tiene un valor los ficheros log de la página web del banco. Si en cambio viese que la proporción de accesos fallidos a la página web (código de éxito diferente de 200), también tendría valor pues se tiene que revisar la página web buscando el enlace que falla para que el cliente se sienta a gusto con su banco.

Si la empresa fuese un concesionario, pasaría igual. Tener los ficheros log por si solos es una pérdida de recursos. Tanto de memoria de la máquina como tiempo de una persona que tratase de sacar algo en claro. Pero utilizando herramientas de minería de datos podrían ver que tipos de coches tienen en stock y mandar una oferta a quién haya estado viendo alguno de los modelos que tienen sin vender.

1.6. Motivación para este trabajo

Tras ver las utilidades de la minería de datos en la asignatura de Programación Paralela, tenía intención de cursar la asignatura que se imparte en la Facultad de Matemáticas de la Universidad Complutense de Madrid, pese a que ya tenía los créditos de la carrera cubiertos. Por incompatibilidades de horario, no pude llegar a cursarla.

Después, en una cena a la que asisto anualmente y que se organiza con cerca de 200 personas de diferentes entornos laborales, recibí un "soplo": ***Estudia Big Data. Es el negocio del futuro.*** *Haz un máster en Big Data.*. ¿Y por qué hacer un máster, cuando lo puedo empezar a estudiar ya en la carrera, cuando ya has empezado a estudiarlo con la asignatura de Programación Paralela?

Pedí consejo a diversos profesores de computación que había tenido a lo largo de la carrera,

para hacer un Trabajo Fin de Grado en Big Data. Fui rebotando de profesor en profesor hasta que llegué a mi actual tutor, el cual me recomendó una serie de programas a estudiar.

Más adelante, tras mucho tiempo negándome, me hice LinkedIn. Allí, de entre los contactos que me sugería la aplicación aparecieron varios de los comensales a aquella cena, de los cuales destacó una persona y su perfil: **Director** y **Data Management & Big Data Consulting at XXX**. Esta persona me hizo otra recomendación: *"Actualmente utilizamos mucho **Hadoop**, y un poco **Spark**. Pero los trabajos nos dicen que se va a usar mucho Spark dentro de poco"*. Casualmente, esta recomendación coincidía bastante con la recomendación de mi tutor.

También me recomendó mi tutor ver el Big Data orientado hacia grandes bases de datos con Hive. Por ello, me recomendó estudiar Hadoop, Spark y Hive.

Capítulo 2

Programación Distribuida y Hadoop

2.1. La necesidad del Big Data

A continuación voy a escribir un texto motivador para el Big Data con el que me crucé mientras investigaba, dado que no soy capaz de expresarlo mejor.

"Nos encontramos en un momento muy emocionante a la hora de trabajar en computación paralela y big data. El gran volumen de datos que hoy en día se genera en todos los campos de la industria y la ciencia está revolucionando la forma como interactuamos con las aplicaciones, creamos productos y estudiamos el mundo a nuestro alrededor. Al mismo tiempo, las herramientas necesarias para trabajar con estos datos se han vuelto mas fáciles de usar que nunca, puesto que los desarrolladores las han hecho accesibles a más y más usuarios, requiriéndoles menos y menos esfuerzos para adoptarlas. Espero que Apache Spark termine siendo una de estas herramientas para ti, que te aporte un nuevo medio para trabajar con datos de manera fácil, potente, e incluso a veces divertida de usar."

Matei Zaharia, CTO en Databricks y Vicepresidente de Apache Spark
Libro en propiedad de la Biblioteca de la Facultad de Ciencias de la Información, UCM [1]

El Big Data consiste en tratar grandes cantidades de datos de una manera eficiente. Consta de tres apartados, tres grandes bloques que no podrían trabajar unos sin los otros. Estos bloques son: **Arquitecto Big Data, Data Science y Visualización de Datos**. El Arquitecto Big Data instala los sistemas y programas para procesar grandes cantidades de datos. Los Data Science se "*pelean*" con los diferentes programas instalados para consumir los datos (que a menudo se les denomina como un pedazo de carbón) y devolver datos útiles (diamantes). Y las tareas propias de la Visualización de Datos trata de devolver los datos útiles en un formato entendible por la mayoría, como por ejemplo con un Diagrama de Barras. En este Trabajo Fin de Grado me voy a centrar en la parte de Arquitecto de Datos y Data Science. Para la parte de Data Science reutilizaré uno de los programas escritos en la asignatura de Programación Paralela del Grado de Matemáticas de la Universidad Complutense de Madrid en el año 2015, mostrado en el Apéndice B. De momento nos bastará con saber el tamaño de los archivos y el tiempo de ejecución.

Cuadro 2.1: Ejemplos Python

Archivo Nasa	Python	Tamaño
super-corto	0.052 s	4 Kb
corto	8.385 s	11 Mb
access-log-Aug95	2 m 11 s	161 Mb
largo	41 m 40 s	5.1 Gb
largo2	1 h 26 m	11 Gb

Como podemos ver, tratar archivos del orden de GigaBytes resulta bastante engorroso, bastante lento. En una era en la que estamos acostumbrados a tratar información en cuestión de minutos, tardar dos minutos para consultar 160 MB puede resultar a menudo demasiado. Vamos a ver varias herramientas (Hadoop y Spark) que nos van a permitir agilizar esos procesos, como ya hemos dicho, utilizando varios ordenadores para procesar esa información de manera distribuida.

2.2. Datos que voy a usar

El archivo con el que trabajaremos en nuestro estudio de Hadoop será los datos de acceso a la pagina web de la NASA de Agosto del año 95. A este tipo de archivos se les denomina ficheros log. Los datos han sido sacados de la página web `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`. Tendremos en total 5 archivos para hacer pruebas, que todos partirán del fichero log de Agosto de 1995. Realizando una simple ordenación y filtrado de las lineas que nos proporciona el archivo, voy a tratar de agrupar las direcciones IP que accedan a los servidores de la NASA, clasificándolos en función de las paginas que visitan en un tiempo determinado, por ejemplo un mismo mes, pero también se podría consultar las páginas que visita una determinada IP cada día. También se podrían consultar las IP que han accedido a una determinada página mas de 4 veces en una semana.

Nuestro objetivo será, partiendo del texto del que anteriormente mostré sus primeras lineas, obtener algo de la forma:

```
IP:                                          "137.204.201.12"
Lista de información general                 [
Lista de direcciones que ha visitado         [
Dirección 1:        "/shuttle/technology/images/sts_body_2-small.gif",
Dirección 2:        "/images/launchpalms-small.gif",
...                          ...
Dirección 16:       "/images/shuttle-patch-small.gif"
Fin de la lista de direcciones               ],
Tiempo maquina del primer acceso de la sesión      807942709.0,
Fin de la lista de información general             1]
```

Los datos vendrán estructurados en este formato:

in24.inetnebr.com - - [01/Aug/1995:00:00:01 -0400] "GET /shuttle/missions/sts-68/news/sts-68-mcc-05.txt HTTP/1.0"200 1839

uplherc.upl.com - - [01/Aug/1995:00:00:07 -0400] "GET / HTTP/1.0"304 0

uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0"304 0

Estos datos están estructurados, separando la información por espacios, los datos:

0 Tenemos una dirección IP, que vendría a ser como la dirección postal de tu ordenador en Internet. En el caso de los datos mostrados, estamos viendo una dirección web.

3 Fecha en la que ha tenido actividad la IP que viene en la posición 0.

6 Dirección web a la que accede la IP (pos 0) en la hora previamente dicha (pos 3)

8 Código de error. Si es código 200, es que se ha obtenido la web (pos 6) sin problemas.

El resto de datos no los necesitaremos.

Estos datos en concreto muestran los accesos de determinadas IP (exitosos o no) a las direcciones web de una misma empresa (en este caso, la NASA) , mostrando la hora de acceso.

Cuadro 2.2: Tamaño de cada línea del archivo

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00	i	n	2	4	.	i	n	e	t	n	e	b	r	.	c
15	o	m		-			[	0	1	/	A	u	g	/	1
30	9	9	5	:	0	0	:	0	0	:	0	1	-	4	0
45	0	]		“	G	E	T		/	s	h	u	t	t	l
60	e	/	m	i	s	s	i	o	n	s	/	s	t	s	-
75	6	8	/	n	e	w	s	/	s	t	s	-	6	8	-
90	m	c	c	-	0	5	.	t	x	t		H	T	T	P
105	/	1	.	0	”		2		0	0		1	8	3	9

En este caso, la primera línea ocupa 130 Bytes. 10 lineas de información vendrían a ser 1 kB aproximadamente. 10.000 lineas serían aproximadamente 1 Mb. El archivo que vamos a usar ocupa un total de 161 MB, con lo cual tiene aproximadamente 1.600.000 lineas. Utilizaremos versiones del mismo archivo de 4 kB(unas 40 líneas), denominado NASA_super_corto, de 11 MB denominado NASA_corto, de 161 MB, de 5 GB denominado NASA_largo y de 11 GB denominado NASA_largo2. En lo sucesivo omitiré la palabra NASA para abreviar el nombre.

2.3. Programación Distribuida

La computación distribuida es una manera de manejar grandes cantidades de datos por medio de un gran número de ordenadores organizados en clusters. Se tratará de una colección de ordenadores separados físicamente, aunque comunicados entre sí bien por cable o por wifi.. Cada máquina posee su propio hardware y software que el programador podrá manejar como si fuese un solo sistema. Vamos a tener los datos separados entre las diferentes máquinas, pero no vamos a necesitar saber qué hay en cada ordenador.

Los sistemas distribuidos deben ser muy confiables, dado que con un gran número de ordenadores es bastante frecuente que falle un ordenador. Bien sobrecalentamiento, bien que llegue al final de su vida útil un disco duro, necesitamos que nuestro sistema distribuido pueda soportar esos fallos sin fallar el sistema y sin perder acceso a la información de la que dispone ese disco duro. Esta característica se la denomina que el sistema sea *tolerante a fallos*.

El tamaño del clúster puede ser muy variado. Puede ser un sistema personal (de unos pocos ordenadores o hosts), un sistema de una ciudad (de cientos o quizás miles de ordenadores) o un sistema mundial, también llamado habitualmente Internet (de millones de hosts). A la propiedad de adaptación ante la aparición de nuevos ordenadores al sistema se le llama *escalabilidad*

Las propiedades que debe de cumplir un sistema distribuido deben de ser:

0 Para cada usuario debe de ser debe de ser similar al trabajo en el sistema centralizado.

1 Cuenta con seguridad interna dentro del sistema distribuido

2 Se ejecuta en múltiples computadoras.

3 Tiene varias copias del mismo sistema operativo o de diferentes sistemas operativos que proveen los mismos servicios

4 Un entorno de trabajo cómodo

5 Dependiente de redes

6 Debe de tener compatibilidad entre todos sus dispositivos conectados.

7 Debe de tener transparencia. No se debe de ver el procesamiento multiple y los accesos remotos.

8 Interacción entre equipos.

9 Diseño de Software compatible con varios usuarios y sistemas operativos

El programa Hadoop y el protocolo de comunicación entre ordenadores SSH nos darán todas esas propiedades. Para instalar Hadoop lo podremos instalar en modo StandAlone (un único nodo, carga de datos desde local), modo Pseudo-Distribuido (un único nodo, carga de datos desde HDFS) y modo distribuido (varios nodos, carga de datos desde HDFS). Para poder encontrar la instalación de Hadoop podemos acudir a la pagina web de Apache Hadoop `http://hadoop.apache.org/docs/stable/`. Adicionalmente está en el Apéndice A. Veamos un poco la historia del programa de Apache.

2.3.1. MapReduce

El algoritmo MapReduce es usado ampliamente en programación paralela. En nuestro caso nos servirá para la programación distribuida. Se compone de dos funciones:

Map encargada de dividir el archivo de entrada en diversos lotes mas pequeños para después procesarlos, cada uno por separado (podemos entender que se procesan cada N lotes en ordenadores separados) y devuelve una tupla clave-valor.

Reduce encargada de recopilar las tuplas clave-valor agrupando los elementos que tengan la misma clave de manera que aparezca la información de la clave junta.

En nuestro ejercicio, que detallo en el Apéndice B, ejecuto primero una función Map (llamada con mucha imaginación tf_mapper), que recoge cada línea del texto y las divide en su información individual, con el comando **line=line.split()**. Después, asigno a cada fragmento de la línea que hemos dividido en valores para manejarlos mejor, como por ejemplo **IP=line[0]**. Así haremos para

FECHA, Archivo y **éxito**. Devuelve una tupla formada por **CLAVE == IP** y **VALOR == (fecha,Archivo)**. Nótese que he extraído "FECHAz devuelve "fecha". Esto se debe a que previamente se aplica una transformación a la fecha para tener lo que podamos utilizar más adelante.

Después ejecuto una función Reduce (llamada, igualmente, tf_reduce) que recoge la información dada por el Map, agrupa las tuplas de misma clave, las ordena y, después, le aplica una condición respecto a la fecha. Si la diferencia entre la primera y la segunda es menor que un tiempo T, añado el segundo archivo a una lista donde ya está el primero. Si el tercero también cumple la condición con el primero, mete el valor en la misma lista. En el momento en el que alguno NO cumpla la condición con respecto al primero, creará una nueva lista donde ya contendrá el archivo nuevo, y se pasará a utilizar el tiempo de ese acceso para ver si los siguientes cumplen la condición de diferir un tiempo T. Al finalizar, devolvemos una tupla clave-valor **CLAVE == (IP, lista de accesos sin repeticiones en una determinada sesión)** y **VALOR == (tiempo original de la sesión en cuestión)**.

Por último, ejecuto un último Reduce. El segundo nos devuelve de nuevo una tupla clave-valor donde podemos reordenarlo. En este caso, si vemos que una misma IP ha accedido a la misma lista de archivos, podremos extraer de ahí un comportamiento repetitivo, dato de interés para las empresas, un *dato con valor*.

2.4. Historia de Hadoop

Cuando hablamos de Hadoop hablamos del software de Programación Distribuida creado por Douglass Read Cutting. Su logo deriva del juguete con forma de elefante que usaba el hijo de este. Fue desarrollado entre 2004 y 2006 a raíz de los documentos publicados por Google en 2004 acerca del procesamiento de información a gran escala para ser trivialmente paralelizada a través de grandes clusters. Su primera versión pública (la 0.1.0) se publicó en 2011 y su actual versión estable (la 2.7.3) salió en Agosto de 2016. Originalmente fue creado para apoyar el proyecto de motor de búsqueda Nutch.

El programa fue adquirido por la empresa Apache y se mantiene como un proyecto de alto nivel para manejar datos del orden de Petabytes. Está escrito en Java y se pueden utilizar con Hadoop programas escritos tanto en Java como en Python.

El programa Apache Hadoop desarrolla software de código abierto para programación distribuida, escalable y confiable. La biblioteca de software de Apache Hadoop es un framework (marco de referencia) que permite la programación distribuida de grandes conjuntos de datos a través de clusters usando modelos de programación sencillos. Está diseñado para funcionar desde en clusters personales y pequeños a cientos de máquinas conectadas con computación y almacenamiento locales. En lugar de confiar en hardware para ofrecer una alta disponibilidad, la biblioteca de Hadoop está diseñada para detectar y operar con los fallos en la aplicación, por lo que ofrece un servicio altamente disponible en un grupo de equipos, cada uno de los cuales puede ser propenso a fallos.

El mayor contribuyente al proyecto Apache Hadoop es la empresa Yahoo!, y es usado además por empresas conocidas por los jóvenes tales como eBay, Facebook, LinkedIn, Tuenti, Twitter, The New York Times, y otras menos conocidas como son AOL, Meebo, Rackspace y Weoh.

2.5. Partes de Hadoop

El programa Hadoop cuenta con tres partes fundamentales:

- Hadoop Common: útiles que soportan los otros módulos de Hadoop
- HDFS (Hadoop Distributed File System): sistema de archivos distribuidos que nos proporciona acceso de alto rendimiento a los datos de la aplicación
- Hadoop YARN: Un marco para la programación de tareas y gestión de recursos en un clúster

2.6. Hadoop Distribuido vs Python

Lo primero, los dos programas están escritos en Python. Recuerdo que estoy tomando de base ejercicios realizados en la asignatura de Programación Paralela en la Facultad de Matemáticas de la UCM. Estoy tratando de hacer una comparativa entre usar el programa Hadoop con su entorno y usar solamente un ordenador y Python. Los programas escritos en Programación Paralela trabajan casi a la perfección en Hadoop, salvo por un detalle que concretaré mas adelante. Para hacer la comparación, voy a precisar nuevamente los datos que utilicé al ejecutar el programa de Python. Antes de continuar, recordemos los datos.

Recuerdo que estoy manejando el fichero log de acceso a la pagina web de la NASA de Agosto de 1995, descargado de `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`, donde el archivo viene en un formato similar al mostrado a continuación:

```
in24.inetnebr.com - - [01/Aug/1995:00:00:01 -0400] "GET
/shuttle/missions/sts-68/news/sts-68-mcc-05.txt
HTTP/1.0" 200 1839
```

En el ejercicio a ejecutar vamos a ver las diferentes IPs que accedieron a la pagina web de la NASA y a agrupar todas las direcciones web que visitó cada IP dentro de la sesión.

Recordamos también el tamaño de los archivos. Por ahorrar espacio, obvio el nombre "NASA":

- "super_corto", tamaño 4 kB (20 filas)
- "corto", tamaño 11 MB (100.000 filas)
- "access_log_Aug95", tamaño 161 MB (1.569.898 filas)
- "largo", tamaño 5.1 GB (51.200.000 filas)
- "largo2", tamaño 11 GB (102.400.000 filas)

Ahora, vamos a ejecutar el programa que anteriormente ejecuté en la terminal de Ubuntu con Hadoop, con un total de 6 ordenadores (el maestro "danaz 5 esclavos) con una memoria RAM de 8 Gb por máquina. Para ejecutar Hadoop, tenemos que ejecutar el comando:

```
python < ejemploHadoop > -r hadoop < ArchivoAEjecutar> donde
EjemploHadoop --> 1ejHadoop.py (Listo en Apendice C.2.)
ArchivoAEjecutar --> hdfs:///user/hadoop/super-corto
```

Voy a añadir al lado el tiempo que se tardaría (supuestamente) ejecutando Hadoop en modo pseudo-distribuido (en lugar de ser 6 ordenadores, solamente uno). Escribo en la tabla el producto del tiempo de Hadoop x6, que vendría a ser un burdo intento de representar lo que tardaría en ejecutarse en Hadoop en modo local, y no distribuido en 6 nodos diferentes.

Archivo Nasa	Python	Hadoop	Hadoop∗6	Tamaño	Ganador
super-corto	0.052 s	34 s	3 m 24 s	4 kB	Python
corto	8.385 s	35 s	3 m 30 s	11 MB	Python
access-log-Aug95	2 m 11 s	51 s	5 m 6 s	161 MB	Hadoop
largo	41 m 40 s	5 m	30 m	5.1 GB	Hadoop
largo2	1 h 26 m	10 m 24 s	1 h 2 m	11 GB	Hadoop

Cuadro 2.3: Ejemplo Python vs Hadoop, cedido para la realización de la clase de Programación Paralela el 25 de Mayo de 2017 de 10:00 a 12:00 por Mario Vargas Gómez

Podemos ver cómo para tamaños de archivos bastante pequeños (menores de 100 MB) no compensa hacer la instalación y posterior ejecución de Hadoop. En cambio, para tamaños de archivos mayores, la diferencia es innegable. Llega a tardar hasta 8 veces menos Hadoop en 6 ordenadores que el ejecutarlo con el programa de Python normal y corriente.

Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
1ejercicio NASA_largo2	MAPREDUCE	default	Thu Apr 6 00:27:18 +0200 2017	Thu Apr 6 00:27:32 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_largo2	MAPREDUCE	default	Thu Apr 6 00:17:08 +0200 2017	Thu Apr 6 00:27:13 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_super_corto	MAPREDUCE	default	Thu Apr 6 00:02:14 +0200 2017	Thu Apr 6 00:02:28 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_super_corto	MAPREDUCE	default	Thu Apr 6 00:01:54 +0200 2017	Thu Apr 6 00:02:09 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_corto	MAPREDUCE	default	Wed Apr 5 23:59:39 +0200 2017	Wed Apr 5 23:59:53 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_corto	MAPREDUCE	default	Wed Apr 5 23:59:18 +0200 2017	Wed Apr 5 23:59:35 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_access_log_Aug95	MAPREDUCE	default	Wed Apr 5 23:57:24 +0200 2017	Wed Apr 5 23:57:41 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_access_log_Aug95	MAPREDUCE	default	Wed Apr 5 23:56:50 +0200 2017	Wed Apr 5 23:57:20 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_largo	MAPREDUCE	default	Wed Apr 5 23:54:45 +0200 2017	Wed Apr 5 23:54:59 +0200 2017	FINISHED	SUCCEEDED
1ejercicio NASA_largo	MAPREDUCE	default	Wed Apr 5 23:49:59 +0200 2017	Wed Apr 5 23:54:40 +0200 2017	FINISHED	SUCCEEDED

Figura 2.1: Ejemplo 1ejHadoop.py ejecutado con Hadoop en cada texto

La imagen 2.1 nos muestra parte de la información que se puede observar en el puerto 8088 del ordenador Dana, una vez ejecutados todos los archivos.

2.7. Problemas encontrados

Al comenzar a instalar Hadoop, traté de instalarlo en tres ordenadores, un maestro y dos esclavos. La vejez de los dos ordenadores esclavos jugó en mi contra.

En el primer ordenador, que podemos denominar de ahora en adelante como "*viejo*", es un ordenador de mas de 10 años de existencia. Eso de por si no debería ser un motivo para tener fallos, pero si los facilita. En este caso, el ordenador *viejo* tenía instalado como sistema operativo Ubuntu 14.04 (como en los demás), y conseguía contactar con *localhost*. Pero uno de los requisitos para ejecutar Hadoop en un clúster es que el maestro se pueda conectar con los esclavos por SSH, y el ordenador *viejo* ni si quiera podía acceder a *localhost* **sin contraseña**, no lo permitía. Al no poder salvar ese problema, pasé a tratar de construir un clúster de dos ordenadores 2.2, formado por los ordenadores que denomino con un arrebato de originalidad *blanco* y *peque*.

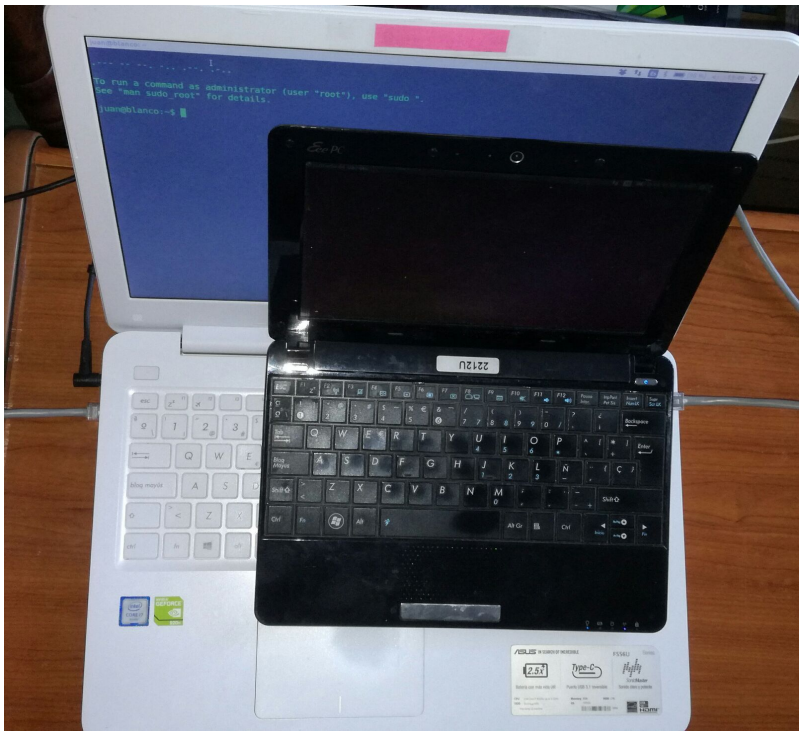


Figura 2.2: Clúster montado con dos ordenadores familiares

La instalación en el ordenador *peque* fue complicada, debido a que es un portátil x32. La instalación en el ordenador *blanco* fue más sencilla. El hecho de no compartir arquitectura fue otra complicación, dado que si compartiesen arquitectura se podría copiar la carpeta contenedora de Hadoop del ordenador usado como Máster, en nuestro caso *blanco* (dado que es en el primero en el que se suele instalar), al ordenador esclavo, en nuestro caso *peque* y tener definidas las direcciones IPs del máster y los esclavos correctamente.

Otra dificultad más fue el hecho de que toda tarea que intentase ejecutar el ordenador *peque* se queda bloqueada. Para mostrar un intento frustrado tengo el presente vídeo:

`https://youtu.be/RXpsclbsmrk`

Otra dificultad encontrada es un ejercicio habitual dentro del entorno de tareas de un Data Science. Habitualmente, los datos no nos vendrán como queremos. Se habrá cambiado la nomenclatura de una base de datos, se habrán desordenado los datos (estropeando nuestros algoritmos) o, entre otros, los archivos vendrán corruptos. Este es el caso que me ha afectado en mi trabajo. Originalmente el archivo NASA_access_log_aug_95 no se leía apropiadamente, ni con Python ni con Hadoop.

Buscando los posibles fallos del archivo descubrí las siguientes líneas que estaban, de una u otra forma, corruptas:

613.146, 613.160(**+1**), 613.187(**+2**), 613.623 (**+3**), 1.019.010 (**+4**), 1.490.989 (**+5**)

Capítulo 3

Spark

3.1. Instalación

Para instalar Spark, nos lo descargamos de la página web de `http://spark.apache.org/downloads.html` de Apache, en donde se nos dará a elegir entre diversas versiones de Spark para diferentes versiones de Hadoop (pues Spark fue diseñado en un principio para apoyarse en el entorno de Hadoop y solventar deficiencias que vieron en el predecesor, no para suplantarlo).

Nada mas descargar Spark de la pagina web, hay que modificar unos pocos parámetros para lograr hacer funcionar el programa en el ordenador. Lo primero que hay que definir es, en el archivo conf/slaves, añadir los ordenadores que van a ser utilizados para ejecutar Spark en paralelo. En este caso, en el archivo tenemos dana, clust-01, clust-02, clust-03, clust-04 y clust-05, cuyas direcciones IPs vienen definidas en el archivo /etc/hosts.

Con esto, Spark ya sabrá que ordenadores usar para ejecutar Spark. Ahora necesitaremos mostrarle a Spark el nodo maestro. Esto se puede definir en el archivo conf/spark-env.sh, añadiendo las líneas

```
SPARK_MASTER_HOST = IP_ordenador_maestro (en este caso la IP de
dana o simplemente "dana")
SPARK_MASTER_WEBUI_PORT=18081
```

Además, habrá que definir dónde está la carpeta de configuración de Hadoop con el parámetro HADOOP_CONF_DIR, que estará en HADOOP_HOME/etc/hadoop/. Más concretamente, en este caso estará en /opt/hadoop/current/etc/hadoop/

Ya estamos listos para ejecutar Spark.

3.2. Ejecución

Los códigos utilizados en esta sección se pueden encontrar en el apéndice C.3.

3.2.1. Arrancado

Para ejecutar Spark, al igual que ocurría con Hadoop, hay que arrancarlo. Para ello, hay dos opciones. O bien

```
/opt/hadoop/current/sbin/start-dfs.sh
/opt/spark/current/./sbin/start-master.sh
/opt/spark/current/./sbin/start-slaves.sh
```

o bien

```
/opt/hadoop/current/sbin/start-dfs.sh
/opt/spark/current/./sbin/start-all.sh
```

3.2.2. Como ejecutar un programa en Spark

Ahora, para ejecutar un programa en Spark, debemos escribir en la terminal

```
$ SPARK_HOME/./bin/spark-submit <EjemploSpark> <ArchivoAEjecutar>
donde:
<EjemploSpark> = 1ejHadoop.py
<ArchivoAEjecutar> = super-corto
```

Obs 1: Si tratamos de ejecutar el archivo 1ejHadoop.py, nos dará **ERROR**. Spark se diseñó pensado solventar los problemas de Hadoop con el Map - Reduce, determinadas páginas web afirman que el Map - Reduce es ineficiente `http://www.baoss.es/apache-spark/`. Spark tiene sus propias herramientas que veremos a continuación. Necesitaremos un archivo e.g. 1ejSpark.py, el cual detallo en el apéndice C.3.

Obs 2: El archivo super-corto NO esta almacenado en LOCAL, sino en el HDFS. Cuando ponemos ./bin/spark-submit 1ejSpark.py super-corto estamos accediendo al archivo hdfs:///user/hadoop/super-corto.

3.2.3. Reducción de datos de procesamiento

Aparte de mostrarnos el mensaje de error, nos mostrará mensajes del tipo

```
17/04/06 10:40:26 INFO SecurityManager: Changing view
acls to: hadoop
17/04/06 10:40:26 INFO SecurityManager: Changing modify
acls to: hadoop
17/04/06 10:40:26 INFO SecurityManager: Changing view
acls groups to:
```

Esta información, aunque útil para descubrir como funciona Spark, resulta inútil para operar con Spark. Puede enmascarar los resultados que tengamos. Podemos cambiar la configuración de Spark para que no nos muestre los mensajes catalogados como ÏNFO"sino que nos muestre aquellos cuyo nivel de información esté al nivel de "WARN.º por encima (.ºFF", "FATAL", .ERROR", "WARN").

Para eliminar las líneas de información .excesivas"que aparecen en Spark al operar con el y que nos podría esconder el resultado, tenemos que copiar el archivo `conf/log4j.properties.template` en `log4j.properties` y editar la línea que pone "`log4j.rootCategory=INFO,console`"por "`log4j.rootCategory=WARN, console`

3.3. Programación en Spark: Tipos de ordenes

A la hora de programar un programa de Python para Spark, debemos añadir lo primero de la librería PySpark los paquetes SparkConf y SparkContext, la cual nos permitirá realizar muchas de las tareas propias de Spark que necesitaremos hacer. La clase SparkContext se instancia automáticamente al utilizar la variable **"sc."**. Los siguientes métodos deberán llevar la siguiente estructura:

1) Nombre de la función o método.

2) Apertura de paréntesis.

3) En caso de que la función o método contenga parámetros:

   a) Nombre de los parámetros obligatorios, separados por comas.

   b) Nombre de los parámetros opcionales entre corchetes, también separados por comas. Si el parámetro opcional tiene un valor por defecto, este se especificará dentro de los corchetes, a continuación del nombre del parámetro y un signo igual (=)

4) Cierre del paréntesis.

3.3.1. RDD

Para leer un archivo del ordenador para procesarlo (cosa que necesitaremos hacer varias veces con Spark), deberemos utilizar el código:

```
>>> miArchivo = sc.textFile(''ruta_al_archivo.txt'')
```

, o sc.textFile(sys.argv[1],1) si queremos que nos sirva para múltiples archivos el mismo ejercicio.

Al hacer esto, hemos creado un Conjunto Resiliente de Datos Distribuidos o RDD (Resilient Distributed Datasets). Spark opera mayoritariamente con estos datos, dado que Spark opera guardando la información en la memoria RAM del ordenador, a diferencia de Hadoop que guarda su información en el HDFS.

Las propiedades de un RDD son:

1 No pueden cambiarse, es decir, son INMUTABLES.

2 Es una colección particionada de registros.

3 Solo se puede construir con transformaciones predefinidas que actúan sobre TODO el conjunto: map, filter, join...

4 NO existe REPLICACIÓN para ofrecer un sistema tolerante a fallos. En su lugar, genera una ruta o bitácora de las transformaciones que han permitido su creación. Se almacena el grafo de las transformaciones que se utilizaron para construir la partición del RDD que se perdió. Si fallara una partición de conjunto, entonces se re-ejecutaría la tarea cuando sea necesario para regenerar la partición perdida.

Hay dos tipos de acciones que se pueden llevar a cabo a un RDD: Acciones y Transformaciones.

Acciones

Sirven para extraer información del RDD de manera que la podamos consultar. Posibles acciones para un RDD pueden ser:

- parallelize(conjunto)
- reduce(función)
- collect()
- collectAsMap()
- count()
- countByKey()
- countByValue()
- first()
- take(n)
- takeSample(conReemplazo,num,[semilla])
- takeOrdered(n,[función de orden])
- foreach(función)
- saveAsTextFile(fichero)
- mean()
- variance()
- stdev()

Transformaciones

Sirven para pasar de un RDD a otro. Como he mencionado, los RDD son inmutables. Esto hace que si queremos modificar un RDD para, por ejemplo, aplicarle un filtro y que solo conserve determinados individuos, esto hará que nos cree un RDD nuevo. Algunas de estas transformaciones son:

- keys()

- values()
- map(función)
- flatMap(función)
- mapValues(función)
- distinct()
- filter(función)
- groupBy(función)
- groupByKey()
- reduceByKey(función)
- sortByKey([ascendente])
- sortBy(función)
- sample(conReemplazo,fraccion,[seed])
- union(otroRDD)
- interseccion(otroRDD)
- zip(otroRDD)
- join(otroRDD)

3.3.2. Optimizador de consultas

A pesar de las instrucciones que nosotros le demos a Spark, éste realizará un proceso de optimización de las acciones necesarias para conseguir el valor deseado. Esto lo descubrí asistiendo a las dos ultimas clases de la asignatura de Programación Paralela del presente año 2017, impartida por el profesor Carlos Gregorio y asistiendo con su permiso como oyente. Un ejemplo de esta optimización la pude observar directamente en esas clases, escribiendo un algoritmo de Spark para realizar una consulta que nos pidió el profesor contra el algoritmo que nos proporcionó el propio profesor.

El ejercicio en cuestión está detallado en el Apendice C subseccion 1. En este ejercicio consumiamos un texto en el que teniamos que ordenar las palabras del texto según su frecuencia de aparición. Una vez separadas todas las palabras y clasificadas como tupla ("palabra", 1), se le denomina como len_2. A esto, se le aplicaba una aplicación más para sumar todos los valores 1 de cada una de las palabras que aparecen, eliminando todas las repeticiones de cada palabra, dejando una con la forma ("palabra", n), donde n es la suma de todos los ünos"de dicha palabra. A ese conjunto de datos nosotros le llamamos len_3. Entonces, tenemos en len_3 un RDD que contiene una lista formada por los pares ("palabras", n).

Una vez teniendo len_3 viene la discrepancia de código entre el profesor y yo. El profesor optó por una solución elegante desde el punto de vista de la programación, escribió el código

```
len_4=len_3.sorBy(lambda x: x[1],False)
```

donde le decimos que ordene el RDD len_3 por el segundo elemento de cada tupla clave-valor que contiene el RDD. El valor booleano "false"le indica al programa que no se ordene de menor a mayor sino de mayor a menor.

Yo opté por una solución más enrevesada:

```
len_5=len_3.map(lambda (x,y)=(y,x))
           .sortByKey(bool(0))
           .map(lambda (x,y):(y,x))
```

Aquí, le digo a Spark que cambie toda la estructura del RDD, que cada clave pase a ser valor y cada valor, clave. Una vez hecho el cambio, le aplicamos la misma ordenación pero al conjunto de las claves, no del primer valor. Y una vez ordenado, deshacíamos el cambio previamente hecho.

Desde un punto de vista de un programador, mi código es mas feo. Se realizan muchas operaciones que, para grandes cantidades de datos, nos lastraría mucho a la hora de obtener el valor deseado. Sin embargo, la experiencia me ha mostrado que el segundo código tarda menos en ejecutarse que el primero. Spark entiende mejor mi código que el que nos proponía el profesor, lo cual resulta sorprendente viendo las tareas que tendría que hacer cada código.

3.3.3. Optimización de los parámetros de Spark

A la hora de realizar mi trabajo, estuve modificando diversos parametros del archivo conf/spark-env.sh, con los que conseguia un menor tiempo de ejecución, como por ejemplo:

SPARK_CLASSPATH: Le definimos la ruta donde se encuentra Spark, por si tratamos de ejecutarlo desde otra carpeta, que el ordenador sepa donde se encuentra Spark.

SPARK_WORKER_CORES

SPARK_WORKER_MEMORY

SPARK_WORKER_INSTANCES

SPARK_EXECUTOR_INSTANCES

SPARK_EXECUTOR_CORES

SPARK_EXECUTOR_MEMORY

A pesar de que los "Worker.están indicados para trabajar en modo Standalone, al modificar esos parámetros para ejecutar Spark distribuidamente se reducían los tiempos de ejecución.

Hay otros muchos parámetros a ejecutar, pero no he tenido necesidad de utilizarlos.

3.3.4. Acerca de los RDD:

No siempre se podrá ejecutar una acción sobre un RDD tal cual se obtiene del disco por varios motivos:

- El RDD podría no contener todos los datos que el programa necesita.

- El RDD podría no estar en un formato adecuado o comprensible o podría tener datos erróneos
- El RDD podría contener más datos de los que se pretenden analizar (por ejemplo, se quiere calcular la altura media de los varones nacidos entre 1975 y 1985, pero se dispone de un RDD que contiene los datos de la población total de hombres y mujeres de todas las edades)
- Los datos que queremos analizar podrían estar dispersos por diferentes RDD.
- Etcétera.

Antes de realizar acciones sobre los RDD, lo habitual es que se deban aplicar **transformaciones** sobre estos, de tal manera que cada RDD contenga todos los datos unificados, filtrados y formateados correctamente para que las acciones que se apliquen sobre él retornen el valor pretendido.

Al aplicar una transformación sobre un RDD original, esta retornará un RDD **nuevo**, resultado de dicha transformación. Esto quiere decir que las transformaciones no modifican el RDD original, y a cada transformación se irá creando un RDD nuevo.

Otro aspecto esencial a tener en cuenta es que Spark evalúa las transformaciones de manera "***perezosa***". Esto quiere decir que aunque se vayan encadenando transformaciones sobre un RDD, Spark no las ejecutará hasta el momento en el que se ejecute una acción sobre el RDD resultante. Esto permite a Spark poder guardar todas las transformaciones de un RDD sin tener que guardar en memoria todos los RDD que se van creando. Pero tiene el inconveniente de que si ejecutamos dos acciones sobre un mismo RDD que es resultado de una transformación, dicha transformación se calculará dos veces. No obstante, los diseñadores de Spark ya han previsto este inconveniente y proveen mecanismos de persistencia para evitar repetir innecesariamente cálculos como se tratará más adelante.

Al igual que con las acciones, habrá transformaciones que son aplicables a cualquier RDD, y otras que solo son aplicables a RDD de clave/valor.

3.4. Programación en Spark

Tenemos el primero de los programas para ejecutar en Spark, que nos proporcionará los archivos a los que ha accedido cada IP en las diferentes sesiones en los apéndices C.3.2 y C.3.3. Veamos ahora la utilidad de Spark. Hasta ahora hemos dicho que guarda la información en memoria RAM y que eso la hace ir más rápido, pero no hemos llegado a probarlo. Para realizar la comparación, voy a copiar la tabla que teníamos anteriormente con la comparación entre Python normal y con Hadoop. Veamos ahora la comparación Spark, Hadoop y Python normal:

Cuadro 3.1: Ejemplos Python Hadoop Spark

Archivo Nasa	Python	Hadoop	Spark	Tamaño	Ganador
super-corto	0.052 s	34 s	7.375 s	4 Kb	Python
corto	8.385 s	35 s	6.401 s	11 Mb	Spark
access-log-Aug95	2 m 11 s	51 s	16.450 s	161 Mb	Spark
largo	41 m 40 s	5 m	3 m 1.51 s	5.1 Gb	Spark
largo2	1 h 26 m	10 m 24 s	7 m 6.41 s	11 Gb	Spark

Como podemos observar, Spark vence a Hadoop en las tareas que nos atañen a nosotros.

Al igual que Hadoop, Spark también dispone de puertos para ver sus propiedades:

```
http://<driver>:4040
```

Puerto del nodo maestro

```
http://<driver>:8080
```

Nota: El puerto 4040 nos proporciona una ventana en el navegador de internet estática y, en cuanto finaliza la tarea, se borra.

3.5. Problemas encontrados

La programación en Spark es casi completamente diferente a como se programaba en la asignatura de programación paralela, por la manera de manipular los RDD. Actualmente todavía hay poca información en Internet, en foros como *stackoverflow*. No es como Python, por ejemplo, que cualquier problema que tengas es resuelta en un plazo muy corto.

Otro problema fue a la hora de imprimir los RDD. No se pueden imprimir directamente. Cuando se intenta imprimir un RDD, salen mensajes que para alguien que no ha programado en Spark resulta incomprensible. Si a ello le añadimos los datos de información que genera spark ejecutandose, los fallos de ejecución quedan dispersos y es mas difícil aún solventarlos.

Otro problema es que al utilizar la memoria RAM de los ordenadores, esta memoria es menor que el tamaño en el Disco Duro. No he llegado a procesar ejercicios que superasen en tamaño la memoria RAM de los ordenadores, con lo cual no he llegado a tener este inconveniente. Sin embargo, si se hubiera superado la memoria RAM del clúster, Spark utilizaría la memoria del Disco Duro para realizar las consultas sobrantes. Ahí se perdería parte de las ventajas que tiene Spark sobre Hadoop. En caso de haber tenido archivos del orden de TeraBytes o PetaBytes, las ventajas hubieran sido inapreciables. Entonces, no se que programa sería mas rápido ejecutandose en la memoria en disco.

Capítulo 4

Hive

4.1. Introducción Hive

El algoritmo "Word Count" está un poco involucrado en la existencia de Hive. Cada vez que se escribe un algoritmo en Hadoop por medio del lenguaje Java, hay más detalles de bajo nivel de los que hay que encargarse. Esa es una tarea sólo al alcance de programadores experimentados de Java, poniendo Hadoop fuera del alcance de usuarios que tratasen de iniciarse en el mundo del Big Data sin haber hecho una carrera especializada, aún incluso cuando los usuarios entendieran el algoritmo.

Muchos de esos detalles de bajo nivel son bastante repetitivos de un trabajo a otro. Aquí es donde entra en juego Hive. No solo provee de un modelo de programación familiar para personas que saben manejar SQL, sino que además elimina mucho código que resulta repetitivo y a veces delicado que debería de hacerse en Java. Es por eso que Hive es tan importante en Hadoop, ya seas un Administrador de Bases de Datos (BDA) o un programador de Java. Hive nos permite completar mucho trabajo con relativamente poco esfuerzo.

Hive esta formado por una interfaz para la línea de comandos (CLI), una interfaz web (HWI), un acceso a través de programas JDBC, ODBC, y un servidor.

Todos los comandos y las consultas van al controlador (driver), que compila la entrada, optimiza los recursos que necesita Hive y ejecuta los pasos requeridos, habitualmente usando procesos MapReduce.

Hive es una buena elección para trabajar con bases de datos, donde el tiempo de respuesta de las consultas y de la modificación por inserción, actualización y borrado no son requeridos. Por supuesto, será una buena elección para aquellos que ya sepan realizar consultas en SQL.

4.1.1. Hive vs Java

Como ejemplo básico comparativo, tenemos el ejercicio de contar palabras. Los detallo al final del texto, en el Anexo C.

En el ejercicio de Java, el programa cuenta con 63 lineas que cada uno tiene que programar.

En el ejercicio de Hive, tan solo cuenta con siete lineas.

4.2. Configuración

Voy a tratar de detallar los archivos a modificar para poder ejecutar Hive en Hadoop. Al igual que Hadoop (se explica al basarse en Hadoop para operar), podemos tener Hive en modo local, modo pseudo-distribuido y modo distribuido.

Para configurarlo en modo distribuido, debemos de modificar el parámetro *hive.metastore.warehouse.dir*, cuyo valor por defecto viene definido como */user/hive/warehouse*

Si deseásemos escribir otro directorio para definir el directorio de la base de datos, debemos escribir el siguiente comando:

```
set hive.metastore.warehouse.dir=/user/juanma/hive/warehouse
```

Esto nos permitirá definir a cada usuario una ubicación diferente de su base de datos. Tendría que hacerlo cada usuario que quisiera tener su base de datos dentro de su directorio personal. Como puede resultar tedioso escribir este comando cada vez que ejecutamos el programa en la línea de comandos, en el libro *Programming Hive* nos recomiendan en la página 36 que añadamos este comando al archivo

```
$HOME/.hiverc
```

4.3. Comandos de inicio de Hive

Para acceder a Hive a través de la línea de comandos, deberemos acceder al ordenador por medio de SSH y teclear en la línea de comandos

```
/opt/hive/current/./bin/hive
```

Nos saldrán muchos mensajes que no entraré en detalle en este trabajo. A continuación, nos disponemos a consultar las variables que tenemos dentro de nuestra instalación con el comando SET

```
hive> set; (Consultar en la terminal)
hive> set env:HOME;
env:HOME=/home/juanma
hive> set -v; (Más datos)
```

Podemos definir una variable nueva para nuestra configuración de Hive:

```
$/opt/hive/current/bin/hive --define foo=bar

hive> set foo;
foo=bar

hive> set hivevar:foo;
hivevar:foo=bar

hive> set hivevar:foo=bar2;

hive> set foo;
foo=bar2
```

```
hive> set hivevar:foo;
hivevar:foo=bar2
```

Podemos incluso incluir las variables de referencia dentro de nuestras consultas

```
hive> create table toss1(i int, ${hivevar:foo} string);
hive> describe toss1;
i           int
bar2        string

hive> create table toss2(i2 int, ${foo} string);
hive> describe toss2;
i2          int
bar2        string

hive> drop table toss1;
hive> drop table toss2;
```

Comentarios en Hive

Podremos añadir comentarios en nuestro programa de Hive añadiendo simplemente - - al comienzo de cada línea

Tipos de datos

Hive soporta muchos de los tipos de datos que nos podremos encontrar fácilmente en bases de datos relacionales

Cuadro 4.1: Tipos de datos primitivos

Tipo	Tamaño	Ejemplo
TINYINT	1 byte	20
SMALLINT	2 bytes	20
INT	4 bytes	20
BIGINT	8 bytes	20
BOOLEAN	Boolean true o false	TRUE
FLOAT	Float de precisión simple	3.14159
DOUBLE	Float de precisión doble	3.14159
STRING	Secuencia de caracteres	“Hola”, ‘Estamos con Hive’
TIMESTAMP(v0.8.0+)	Integer, float o string	Que hora es... ”20:57”
BINARY (v0.8.0+)	Array de Bytes	

Cuadro 4.2: Tipos de datos de conjunto

Tipo	Tamaño	Ejemplo
STRUCT	Se puede acceder a los datos usando una notación por punteros parecida a la usada por C. Por ejemplo, si las columnas tienen una estructura del tipo STRUCTnombre STRING; apellido STRING, se puede acceder (o referenciar) al dato del nombre como .nombre	struct('juan','gómez')
MAP	Una colección de tuplas clave-valor, donde se accede a los datos por medio de arrays (e.g. ['clave']). Por ejemplo, si una columna *nombre* es de tipo MAP con pares clave-valor, 'clave':: "juan", 'valor':: "gómez", entonces el valor se puede consultar escribiendo el nombre ['valor']	map('clave', "juan", 'valor', "gómez")
ARRAY	Secuencia ordenada de un mismo tipo que puede ser referenciada con enteros. Por ejemplo, si tenemos la columna nombre del modo ['juan','gómez'], entonces para referirnos a 'gómez' podremos sacarlo como nombre[1]	array('juan','gomez')

También nos proporciona diversos tipos de funciones SQL basicas, tales como:

- **INSERT INTO SELECT**
- **FROM...JOIN...ON WHERE**
- **GROUP BY**
- **HAVING**
- **ORDER BY**
- **LIMIT**
- **CREATE/ALTER/DROP TABLE/DATABASE**

Esta podría ser una posible declaración de una tabla ficticia destinada a almacenar los datos de los empleados dentro de una empresa:

```
CREATE TABLE empleados (
  nombre STRING,
  sueldo FLOAT,
  subordinados ARRAY <STRING>,
  deducciones MAP <STRING, FLOAT>,
  dirección STRUCT < calle:STRING, ciudad:STRING,
          pais:STRING, C.P.:INT>);
```

4.3.1. Hadoop y HDFS

Podemos realizar operaciones de consulta en el DFS que podíamos realizar en Hadoop

```
hive> dfs -ls /;
Found 5 items
drwxr-xr-x   - hadoop supergroup          0 2016-04-08 14:52 /apps
drwxr-xr-x   - hadoop supergroup          0 2016-06-15 10:51 /home
drwxrwxrwx   - hadoop hadoop              0 2017-02-07 10:13 /tmp
drwxr-xr-x   - hadoop supergroup          0 2017-03-27 11:41 /user
drwxr-xr-x   - hadoop supergroup          0 2016-04-07 13:55 /usr

hive> dfs -ls /user/juanma;
Found 9 items
drwxr-xr-x   - juanma supergroup    0 2017-05-02 14:32 /user/juanma/
                                                          .hiveJars
drwxr-xr-x   - juanma supergroup    0 2017-04-29 19:34 /user/juanma/
                                                 20170429193451_salida
...
```

4.4. Arrancando Hive

Ahora, voy a mostrar los comandos usados para acceder a Hive. En nuestro caso, Hive está descargado en la carpeta /opt/hive/

```
$ /opt/hive/current/bin/hive
> show databases;
OK
default
ejemplosimple
pokemonbd
Time taken: 0.205 seconds, Fetched: 3 row(s)
```

Previamente ya hemos definido las bases de datos *pokemonbd* y *ejemplosimple*. Aporto diversas instrucciones de utilidad para operar con Hive:

```
hive> CREATE DATABASE IF NOT EXISTs pokemonbd2;
hive> SHOW DATABASES;
hive> SHOW DATABASES LIKE 'pokemon*';
```

Podemos cambiar la localización de las bases de datos si quisiésemos:

```
hive> CREATE DATABASE IF NOT EXISTS pokemonbd3
      LOCATION
      'file:///home/juan/paraPruebas/baseDeDatos';
hive> CREATE DATABASE IF NOT EXISTS POKEMONBD4
      COMMENT 'Estoy haciendo pruebas';
hive> CREATE DATABASE pokemonbd5
      WITH DBPROPERTIES ('creator'='Juan Manuel',
      'date'='2017-05-14');

hive> DESCRIBE DATABASE pokemonbd3;
OK
pokemonbd3        NULL::character varying
file:/home/juanma/paraPruebas/baseDeDatos
```

```
juanma  USER
Time taken: 0.03 seconds, Fetched: 1 row(s)

hive> DESCRIBE DATABASE pokemonbd4;
OK
pokemonbd4      Estoy haciendo pruebas
hdfs://dana:9000/user/hive/pokemonbd4.db
juanma  USER
Time taken: 0.025 seconds, Fetched: 1 row(s)

hive> DESCRIBE DATABASE pokemonbd5;
OK
pokemonbd5      NULL::character varying
hdfs://dana:9000/user/hive/pokemonbd5.db
juanma  USER
Time taken: 0.026 seconds, Fetched: 1 row(s)

hive> DROP DATABASE IF EXISTS pokemonbd2;

hive> DESCRIBE DATABASE EXTENDED pokemonbd5;
OK
pokemonbd5      NULL::character varying
hdfs://dana:9000/user/hive/pokemonbd5.db
juanma  USER    {date=2017-05-14, creator=Juan Manuel}
Time taken: 0.031 seconds, Fetched: 1 row(s)

hive> ALTER DATABASE pokemonbd5 SET DBPROPERTIES
                        ('edited-by'='Gómez de Parada');
OK
Time taken: 0.217 seconds

hive> describe database extended pokemonbd5;
OK
pokemonbd5      NULL::character varying
hdfs://dana:9000/user/hive/pokemonbd5.db
juanma  USER    {edited-by=Gómez de Parada,
date=2017-05-14, creator=Juan Manuel}
Time taken: 0.028 seconds, Fetched: 1 row(s)
```

NOTA: No se podrá cambiar o borrar una propiedad de las bases de datos.

Al crear una tabla, al definir cada columna de la tabla podemos añadir un comentario para cada una de las columnas añadiendo "COMMAND « comando » ". A continuación de la definición de la tabla.

Podemos **copiar** el esquema de una tabla con el comando "LIKE"

```
hive> CREATE DATABASE IF NOT EXISTS pokemonbd6
      LIKE pokemonbd4;
```

Para usar una base de datos en concreto se utiliza el comando **USE** y para mostrar las tablas que contiene, **SHOW TABLES**

```
hive> USE pokemonbd;
```

```
OK
Time taken: 0.058 seconds

hive> SHOW TABLES;
OK
pokemons
u_data
Time taken: 0.069 seconds, Fetched: 2 row(s)
```

Para mostrar los elementos de la tabla *pokemons*, se usan instrucciones propias de SQL. Para mostrar los nombres y propiedades de cada columna, se utiliza el comando **DESCRIBE**.

```
hive> DESCRIBE pokemons;
OK
numero                  double
nombre                  string
tipo                    string
resistencia             double
ataque                  double
defensa                 double
Time taken: 0.104 seconds, Fetched: 6 row(s)

> SELECT * FROM pokemons;
OK
1.0     Bulbasaur       Planta-Veneno     90.0   118.0   118.0
2.0     Ivysaur         Planta-Veneno     120.0  151.0   151.0
3.0     Venusaur        Planta-Veneno     160.0  198.0   198.0
4.0     Charmander      Fuego             78.0   116.0   96.0
...
...
...
248.0   Tyranitar       Roca-Siniestro    200.0  251.0   212.0
249.0   Lugia           Psíquico-Volador  212.0  193.0   323.0
250.0   HoOh            Fuego-Volador     212.0  263.0   301.0
251.0   Celebi          Psíquico-Planta   200.0  210.0   210.0
```

4.5. Operando con Hive: Ejemplo

Para el siguiente ejemplo vamos a seguir el que sale en la pagina web `https://www.adictosaltrabajo.com/tutoriales/hive-first-steps/#05`. En el, nos proponen que descarguemos los datos de los presupuestos municipales del Ayuntamiento de Madrid para el año 2015 (para nosotros 2017), y nos proponen que ejecutemos una interfaz de Hive en el navegador web. Voy a tratar de hacer los mismos casos a través del CLI.

Definimos la tabla en la que vamos a volcar los datos

```
hive> CREATE TABLE IF NOT EXISTS presupuestos
    > (
    > centro string,descripcion_centro string,
    > seccion string,
    > descripcion_seccion string,programa int,
    > descripcion_programa string,
    > capitulo int, descripcion_capitulo string,
    > economico int, descripcion_economico string,
    > importe string
    > )
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\073';

hive> show tables;
OK
presupuestos
Time taken: 0.044 seconds, Fetched: 1 row(s)

hive> describe presupuestos;
OK
centro                  string
descripcion_centro      string
seccion                 string
descripcion_seccion     string
programa                int
descripcion_programa    string
capitulo                int
descripcion_capitulo    string
economico               int
descripcion_economico   string
importe                 string
Time taken: 0.346 seconds, Fetched: 11 row(s)
```

Hemos conseguido crear la tabla. En el ejemplo en la web nos indica "FIELDS TERMINATED BY ';"', pero ';' nos da error al ejecutarlo.

Añadimos los datos de "gastos" a la tabla:

```
hive> LOAD DATA INPATH '/user/juanma/gastos.csv'
    > OVERWRITE INTO TABLE presupuestos;
Loading data to table presupuestos_madrid_2017.presupuestos
OK
Time taken: 1.373 seconds
```

Y ahora, mostramos los datos de la tabla. Por hacerlo algo mas legible, he decidido mostrar solo 3 de sus lineas

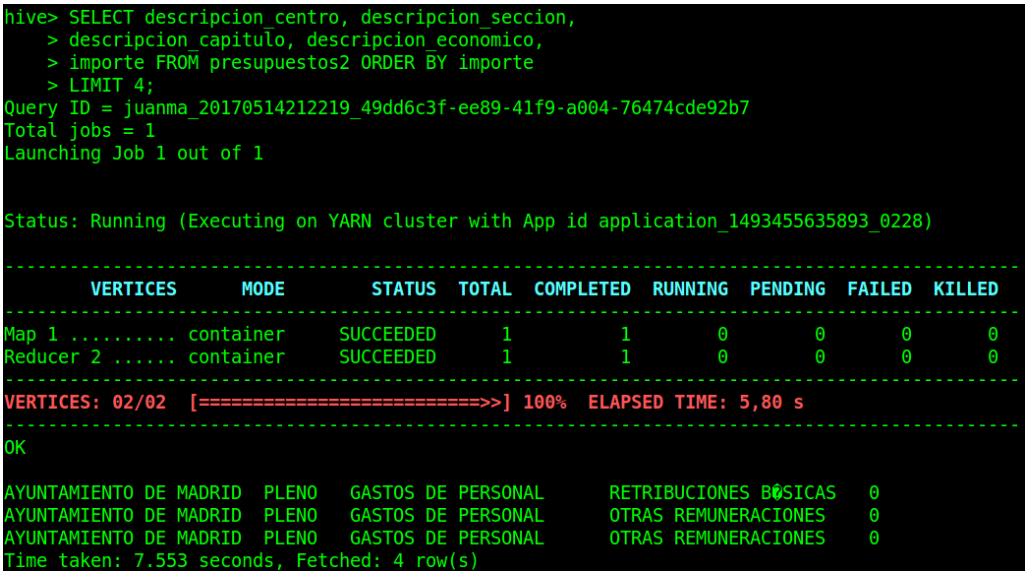


Figura 4.1: Ejecución de Hive para los Gastos del Ayuntamiento de Madrid

Si queremos observar los gastos de una sección en concreto, basta con aplicar un comando del tipo:

```
hive> SELECT descripcion_centro, descripcion_seccion,
    > descripcion_capitulo, descripcion_economico,
    > importe FROM presupuestos2
    > WHERE descripcion_seccion='PLENO'
    > LIMIT 5;
OK
AYTO. DE MADRID PLENO    GASTOS PERSONAL
            RETRIBUCIONES BÁSICAS          29946

AYTO. DE MADRID PLENO    GASTOS PERSONAL
            RETRIBUCIONES COMPLEMENTARIAS          95574

AYTO. DE MADRID PLENO    GASTOS PERSONAL
            SUELDOS DEL GRUPO C2 18632

AYTO. DE MADRID PLENO    GASTOS PERSONAL
            TRIENIOS      5107

Time taken: 0.346 seconds, Fetched: 4 row(s)
```

Capítulo 5

Conclusiones

Volviendo a la tabla anterior en la que comparaba las bondades de Hadoop y Spark, podemos observar que para archivos cortos (del orden de unos pocos megabytes) es contraproducente instalar los programas propios de Big Data tales como Hadoop o Spark.

Para archivos más grandes, llegando a la categoría de gigabytes, hay poca diferencia entre Hadoop y Spark. Pero de entre los dos podemos observar una ventaja de Spark frente a Hadoop, que se debe a la carga de datos en memoria RAM. En los casos que he ejecutado no llego a saturar la memoria RAM del clúster al que he tenido acceso. Los archivos no han superado los 11 Gb mientras que la memoria RAM de las máquinas son 8 Gb/máquina $*6$ máquinas = 48 Gb. Para comprobar hasta donde llega la rapidez de Spark, debemos de probar con archivos de ese tamaño aproximado. Para comprobar la eficiencia de Spark frente a Hadoop, debemos ejecutar archivos del orden de 100 Gb, ejecutándolos por partes. Si con un archivo de 60 Gb ya vemos discrepancias de tendencia en la diferencia de ambos programas, podremos decidir cual será mejor con archivos más grandes.

Teniendo en cuenta que Spark se ejecuta mayoritariamente en memoria RAM y que los programas de Big Data están pensados para manejar cantidades de datos del orden de petabytes (aproximadamente 10^6 Gb), si viesemos que se reduce en algún momento la diferencia entre ambos programas, Hadoop será el candidato a ejecutar programas del estilo de los mios. Si viésemos que la diferencia se estabiliza (pues aumenta a medida que aumenta el archivo), entonces podremos decir que nos conviene usar Spark, pues nos proporcionaría una ventaja especializada.

Cuadro 5.1: Ejemplos Python Hadoop Spark

Archivo Nasa	Python	Hadoop	Spark	Tamaño	Ganador
super-corto	0.052 s	34 s	7.375 s	4 Kb	Python
corto	8.385 s	35 s	6.401 s	11 Mb	Spark
access-log-Aug95	2 m 11 s	51 s	16.450 s	161 Mb	Spark
largo	41 m 40 s	5 m	3 m 1.51 s	5.1 Gb	Spark
largo2	1 h 26 m	10 m 24 s	7 m 6.41 s	11 Gb	Spark

Con Hive hemos podido ver que las bondades de la programación distribuida también se aplican a las bases de datos, proporcionandonos una forma de programarlos bastante sencilla. Hive es una base de datos apropiada para realizar análisis de datos, pero no para ser una base de datos dinámica, viva. Esto es así por no permitirse la eliminación de ninguna fila en las bases de datos creadas con

Hive.

Dado que Hive se apoya en Hadoop para funcionar, conviene destacar que con Spark también se pueden manejar bases de datos. Como la formación para lograr dominar esa tarea excedía el tiempo de dedicación de este Trabajo Fin de Grado, no he profundizado en ello, pero así como existe Hive para Hadoop, un programa llamado HBase está relacionado con Spark, manejando bases de datos NoSQL, y cuya ejecución no es tan sencilla como en Hive. HBase nos proporcionaría una base de datos dinámica.

Sin embargo, aunque pudiésemos declarar un vencedor en cuanto a Hadoop o Spark, no podríamos declararlo el rey del Big Data. Tenemos una larga serie de programas utilizados para el análisis de datos y las bases de datos. Por ejemplo, los que se estudian en el Máster en Business Inteligence & Big Data del centro MBitSchool:

Data Architect & Data Science

Pentaho, desde 2004[1]

Hadoop, desde 2004 - 2006 [2]

Cloudera, desde 2008 [3]

Flume, desde 2009 [4]

Hive, posterior a Hadoop [5]

Neo4j, desde 2007 [6]

Cassandra, desde 2008 [7]

Mongo, desde 2009 [8]

HBase, desde 2006 [9]

Apara, desde 2013 [10]

Spark, desde 2014 [11]

Visualización de datos

QLikView, desarrollado por QLik fundada en 1993 [12]

Tableau, desde 2003 [13]

Power BI, desde 2013 [14]

[1] http://www.pentaho.com/blog/12-years-self-service-business-analytics-pentaho
[2] https://en.wikipedia.org/wiki/Apache_Hadoop
[3] https://es.wikipedia.org/wiki/Cloudera
[4] Professional Hadoop, libro disponible parcialmente en Google Play
[5] https://en.wikipedia.org/wiki/Apache_Hive
[6] https://en.wikipedia.org/wiki/Neo4j
[7] https://es.wikipedia.org/wiki/Apache_Cassandra
[8] https://en.wikipedia.org/wiki/MongoDB
[9] https://es.wikipedia.org/wiki/Apache_HBase
[10] http://www.apara.es/en/news-predictive-analytics/contenido/17-2013/
[11] https://en.wikipedia.org/wiki/Apache_Spark
[12] https://en.wikipedia.org/wiki/Qlik
[13] https://es.wikipedia.org/wiki/Tableau_Software
[14] https://en.wikipedia.org/wiki/Power_BI

Como podemos ver, es un mundo bastante activo, bastante vivo. Se crea cada muy poco tiempo un programa que trata de reemplazar al anterior. Una persona que quiera trabajar en el mundo del Big Data tiene que tener en cuenta que es un mundo de continua formación.

Apéndice A

Instalación de Hadoop

Vamos a ver los pasos que hay que seguir para instalar los diferentes componentes. Todo ello puede encontrarse (en inglés) en la página web `http://hadoop.apache.org/docs/stable/`

A.1. Requisitos:

A.1.1. Versión de Java:

Para empezar, lo que hay que hacer es ser conocedor de la versión de Java que tenemos instalado en el ordenador en cuestión. En el ordenador que lo probé tenía la versión java1.7.0-openjdk-amd64. En caso de desconocer la versión de Java instalada en el ordenador, haciendo

```
ls -l /usr/lib/jvm
```

en la terminal de Ubuntu nos dirá que versiones de Java están instaladas en el ordenador en la carpeta en cuestión. En la página web `https://wiki.apache.org/hadoop/HadoopJavaVersions` podemos encontrar ayuda acerca de qué versiones de java soporta cada versión de Hadoop.

A.1.2. SSH:

SSH es un intérprete de ordenes seguro. Permite manejar por completo (con los suficientes permisos) una computadora desde otra diferente. A nosotros nos permitirá que se comuniquen entre si los ordenadores de nuestro clúster. También nos servirá más adelante para acceder remotamente a los ordenadores facilitados por mi tutor, Luis Llana.

Fuente: `https://es.wikipedia.org/wiki/Secure_Shell`

Lo que tenemos que hacer es tener instalado el ssh y el sshd. Para ello, desde la página de Apache Hadoop nos sugieren los comandos

- sudo apt-get install ssh
- sudo apt-get install rsync

Modificando el orden de instalación que aparece en la página web de Apache, voy a tratar de conectar mi ordenador a *localhost*. Acceder a localhost se utiliza para probar la viabilidad de la conexión. Al tratar de acceder a localhost en realidad estamos intentando llamar a la dirección IP 127.0.0.1, que está dentro de cada ordenador, reservada. Al final, conectar con localhost es como conectar una máquina consigo misma para hacer pruebas. Si consigo entrar, ya podré empezar a plantearme entrar desde otros ordenadores. Hasta que no consiga entrar en Localhost, no recomiendo intentar dar el siguiente paso. Este es uno de los pasos en donde fracasé y desistí de instalar Hadoop en el ordenador "viejo", reduciendo mi clúster de 3 ordenadores a 2.

Para acceder a localhost, basta con escribir en la CLI de Ubuntu

```
$ ssh localhost
```

Nos debería devolver algo parecido a esto

```
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-59-generic x86\_64)
Documentation:  https://help.ubuntu.com/
Last login: Mon Mar 27 10:59:20 2017
.--- -- --. -.. .--. .-..
To run a command as administrator (user "root"), use "sudo ".
See "man sudo\_root" for details.
```

lo cual nos indicará que hemos accedido a nuestra propia terminal de nuevo. Si al hacer el comando $ ssh localhost no necesitamos ningún tipo de contraseña, entonces podremos saltarnos el siguiente paso. En caso contrario:

Que hacer si la conexión al localhost por SSH nos pide contraseña:

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

Volviendo a probar el comando "ssh localhost"nos debería dar la bienvenida esperada.

A.2. Hadoop en Local

Al entrar en la página `http://hadoop.apache.org/docs/stable/index.html`, encontramos una recomendación. Instalar primero Hadoop en nodo individual, para que veamos como instalar Hadoop paso a paso y, que una vez lo tengamos instalado, pasemos a la instalación de Hadoop en un clúster de diferentes nodos. Con Hadoop instalado en un ordenador individual ya podremos ejecutar ejercicios de MapReduce en Local, para ver si funciona. De manera que allá vamos.

La pagina `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html` nos proporcionará la mayoría de comandos necesarios, salvo aquellos tan básicos que se asume el instalador conoce.

Después de esperar a que termine de hacer las instalaciones, la web nos propone descargar una versión estable de Hadoop desde la pagina web `http://www.apache.org/dyn/closer.cgi/hadoop/common/`. Descomprimimos la versión de Hadoop descargada en la carpeta destino de nuestra elección. Optamos por descomprimirlo en /opt/. Con la descarga y la descompresión básicamente quedaría instalado Hadoop. Ahora solamente habría que configurarlo.

Primero tenemos que configurar el archivo /opt/hadoop-2.7.X/etc/hadoop/hadoop-env.sh, donde en la línea 25 del archivo podemos encontrar el fragmento de código (A.1) a modificar donde

podremos cambiar $JAVA_HOME por /usr/lib/jvm/java-1.7.0-openjdk-amd64 (version de Java mencionada anteriormente) como en la imagen (A.2)

```
# Set Hadoop-specific environment variables here.

# The only required environment variable is JAVA_HOME.  All others are
# optional.  When running a distributed configuration it is best to
# set JAVA_HOME in this file, so that it is correctly defined on
# remote nodes.

# The java implementation to use.
export JAVA_HOME=${JAVA_HOME}
```

Figura A.1: $HADOOP_HOME/etc/hadoop_env.sh original

```
# Set Hadoop-specific environment variables here.

# The only required environment variable is JAVA_HOME.  All others are
# optional.  When running a distributed configuration it is best to
# set JAVA_HOME in this file, so that it is correctly defined on
# remote nodes.

# The java implementation to use.
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64/
```

Figura A.2: $HADOOP_HOME/etc/hadoop_env.sh modificado

También podremos configurar la variable JAVA_HOME para que apunte a la versión de Java que queremos usar. A continuación nos piden que ejecutemos en terminal el comando bin/hadoop (A.3), lo cual nos devuelve posibilidades para ejecutar Hadoop en nuestro ordenador.

```
juan@blanco:/opt/HADOOP/hadoop-2.7.3$ bin/hadoop
Usage: hadoop [--config confdir] [COMMAND | CLASSNAME]
  CLASSNAME            run the class named CLASSNAME
 or
  where COMMAND is one of:
  fs                   run a generic filesystem user client
  version              print the version
  jar <jar>            run a jar file
                       note: please use "yarn jar" to launch
                             YARN applications, not this command.
  checknative [-a|-h]  check native hadoop and compression libraries availability
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
  classpath            prints the class path needed to get the
  credential           interact with credential providers
                       Hadoop jar and the required libraries
  daemonlog            get/set the log level for each daemon
  trace                view and modify Hadoop tracing settings

Most commands print help when invoked w/o parameters.
```

Figura A.3: $HADOOP_HOMEbinhadoop

Posteriormente utilizaremos en gran medida el comando fs, que nos dará acceso al Sistema de Ficheros de Hadoop.

Ahora nos propone tres modos de instalación con Hadoop:

- Modo local (o autónomo)
- Modo pseudo-distribuido
- Modo distribuido

A.2.1. Modo Local

Por defecto, Hadoop viene configurado para ejecutarlo en modo local, como un simple proceso de Java. Esto es útil para procesos de depuración.

Para configurar el modo autónomo, tenemos que ejecutar una serie de comandos en la terminal de Ubuntu. Situándonos en la carpeta hadoop–2.7.X/, tenemos que ejecutar

$ mkdir input

$ cp etc/hadoop/∗.xml input

$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar grep input output ´[a-z.]+ ´

$ cat output/∗

A.3. Modo Pseudo-Distribuido

Podemos ejecutar Hadoop en el modo pseudo-distribuido en un único ordenador. Esto nos va a permitir hacer una primera configuración de Hadoop para comprobar que funciona y, una vez hecha esa comprobación, pasaremos a ver la instalación de Hadoop completamente distribuida.

A.3.1. Archivos a configurar

Lo primero que hay que hacer es modificar dos archivos:

El archivo HADOOP_HOME:$etc/hadoop/core-site.xml añadiendo

```
<configuration>
  <property>
       <name>fs.defaultFS</name>
       <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

.

En este archivo definimos el puerto del ordenador en el que configurar HDFS

El archivo HADOOP_HOME:$etc/hadoop/hdfs-site.xml añadiendo

```
<configuration>
  <property>
       <name>dfs.replication</name>
       <value>1</value>
  </property>
</configuration>
```

.

En este archivo definimos el número de copias de los archivos entre los ordenadores del futuro clúster. En caso de tener muchos ordenadores y querer tener un sistema distribuido tolerante a fallos, conviene que el número de copias de seguridad no sea menor a 3.

Ejecución:

1 Formateo del sistema de ficheros: $ bin/hdfs namenode -format

2 Arrancar las herramientas del datanode y del namenode $ sbin/start-dfs.sh

(opcional) Podemos mirar la interfaz web del Namenode en el puerto del ordenador 50070: http://localhost:50070/ (adjunto imagen del puerto 50070 del ordenador "dana" A.4 facilitado por mi tutor del Trabajo Fin de Grado)

3 Crear los directorios en el HDFS necesarios para ejecutar Hadoop:

- bin/hdfs dfs -mkdir /user

- bin/hdfs dfs -mkdir /user/"username" (a lo largo del trabajo utilizaré usuarios como juan, chusky, hadoop, hive, spark...)

4 Copiar archivos de local a HDFS: $bin/hdfs dfs -put etc/hadoop input

5 Ejecutar ejercicio propuesto: $bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar grep input output 'dfs[a-z.]+'

6 Copiar los resultados de HDFS a local y consultarlos:

- bin/hdfs dfs -get output output

- cat output/∗

(opcional) Tambien podemos consultarlo desde el HDFS: $bin/hdfs dfs -cat output/*

7 Parar los datanode y el namenode $sbin/stop-dfs.sh

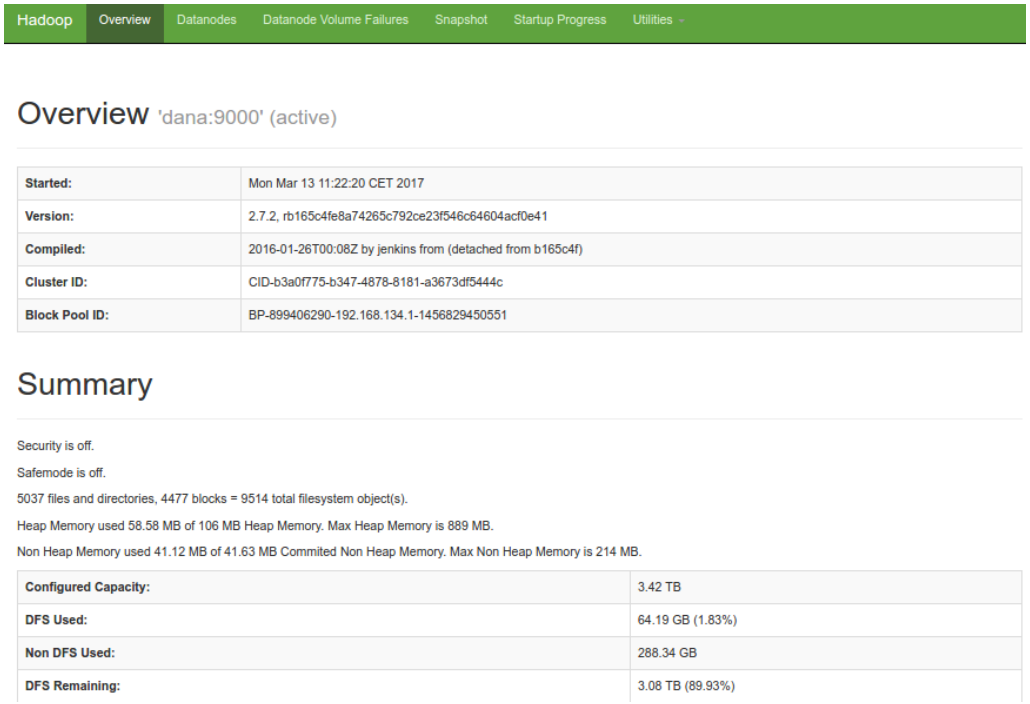


Figura A.4: Puerto 50070 Ordenadores Cluster Dana

A.3.2. YARN:

La función que tiene el Yarn es dividir las tareas del gestor de recursos (ResourceManager) y monitorizar dichas tareas por separado. El ResourceManager y el NodeManager son los responsables de las tareas hechas. El NodeManager se encarga de gestionar los paquetes de datos de cada ordenador, controlando el estado de la CPU, la memoria, el espacio en disco y la red y enviar esa información al ResourceManager.

Después tenemos la Aplicación Maestra, encargada de negociar los recursos con el ResourceManager y trabajar con los NodeManager para ejecutar y controlar las tareas.

Configuración del YARN:

Para configurar el Yarn, hay que modificar dos archivos que tenemos tras descomprimir Hadoop en el ordenador.

El archivo etc/hadoop/mapred-site.xml añadiendo

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

.

El archivo etc/hadoop/yarn-site.xml añadiendo

```
<configuration>
  <property>
      <name>yarn.nodemanager.aux-service</name>
      <value>mapreduce_suffle</value>
  </property>
</configuration>
```

. Una vez realizadas esas tareas, ya estará listo el ordenador para arrancar el Yarn

0 Arrancar el dfs: sbin/start-dfs.sh

1 Arrancar el Yarn: sbin/start-yarn.sh

(opcional) Consultar el puerto 8088 para ver la información que proporciona el ResourceManager

2 Ejecutar las tareas que nos propongamos

3 Parar el Yarn: sbin/stop-yarn.sh

(bis) 3 Parar el dfs: sbin/stop-all.sh

Después del paso 2, ya podremos ejecutar programas de Python (y de Java) de manera que podamos ver la ejecución de la tarea en el puerto 8080. La interfaz que veremos será estática. Para ver la evolución del programa, deberemos actualizar la ventana.

A.4. Modo Distribuido

A.4.1. Archivos a configurar

Para instalar Hadoop en el clúster, deberemos copiar los archivos en cada ordenador (asegurándonos de tener la misma versión de Java) y modificar los siguientes archivos

1 etc/hadoop/core-site.xml

```
<configuration>
  <property>
      <name>fs.defaultFS</name>
      <value>hdfs://IPmaestro:9000</value>
  </property>
</configuration>
```

.

2 etc/hadoop/hdfs-site.xml

```
<configuration>
  <property>
      <name>dfs.namenode.name.dir</name>
```

```
        <value>/srv/hdfs/namenode</value>
        <!--Debemos crear ese directorio primero-->
    </property>
    <property>
        <name>dfs.blocksize</name>
        <value>2097152</value>
        <!-- Debe de ser potencia de 2 -->
    </property>
    <property>
        <name>dfs.datanode.name.dir</name>
        <value>/srv/hdfs/datanode</value>
        <!--Debemos crear ese directorio primero-->
    </property>
    <!-- </property>
        <name>dfs.hosts/dfs.hosts.exclude</name>
        <value>Lista de nodos de datos permitidos
        /excluidos. No lo hemos necesitado en
        nuestro trabajo, pero es previsible que
        resultará util cuando manejas clusters
        de muchos nodos </value>
        </property> -->
</configuration>
```

.

3 etc/hadoop/yarn-site.xml

```
<property>
    <name>yarn.acl.enable</name>
    <value>true</value>
  </property>

  <property>
    <name>yarn.admin.acl</name>
    <value>Admin ACL</value>
  </property>

  <property>
    <name>yarn.log.aggregation-enable</name>
    <value>false</value>
  </property>

  <property>
    <name>yarn.resourcemanager.hostmane</name>
    <value>IPmaestro</value>
    <!-- Le indicamos a Hadoop donde está el
    ResourceManager, en este caso en el nodo
    maestro -->
  </property>

  <property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>/opt/HADOOP/hadoop-2.7.3/yarn/data
    </value>
  </property>

  <property>
    <name>yarn.nodemanager.logs-dirs</name>
    <value>/opt/HADOOP/hadoop-2.7.3/yarn/logs
    </value>
  </property>

  <!-- </property>
      <name>dfs.hosts/dfs.hosts.exclude</name>
      <value>Lista de nodos de datos permitidos/
   excluidos. No lo hemos necesitado en nuestro
   trabajo, pero es previsible que resultará
   util cuando manejas clusters de muchos
   nodos </value>
      </property> -->
</configuration>
```

.

4 etc/hadoop/mapred-site.xml. Habrá que añadir

```
<property>
  <name>mapred.job.tracker</name>
  <value>IPmaestro:54311</value>
  <description> El ordenador y el puerto donde los
  trabajos de seguimiento MapReduce se realizarán.
  Si se realizan en local, entonces los trabajos
  se ejecutaran como un simple map y un reduce.
  </description>
</property>
```

A.4.2. Puertos de Hadoop

Para consultar el estado de los trabajos realizados y de las herramientas de Hadoop, podemos visitar los puertos **50070** (para consultar el estado de los namenodes), **8088** (para consultar el estado del ResourceManager) y **19888** (para consultar el histórico de trabajos de Hadoop)

Ya estamos listos para ejecutar tareas en Hadoop.

Apéndice B

Códigos de Python

El primer ejercicio fue realizado en la asignatura de Programación Paralela de la Facultad de Matemáticas de la Universidad Complutense de Madrid.

B.1. 1ejercicio.py

```
# -*- coding: utf-8 -*-
from mrjob.job import MRJob
from mrjob.step import MRStep
import time


class MRTrabajo(MRJob):
    SORT_VALUES = True

    def tf_mapper(self, _, line):
        line = line.split()
        if len(line)>=2:
            IP=line[0]
            FECHA=line[3]
            Archivo=line[6]
            Exito=line[8]
            if (Exito=='200'):
                fecha= time.mktime(time.strptime(FECHA,
                                    '[%d/%b/%Y:
                                    %H:%M:%S']))
                yield IP,(fecha,Archivo)
```

```
    def tf_reducer(self,IP,values):
        T=30*24*60*60
        hora=0
        i=0
        horas=[]
        archivos=[]
        for fecha,archivo in values:
            if (fecha-hora>=0 and fecha-hora<T):
                archivos[i-1].append(archivo)
            else:
                hora=fecha
                horas.append([hora])
                i=i+1
                archivos.append([archivo])

        for v in range(len(archivos)):
            yield (IP,list(set(archivos[v]))),
                                        (horas[v][0],i)
    def reducido(self,key,values):
        hora=0
        T=385000
        for horas,n in values:
            if horas-hora>T:
                hora=horas
                yield key[0],(key[1],horas,n)
    def steps(self):
        return [
            MRStep(mapper = self.tf_mapper,
                   reducer = self.tf_reducer),
            MRStep(reducer = self.reducido)
        ]


if __name__ == '__main__':
    tiempoInicio=time.time()
    MRTrabajo.run()
    tiempoFinal=time.time()-tiempoInicio
    print (str(tiempoFinal))
```

B.2. 1ejHadoop.py

Se trata de una versión de nuestro ejercicio 1ejercicio.py adaptada a Hadoop. La adaptación es mínima, pero merece ser descrita. Nótese que la impresión de tiempo en el programa de Hadoop no la he ejecutado. Esto es debido a que son incompatibles la función time.time() con Hadoop, lo cual no es demasiado importante porque Hadoop ya se encarga de contar el tiempo de ejecución de los archivos ejecutados en el YARN.

También cabe destacar que podemos definir el nombre con el que aparece reflejada la ejecución de la tarea en el puerto 8088, entre otros valores configurables.

```
# -*- coding: utf-8 -*-

from mrjob.job import MRJob
from mrjob.step import MRStep
import time
```

```
class MRTrabajo(MRJob):
    SORT_VALUES = True
    def tf_mapper(self, _, line):
        line = line.decode("utf-8").split()
        if len(line)>=2:
            IP=line[0]
            FECHA=line[3]
            Archivo=line[6]
            Exito=line[8]
        if (Exito=='200'):
            fecha= time.mktime(time.strptime
                        (FECHA,'[%d/%b/%Y:%H:%M:%S'))
            yield IP,(fecha,Archivo)


    def tf_reducer(self,IP,values):
        T=30*24*60*60
        hora=0
        i=0
        horas=[]
        archivos=[]
        for fecha,archivo in values:
            if (fecha-hora>=0 and fecha-hora<T):
               archivos[i-1].append(archivo)
            else:
                hora=fecha
                horas.append([hora])
                i=i+1
                archivos.append([archivo])

        for v in range(len(archivos)):
            yield (IP,list(set(archivos[v]))),(horas[v][0],i)
    def reducido(self,key,values):
        hora=0
        T=385000
        for horas,n in values:
            if horas-hora>T:
                hora=horas
                yield key[0],(key[1],horas,n)
    def steps(self):
        return [
            MRStep(mapper = self.tf_mapper,
                   reducer = self.tf_reducer),
            MRStep(reducer = self.reducido)
        ]


if __name__ == '__main__':
    MRTrabajo.JOBCONF = {
         #"mapreduce.job.reduces": 10,
         #"mapreduce.task.io.sort.mb":1200,
         #"mapreduce.map.memory.mb":3000,
         #"mapreduce.map.java.opts":"-Xmx1900M -XX:+UseSerialGC",
         #"mapreduce.reduce.memory.mb":3000,
         #"mapreduce.reduce.java.opts":"-Xmx1800M -XX:+UseSerialGC",
         #"mapreduce.task.timeout":0,

         "mapreduce.job.name":"1ejhadoop.py"
    }

    MRTrabajo.run()
```

Apéndice C

Funciones para los RDD

Aquí mostraré las funciones básicas y mas comunes que se pueden utilizar con Spark, tanto acciones como transformaciones

C.1. Acciones:

parallelize(conjunto)

Paraleliza en memoria una lista

reduce(func)

Agrega los elementos de un RDD según la función que se le pase como parámetro. Dicha función ha de tener dos parámetros y debe retornar un valor. Además, esta función debe de cumplir las propiedades conmutativa y asociativa, para que pueda ser correctamente computada en paralelo. El parámetro `func` puede ser el nombre de una función previamente creada o una función *lambda*. Por ejemplo:

```
>>> rdd = sc.parallelize([4, 1, 2, 6, 1, 5, 3, 3, 2, 4])
>>> suma = rdd.reduce(lambda x,y: x+y)
>>> print ("Los elementos de 'rdd' suman en total = %d" %suma)
Los elementos de 'rdd' suman un total = 31
```

collect()

Nos devuelve todos los elementos de un RDD como una lista de Python. Por ejemplo:

```
>>> lista = rdd.collect()
>>> print ("El tercer elemento de la lista es %d" %lista[2])
El tercer elemento de la lista es 2
```

collectAsMap()

(Key-Value) Nos devuelve los elementos de un RDD clave/valor como un diccionario de Python. Por ejemplo:

```
>>> sc.parallelize([('a','b'),('c','d')])
   .collectAsMap()
{'a':'b', 'c':'d'}
```

count()

Nos devuelve el numero de elementos del RDD. Por ejemplo:

```
>>> print ("El RDD contiene %d elementos" %rdd.count())
El RDD contiene 10 elementos
```

countByKey()

Nos devuelve un diccionario donde las claves son las diferentes claves que el RDD contiene , y los valores son el número de veces que cada clave aparece. Por ejemplo:

```
>>> cv = sc.parallelize([("Juan", 185), ("Pedro",
170), ("Eva", 175), ("Juan", 178)])
>>> cv.countByKey()
{'Juan':2, 'Pedro':1, 'Eva':1}
```

countByValue()

Nos devuelve un diccionario con el número de apariciones de cada elemento en un RDD. Por ejemplo:

```
>>> rdd.countByValue()
{1: 2, 2: 2, 3: 2, 4: 2, 5: 1, 6: 1}
```

first()

Nos devuelve el primer elemento del RDD. Por ejemplo:

```
>>> rdd.first()
4
```

take(n)

Nos devuelve los n primeros elementos en forma de lista. Por ejemplo:

```
>>> rdd.take(4)
[4, 1, 2, 6]
```

takeSample(conReemplazo,num,[semilla])

Nos devuelve una lista con una muestra aleatoria de los elementos contenidos en el RDD. El parámetro *num* indica cuántos elementos contendrá nuestra muestra; *conReemplazo* (True o False) indica si la muestra es con reemplazo o sin reemplazo. El parámetro opcional *semilla* inicializa el generador de números aleatorios. Por ejemplo:

takeOrdered(n,[funcOrden])

Nos devuelve los n primeros elementos del RDD, ya sea según su orden natural o según el orden que se le pase como una función en el parámetro *funcOrden*. Por ejemplo:

```
>>> rdd = sc.parallelize([4, 1, 2, 6, 1, 5, 3, 3, 2,
4])
>>> rdd.takeOrdered(5)
[1, 1, 2, 2, 3]

>>> rdd.takeOrdered(5,lambda x: -x)
[6, 5, 4, 4, 3]
```

foreach(func)

Ejecuta la función que se le pasa por parámetro sobre cada elemento del RDD. Por ejemplo:

```
def impar(x):
  if x %2 == 1:
    print("%d es impar" %x)
>>> rdd.foreach(impar)
1 es impar
5 es impar
3 es impar
3 es impar
1 es impar
```

saveAsTextFile(fichero)

Nos guarda el RDD como un conjunto de archivos de texto dentro del *fichero* que le digamos.

mean(), variance(), stdev()

Nos devuelven, respectivamente, la media, la varianza y la desviación típica estándar de los valores del RDD sobre el que se llaman.

C.2. Transformaciones:

keys()

(Key-Value) Crea un nuevo RDD que únicamente contiene las claves del RDD original.

values()

(key-value) Crea un nuevo RDD que únicamente contiene los valores del RDD original.

```
>>> kv = sc.parallelize([('a', 1), ('b', 2),
                         ('c', 3)])
>>> kv.keys().collect()
['a', 'b', 'c']
>>> kv.values().collect()
[1, 2, 3]

[('a', 4), ('b', 6)]
```

map(func)

Nos devuelve un nuevo RDD resultado de pasar cada uno de los elementos del RDD original como parámetro de la función *func*.
Por ejemplo:

```
>>> rdd = sc.parallelize([4, 0, 2, 6, 1, 5, 3, 9,
7, 8])
>>> t1 = rdd.map(lambda x: x*2)
>>> t1.collect()

[8, 0, 4, 12, 2, 10, 6, 18, 14, 16]

>>> claves = t1.map(lambda x: (x, chr(ord('a')+x)))
>>> claves.collect()

[(8, 'i'), (0, 'a'), (4, 'e'), (12, 'm'), (2, 'c'),
(10, 'k'), (6, 'g'), (18, 's'), (14, 'o'), (16, 'q')]
```

flatMap(func)

Es similar a **map(func)**, pero en caso de que *func* retorne una lista o tupla, esta no será mapeada directamente en el RDD resultante, sino que se mapearán individualmente los elementos contenidos. Por ejemplo:

```
>>> lineas= sc.parallelize(["En un lugar de la
    mancha", "de cuyo nombre no quiero
    acordarme", "no ha mucho tiempo que vivía",
    "un hidalgo de los de lanza en astillero",
```

```
    "adarga antigua, rocín flaco y galgo corredor"])
>>> palabras=lineas.flatMap(lambda lin: lin.split())
>>> palabras.collect()

['En', 'un', 'lugar', 'de', 'la', 'mancha', 'de',
'cuyo', 'nombre', 'no', 'quiero', 'acordarme', 'no',
'ha', 'mucho', 'tiempo', 'que', 'vivía', 'un',
'hidalgo', 'de', 'los', 'de', 'lanza', 'en', 'astillero',
'adarga', 'antigua', 'rocín', 'flaco', 'y',
'galgo', 'corredor']
```

Hay que tener cuidado en el empleo de la función flatMap, pues no nos dará una lista de palabras, sino una lista cuyos elementos son otras listas

mapValues(func)

(key-value) Crea un nuevo RDD de pares clave-valor, resultado de aplicar únicamente sobre los valores la función *func*, que recibe un solo parámetro.

```
>>> kv = sc.parallelize([("a", 1), ("b", 2),
                         ("c", 3)])
>>> kv.mapValues(lambda x: x*2).collect()

[('a', 2), ('b', 4), ('c', 6)]
```

distinct()

Nos devuelve un nuevo RDD que contiene una sola copia de los diferentes elementos del RDD original. Por ejemplo:

```
>>> num = sc.parallelize([1, 2, 3, 4, 4, 3, 2, 5])
>>> num.distinct().collect()

[4, 1, 5, 2, 3]
```

filter(func)

Nos devuelve un nuevo RDD que solo contiene los elementos del RDD original que satisfacen el predicado especificado en la función *func* (que retornará *True* o *False*). Por ejemplo:

```
>>> num = sc.parallelize([1, 2, 3, 4, 5, 6, 100,
2000, 4000])
>>> menor50 = num.filter(lambda x: x < 50)
                 .collect()

[1, 2, 3, 4, 5, 6]
```

```
>>> palabrasMAL=lineas.flatMap(lambda lin: lin.split())
>>> palabrasMAL.collect()

[['En', 'un', 'lugar', 'de', 'la', 'mancha'], ['de',
'cuyo', 'nombre', 'no', 'quiero', 'acordarme'], ['no',
'ha', 'mucho', 'tiempo', 'que', 'vivía'],
['un', 'hidalgo', 'de', 'los', 'de', 'lanza', 'en',
'astillero'],    ['adarga', 'antigua', 'rocín', 'flaco',
'y', 'galgo', 'corredor']]
```

groupBy(func)

Nos devuelve un nuevo RDD de pares clave/valor que agrupa los elementos del RDD original según el resultado de evaluarlos a través de la función *func*. Los valores son agrupados como un objeto de la clase *Iterable*, que declara los métodos necesarios para que Spark pueda acceder secuencialmente a los objetos de una colección de datos. Por ejemplo:

```
>>> ciudades= sc.parallelize(["Albacete', 'Almería',
    'Badajoz', 'Barcelona', 'Cáceres'])
>>> (ciudades.groupBy(lambda nom: nom[0])
             .map(lambda c: (c[0], list (c[1])))
             .collect())
[('A', ['Albacete', 'Almería']), ('B', ['Badajoz',
'Barcelona']), ('C',['Cáceres])]
```

groupByKey()

(key-value) Crea un nuevo RDD de pares clave-valor donde las claves guardan cada una de las diferentes claves del RDD original, y los valores asociados se agrupan en un objeto iterable con los valores que estaban asociados a la misma clave en el RDD original.

reduceByKey(func)

(key-value) Crea un nuevo RDD de pares clave-valor, donde cada clave única se corresponde con las diferentes claves del RDD original, y el valor es el resultado de aplicar una operación *reduce* sobre los valores correspondientes a una misma clave.

```
>>> r = sc.parallelize([("a", 1), ("b", 2),
                        ("a", 3), ("b", 4)])
>>> r.reduceByKey(lambda x,y: x+y).collect()

[('a', 4), ('b', 6)]
```

sortByKey([ascendente])

(key-value) Crea un nuevo RDD resultado de la ordenación natural de las claves del RDD original. El parámetro booleano *ascendente* indica si se deben ordenar en orden ascendente *True*

o descendente *False*

sortBy(func)

Ordena un RDD según el criterio especificado en la función *func*. Por ejemplo:

```
>>> nums = sc.parallelize([2, 1, 4, 3])
>>> nums.sortBy(lambda x: x).collect()
[1, 2, 3, 4]
>>> nums.sortBy(lambda x: -x).collect()
[4, 3, 2, 1]
```

sample(conReemplazo, fracción, [seed])

Su funcionamiento es similar a la acción takeSample anteriormente explicada, con la diferencia de que la transformación sample retorna un RDD, y el parámetro *fracción* no indica el número de elementos que muestrear, sino la fracción del RDD sobre el que se aplica. Por ejemplo, para *fracción=0.25* se devolvería un RDD cuyo tamaño es una cuarta parte del RDD original. Para *fracción=2* se devolvería un RDD cuyo tamaño es el doble del RDD original, siempre que *conReemplazo=True*, o el mismo tamaño que del RDD original si *conReemplazo=False*

union(otroRDD)

Nos devuelve un nuevo RDD que contiene la unión de los elementos del RDD original y del que se pasa como argumento. Por ejemplo:

```
>>> ciud1 = sc.parallelize(["Barcelona", "Madrid",
"Paris"])
>>> ciud2 = sc.parallelize(["Madrid", "Londres",
"Roma"])
>>> ciud1.union(ciud2).collect()

['Barcelona', 'Madrid', 'Paris', 'Madrid', 'Londres',
'Roma']
```

interseccion(otroRDD)

Nos devuelve un nuevo RDD que contiene la intersección de los elementos del RDD original y del que se pasa como argumento. Por ejemplo:

```
>>> ciud1 = sc.parallelize(["Barcelona", "Madrid",
"Paris"])
>>> ciud2 = sc.parallelize(["Madrid", "Londres",
"Roma"])
>>> ciud1.interseccion(ciud2).collect()

['Madrid']
```

zip(otroRDD)

Crea un RRD de pares clave/valor donde las claves son cada uno de los elementos del RDD original y los valores son los correspondientes elementos, según orden, del RDD pasado como parámetro. Por ejemplo:

```
>>> a = sc.parallelize(["a", "b", "c"])
>>> b = sc.parallelize([1,2,3])
>>> a.zip(b).collect()

[('a', 1), ('b', 2), ('c', 3)]
```

join(otroRRD)

(key-value) Crea un nuevo RDD que agrupa las claves en común en el RRD original y el RRD que pasamos como parámetro, cuyo valor es una tupla con los valores de ambos RDD asociados a una misma clave. Cuando hay más de una coincidencia, se añaden varios pares clave-valor. Por ejemplo:

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a",2), ("a",3)])
>>> x.join(y).collect()

[('a', (1, 2)), ('a', (1, 3))]
```

C.3. Códigos de Spark

C.3.1. Ejemplo

Este ejemplo fue escrito íntegro en una clase impartida la última semana de curso en la asignatura Programación Paralela de 2017

```
from pyspark import SparkContext
from operator import add

import sys
import time


sc = SparkContext()
sc.setLogLevel("ERROR")
data=sc.textFile(sys.argv[1],1)

len_lines_rdd = data.flatMap(lambda x: x.split())
print(len_lines_rdd.take(10))

len_2=len_lines_rdd.map(lambda x: (x,1))
print(len_2.take(10))
len_3=len_2.reduceByKey(lambda x,y: x+y)
```

```
tiempoInicio1=time.time()
len_4=len_3.sortBy(lambda x: x[1],False)
tiempoFinal1=time.time()-tiempoInicio1

tiempoInicio2=time.time()
len_5=len_3.map(lambda (x,y): (y,x)).sortByKey(bool(0)).map(lambda (x,y):(y,x))
tiempoFinal2=time.time()-tiempoInicio2

print('RESULTS----------------------------')
print('word frequency',len_4.take(10), 'tiempo SortBy', str(tiempoFinal1))
print('word frequency',len_5.take(10), 'tiempo SortByKey', str(tiempoFinal2))
```

C.3.2. 1ejSpark.py

```
from __future__ import print_function

import time
import datetime
import sys
from operator import add

from pyspark import SparkConf, SparkContext

def fecha_de_sesion(datos):
    IP=datos[0]
    FECHA=datos[3]
    archivo = datos[6]
    if (len(datos)>2):
        fecha= time.mktime(time.
              strptime(FECHA,'[%d/%b/%Y:%H:%M:%S'))
        return (IP,(fecha,archivo))



def accesos_en_sesion(IP,valores):
    T=24*60*60 #Defino 1d como el tiempo entre sesiones
            para separar comportamientos de los usuarios
    hora=0
    i=0
    horas = []
    archivos = []
    for (fecha, archivo) in valores:
        if (fecha-hora>=0 and fecha-hora<T):
            archivos[i-1].append(archivo)
        else:
            hora=fecha
            horas.append(fecha)
            i=i+1
            archivos.append([archivo])
    for v in range(len(archivos)):
```

```
        return (IP),(horas[v],
                     list(set(archivos[v])),v)



if __name__=="__main__":
    tiempoInicio=time.time()
    if len(sys.argv)!=2:
        print("Usage:wordcount <file>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="Contar IPs")
    sc.setLogLevel("ERROR")
    ips = sc.textFile(sys.argv[1],1)
            .map(lambda x: x.split(' '))

    ipsGood = ips.filter(lambda x: x[7]==('200'))
                 .sortBy(lambda x: x[3])
                 .map(lambda x: (fecha_de_sesion(x)))
                 .groupByKey()

    ipsGoods = ipsGood.map(lambda x:
                           accesos_en_sesion(x[0],x[1]))

    print ("c1")
    ipS=ipsGoods.collect()
    ipsAgrupadas=ipsGoods.groupByKey().count()
    print ("c2")


    # x.split(' ')[0] := IP
    # x.split(' ')[3] := FECHA en formato
    #                    [03/Aug/1995:22:49:15
    # x.split(' ')[6] := WEB que solicita
    #                    /images/MOSAIC-logosmall.gif"
    # x.split(' ')[7] := Exito o no
    ts =  datetime.datetime
                 .fromtimestamp(time.time())
                 .strftime("%Y %m %d %H %M %S")
    #ipsGoods.saveAsTextFile(ts+"_salida")
    #for x in ipS:
         #print (":"+str(x)+":")
         #print (":"+str(x[0])+":-->: "
                    +str(len(x[1][1]))+":")

    #print (":"+str(ipsAgrupadas)+":")
    #print (":" + str(ipsGoods.count())+":")
    tiempoFinal=time.time()-tiempoInicio
    hor=(int(tiempoFinal/3600))
    minu=int((tiempoFinal-hor*3600)/60)
    seg=tiempoFinal-((hor*3600)+(minu*60))
    print(str(hor)+"h:"+str(minu)+"m:"+str(seg)+"s")
    sc.stop()
```

C.3.3. 2ejSpark.py

```
 from __future__ import print_function

import time
import sys

from operator import add

from pyspark import SparkConf, SparkContext

def fecha_de_sesion(datos):
    IP=datos[0]
    FECHA=datos[3]
    archivo = datos[6]
    if (len(datos)>2):
        fecha= time.mktime(time.strptime(FECHA,
               '[%d/%b/%Y:%H:%M:%S'))
        return (IP,(fecha,archivo))



def accesos_en_sesion(IP,valores):
    T=24*60*60
    hora=0
    i=0
    horas = []
    archivos = []
    for (fecha, archivo) in valores:
        if (fecha-hora>=0 and fecha-hora<T):
            archivos[i-1].append(archivo)
        else:
            hora=fecha
            horas.append(fecha)
            i=i+1
            archivos.append([archivo])
    for v in range(len(archivos)):
        return (IP,tuple((horas[v],list(set(archivos[v]))
                                  ,v+1)))


def tiempo_entre_conexiones(IP,valores):
    fecha=0
    diferencia_fecha=0
    for fecha_dada,lista,conn_realizadas in valores:
        diferencia_fecha=int(fecha_dada)-fecha
        fecha=int(fecha_dada)
        return ((IP,lista),tuple((diferencia_fecha,fecha_dada
                                  ,conn_realizadas,1)))


def comportamientos_repetidos(clave,valores):
```

```
    i=0
    clAve=list(clave)
    vAlores=tuple(valores)
    #return valores
    for diferencia_fecha,fecha_dada,conn_realizadas,conn
                                                in valores:
        i=i+1
    return ((clAve[0]),(clAve[1],i))



if __name__=="__main__":
    tiempoInicio=time.time()
    if len(sys.argv)!=2:
        print("Usage:wordcount <file>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="Contar IPs")
    sc.setLogLevel("ERROR")
    ips = sc.textFile(sys.argv[1],1)
            .map(lambda x: x.split(' '))
    ipsGood = ips.filter(lambda x: x[7]==('200'))
                 .sortBy(lambda x: x[3])
                 .map(lambda x: (fecha_de_sesion(x)))
                 .groupByKey()
    ips2 = ipsGood.map(lambda x:
                accesos_en_sesion(x[0],x[1]))
    ips3 = ips2.groupByKey()
    ips4 = ips3.map(lambda x:
                tiempo_entre_conexiones(x[0],x[1]))
               .groupByKey()
    ips41= ips3.map(lambda x:
                tiempo_entre_conexiones(x[0],x[1]))
               .collect()
    ips5 = ips4.map(lambda x:
                comportamientos_repetidos(x[0],x[1]))
               .reduceByKey(add)
    ips6 = ips5.collect()
    ips5.saveAsTextFile("salida-ejemplo2")
    for x in ips6:
         print (":"+str(x)+":")


    tiempoFinal=time.time()-tiempoInicio
    print(str(tiempoFinal))
    sc.stop()
```

Apéndice D

Hive: WordCount

D.1. Java

Fuente: https://www.linkedin.com/pulse/word-count-program-using-r-spark-map-reduce-pig-hive-python-sahu

```
public static class MapClass extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}


public static class Reduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

```
public class WordCount {

    public static void main(String[] args) throws IOException {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
                            // the keys are words (strings)
        conf.setOutputKeyClass(Text.class);
                            // the values are counts (ints)
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(MapClass.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputPath(new Path(args[0]));
        conf.setOutputPath(new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

D.2. Hive

```
1 drop table if exists doc;

2 create table doc(
      text string
3 ) row format delimited fields terminated by '\n'
    stored as textfile;

4 load data local inpath '/home/Words' overwrite into table doc;

5 SELECT word, COUNT(*) FROM doc LATERAL VIEW
  explode(split(text, ' ')) Table as word GROUP BY word;
```

Bibliografía

Bibliografía en papel:
Mario Macías, Mauro Gómez, Rubèn Tous, Jordi Torres *Introducción a Apache Spark: para empezar a programar el big data* 1ª ed. Barcelona: Editorial UOC, 2015

Bibliografía en la red:
Apache Hadoop `hadoop.apache.org`
Apache Spark `spark.apache.org`
Apache Hive `hive.apache.org`
Programación Distribuida https://es.wikipedia.org/wiki/Programación_distribuida
Historia de Hadoop `https://en.wikipedia.org/wiki/Apache_Hadoop`
"Map-Reduce es ineficiente" `http://www.baoss.es/apache-spark/`

Fuentes de Imágenes:
Bit vs Byte: `http://www.classicaonline.com/musicologia/saggi/15-06-15.html`
Tabla ASCII: `http://conceptodefinicion.de/ascii/`
Clúster: `http://datacentrum.kostecky.cz/`
Clúster manipulado: https://dac.com/blog/post/google-datacenter-not-everything-it-seems
Proceso Big Data: `https://www.sonda.com/static/landing/bigdata/img/proceso-big-data.jpg`

Fuentes de datos usados:

El caballero de la armadura oxidada: `https://archive.org/stream/librosEDS/caballeroarmaduraoxidada_djvu.txt`

El Quijote `https://www.gutenberg.org/cache/epub/2000/pg2000.txt`

Tamaño de la Biblioteca del Congreso de los Estados Unidos de América https://es.wikipedia.org/wiki/Biblioteca_del_Congreso_de_Estados_Unidos

Usuarios de Facebook http://www.trecebits.com/2017/02/02/facebook-ya-tiene-1-860-millones-de-usuarios/

Usuarios de Twitter https://www.cnet.com/es/noticias/twitter-319-millones-usuarios-febrero-2017-q4-2016/

Usuarios de Google https://www.multiplicalia.com/redes-sociales-mas-usadas-2017/

Tendencias de Instagram https://www.brandwatch.com/es/2016/08/96-estadisticas-redes-sociales-2016/

Neveras inteligentes https://www.xataka.com/otros-dispositivos/duelo-de-frigorificos-y-sistemas-operativos-lg-le-pone-windows-10-y-samsung-elige-tizen

Neveras de El Corte Inglés http://www.tecnologiadetuatu.elcorteingles.es/actualidad/frigorificos-inteligentes-cuando-las-neveras-hacen-mucho-mas-que-enfriar/

Comentario acerca del Big Data http://www.expansion.com/economia-digital/innovacion/2016/09/20/57e028c6468aeb1e0a8b4620.html

Coches inteligentes http://valencianews.es/motor/bosch-en-el-salon-del-automovil-de-barcelona-el-coche-inteligente-y-conectado/

Tráfico de datos en 2011 http://www.elmundo.es/elmundo/2011/02/08/navegante/1297179889.html

Tráfico de datos en 2016 https://es.wikipedia.org/wiki/InternetTama.C3.B1o

Datos para pruebas:

`http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`

`https://www.adictosaltrabajo.com/tutoriales/hive-first-steps/#05`