Omar KANAN
MS BIG DATA 2016-2017

SHAVADOOP

I. Launching the program:

Shavadoop is a program that takes a file text input and counts the occurrence of each word, then prints it to an output file, all using a cluster of computers with a distributed file system and a MapReduce-like operation. Here are the requirements for it to perform correctly:

- Have a single folder on the distributed file system containing the jar files *"Master.jar"* and *"Slave.jar"*;
- Add in this folder the input text file;
- Also, add a text file containing the list of IP addresses (or hostnames) of the computers you want to use for the task. The format must be one address on each line;
- Now open a terminal and navigate to this folder;
- Run this command:
  "java -jar Master.jar <input text file name> <IP addresses file name> <timeout in ms>"
  The timeout is for testing the SSH connection with the computers: a test request will be sent to each one of them, and if a computer doesn't respond before the end of the timeout, it will be considered as non-available. For example, with a 2 second timeout:
  "java -jar Master.jar input.txt adresses.txt 2000"
- The result with be printed in a file named *"wordcount.txt"* in sorted order. Several folders containing intermediate results will also be created.

II. SSH connectivity:

The main class of the program is the *"Master"* class. The first thing it does is to test the SSH connection with all the computers (slaves) given in the IP addresses file. But we don't want to do this one by one for obvious timing reasons, so we create one thread for each slave via the *"SSHConnectivityTester"* class which implements the *"Runnable"* interface, with the IP address as a parameter, and we launch in parallel all the threads. Each instance contains a flag (default to *"false"*), which is only updated to *"true"* when the SSH connection succeeds.

In the meantime, just after launching the threads, the main thread will sleep for a period equal to the timeout given by the user as an argument. Then after waking up, it will interrupt all the started threads, and check for each one of them if the connection has succeeded, using the instance flag. For each success, the IP address is added to a list which will be used in all the program.

This program assumes that, when an SSH connection succeeds here, the corresponding computer will stay available during the whole execution of the program. In other words, the "fault-tolerance" characteristic of MapReduce isn't implemented here.

III. Splitting:

Now that we have a list of available slaves, we split the input text file into several "Sx" files, one for each slave. For this we use the length of the input file in bytes, and split it equally between each slave. The problem here is that we don't want to cut the file at the middle of words, so for each part, we keep adding bytes one by one until we reach a space character, which byte representation is 32. This means that the last "Sx" file will generally be smaller than the others, but it won't be noticeable for a not-too-small input file.

IV. Mapping:

The core of the program begins here. We now have several "Sx" files, and the same number of available slaves, and we want each slave to do the mapping job on one "Sx" file. So, we create a list of instances of the *"JobLauncher"* class, one for each slave (note that the same class is used for launching both mapping and reducing jobs, but it would have been smarter to create one class for each case). We pass all the parameters needed to know how to correctly execute the job.

Then each *"JobLauncher"* (which implements *"Runnable")* starts a thread which will create an SSH process using the native *"ProcessBuilder"* class, with the corresponding slave as recipient and all the needed arguments. Each instance also contains an *"ArrayBlockingQueue"* which is an array that allows one write and one read action to be executed concurrently. This array will be used to store the responses of the slave, while the master retrieves them at the same time.

Now each slave will run the mapping job on its "Sx" file. So, it reads the file line by line, and for each line it applies the following normalization operations:

- Putting the text to lower case;
- Removing accentuation (to prevent duplicate words and biased results);
- Replacing all special characters (different from [a-z]) by a space character;
- Removing all frequent words using a given list (which can be modified if needed);
- Splitting the text in words, using space characters as delimiters;

Then, for each word of the line, we write to the "UMx" output file *"word 1"*. If it is a new word (not seen before), we also send it to the corresponding *"JobLauncher"* via SSH using the standard output (to know if it is a new word or not, we use a *"HashSet"* that we update continuously).

Note that we implemented a multi-threaded version of the mapping job, using the *"MapLauncher"* class with a number of threads equal to the number of available processors of the slave.

In the meantime, from the master side, each *"JobLauncher"* has an open stream with its slave, which retrieves one by one the unique words sent by the slave and places them at the tail of its *"ArrayBlockingQueue"*, until the process is over. While so, the master keeps looping on all the arrays until every slave has ended its job, and for each word retrieved, it adds it to a *"HashMap"* as a key, and adds the path of the "UMx" file that contains this word to a list of paths which is the value associated to the key (this path can be retrieved because the master knows which "UMx" is created by which slave).

V. Shuffling:

We now have a set of “UMx” files, and a map of the unique words (keys) associated with lists of “UMx” files containing each word. We then randomly split the keys between each slave, and for each group of keys, we merge into a set the lists of “UMx” files containing these keys. As in the mapping phase, we create a list of *“JobLauncher”* with the necessary parameters, but we can’t yet launch the SSH processes.

Yeah, because poor SSH doesn’t like it when we send a huge number of arguments, so we can’t send all the keys here. Even when constructing a single argument by joining all the keys with a separator (using a sequence of characters that doesn’t interfere in the SSH command compilation, here triple-underscore), he was still crying over the length of the arguments. So, we create a “Keys” file for each group of keys, then launch in new threads every SSH process, specifying in the arguments the corresponding “Keys” file, as well as the list of corresponding “UMx” files concatenated with the triple-underscore separator (we can do it here because the number of “UMx” files to send is small, not greater than the number of slaves).

VI. Reducing:

Each slave will now do its reducing job using the following operations:

- Reading the given “Keys” file;
- Storing the retrieved keys in a *“HashMap”* with 0 as initial value;
- Reading all the given “UMx” files line by line;
- For each line (which contains *“key 1”),* incrementing the value associated to the found key in the *“HashMap”*.

Then for each key, we send *“key count”* to the corresponding *“JobLauncher”* via the standard output, and we also write it to the “RMx” output file. Same as for the mapping phase, each *“JobLauncher”* reads the pairs sent by its slave and stores them in an *“ArrayBlockingQueue”*, while the master loops on every array to retrieve them until every slave has ended its job.

Note that, as in the mapping phase, we implemented a multi-threaded version of the reducing job, using the *“ReduceLauncher”* class with a number of threads equal to the number of given “UMx” files. But here, because most of the job here is reading the input files and writing the output file, which cannot be parallelized, the performance doesn’t change, but it would increase if the actual reducing operations were the longest part.

VII. Assembling:

Finally, the master has a *“HashMap”* of each word and its number of occurrences, and we now create a sorted list of the words using the *“KeyComparator”* class which implements the *“Comparator”* interface, with the most frequent words first. Then we just print the sorted pairs to the output file “wordcount.txt”. Note that this operation wouldn’t be done in MapReduce, the results would be the “RMx” files with sorted pairs (here we didn’t sort them because we don’t use the “RMx” files).

VIII. Results:

For "*forestier_mayotte.txt*":

biens 8	agroforestiere 2	forestiere 2
forestier 7	volontairement 2	aides 2
code 6	proprietaires 2	article 2
partie 5	livre 2	constitue 2
gestion 4	titre 2	garanties 2
dispositions 4	attachees 2	benefice 2
agroforestiers 4	demembrer 2	engagement 2
communes 4	regis 2	propriete 2
mayotte 4	bonne 2	mayotted 1
forestiers 4	presentant 2	plan 1
legislative 3	unite 2	matieres 1
publiques 2	prioritairement 2	sommaire 1
preliminaire 2	accorde 2	
souscrivent 2	table 2	

For "*deontologie_police_national.txt*" (best of 50):

police 38	donne 4	responsable 2
nationale 20	autorites 4	nature 2
article 20	droits 4	tenu 2
titre 13	public 4	place 2
autorite 11	personnes 4	actes 2
fonctionnaires 10	missions 4	loi 2
code 9	personne 4	conditions 2
fonctionnaire 9	subordonne 4	penale 2
commandement 8	fonctions 3	executer 2
ordre 8	generaux 3	judiciaire 2
devoirs 7	echeant 3	acte 2
deontologie 7	ier 3	part 2
controle 6	respectifs 3	respect 2
execution 5	present 3	citoyen 2
ordres 5	preliminaire 3	inspection 2
responsabilite 4	regles 3	proteger 2
doit 4	hierarchique 3	

For "*domaine_public_fluvial.txt*" (best of 50):

article 175	prix 36	trouve 24
bateau 106	saisie 36	patron 24
immatriculation 82	nom 34	instance 24
tribunal 72	delai 34	livre 23
lieu 69	inscriptions 32	public 22
bureau 50	doit 32	encheres 22
code 48	date 32	declaration 22
titre 48	certificat 32	mise 22
navigation 46	batelier 30	conditions 21
bateaux 43	jours 30	navigables 21
domicile 42	vente 30	voies 21
creanciers 40	greffe 28	greffier 20
interieure 39	acte 26	dispositions 19
juge 38	faite 26	tenu 18
proprietaire 37	hypotheque 26	grande 18
commerce 36	registre 26	procede 18
inscription 36	execution 25	

For "*sante_publique.txt*" (best of 50):

article 46868	section 4734	ministre 3660
sante 20317	prevues 4716	regionale 3624
agence 8804	code 4563	commission 3580
etablissement 8616	etablissements 4456	medicale 3562
directeur 8608	membres 4414	arrete 3544
conditions 8104	soins 4392	modalites 3412
conseil 7586	articles 4316	exercice 3372
etat 7362	personnes 4192	president 3276
general 6920	medicament 4184	medecin 3162
autorisation 6702	alinea 4036	toute 3124
dispositions 5926	avis 4036	representant 3104
charge 5658	mentionnes 3966	deux 3078
securite 5288	delai 3962	activite 3058
demande 5206	present 3792	doit 3044
titre 5170	nationale 3762	ordre 3044
produits 5138	mise 3688	medicaments 3036
application 5052	personne 3672	

IX. Performances:

	1 slave	3 slaves	5 slaves	7 slaves	10 slaves
forestier_mayotte.txt	1s	7s	10s		
deontologie_police_national.txt	1s	7s	10s		
domaine_public_fluvial.txt	1.5s	8s	9s		
sante_publique.txt	32s	28s	22s	18s	20s

We notice that the number of slaves must be adjusted to the size of the input file, because too many slaves decreases the performance (too much communication time compared to computing time).
These timings are averaged over 5 iterations.