

Sistemes Distribuïts

Pràctica 2: Cloud Map-Reduce

Aleix Marín i Tena
Cristòfol Daudén Esmel

Introducció

Map-Reduce és un model de programació i una implementació associada per processar i generar grans conjunts de dades amb un algoritme distribuït paral·lel en un clúster.

En la pràctica anterior es va implementar aquest programa utilitzant un model d'actors mitjançant la llibreria pyactor del llenguatge de programació python.

En aquesta pràctica en canvi, hem utilitzat tecnologies Cloud. Nosaltres hem escollit Amazon Web Services (AWS). Aquestes tecnologies són oferides per Amazon per Internet i donen una gran quantitat de serveis al núvol.

Alguns dels serveis oferits per AWS que nosaltres utilitzarem en aquesta pràctica son **Lambda, S3 i CloudWatch**:

- Lambda es una plataforma de Serverless computing, aquest servei proporcionat permet executar codi en el núvol.
- S3 és un sistema de fitxers al núvol, altament escalable i que ofereix molts altres serveis i prestacions associades, com per exemple, la generació d'events en modificar, crear o esborrar dades.
- CloudWatch és un servei d'anàlisi de dades i de monitorització d'ús dels diferents serveis dels AWS.

Problema a resoldre

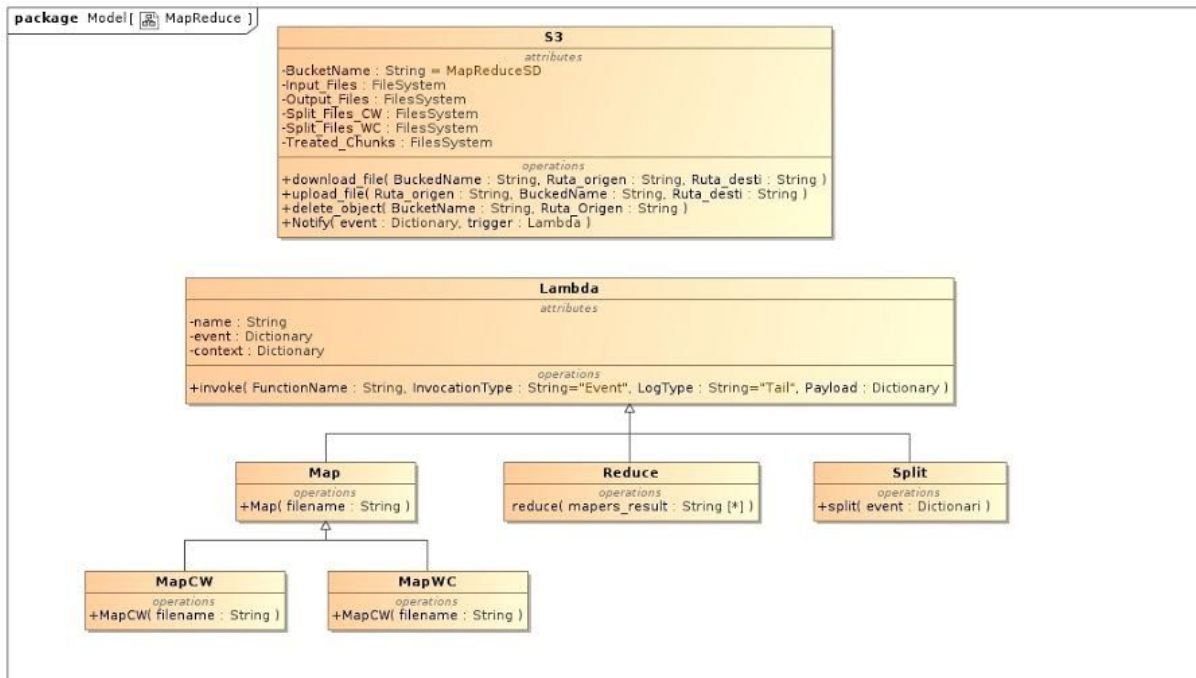
En aquesta pràctica s'ha implementat el sistema Map-Reduce aplicat a resoldre dos problemes: el comptatge del nombre de paraules dins d'un text i el comptatge del nombre d'ocurrències de cada paraula dins d'un text.

L'ús dels AWS ens permetria escalar aquest sistema Map-Reduce de manera massiva, aconseguint una paral·lelització i uns speed-ups superiors. A més, els càlculs no s'executen en la nostra màquina, sinó que disposem de servidors que duen a terme el treball que faria la nostra CPU.

Tot i que la major part del treball s'ha dut a terme editant online els fitxers, trobareu la versió final del codi al [nostre repositori de GitHub](#).

Creació de l'entorn de treball

En aquest apartat explicarem quins són els passos (a grans trets) que hem hagut de fer per tal de crear la arquitectura per a poder executar la nostra solució. Podem trobar els “components” més importants en el següent diagrama de classes:









I cada pas en detall:

1. Creem un compte AWS.

- Creem un IAM roll donant-li els permisos necessaris per a fer les operacions desitjades. Nosaltres hem optat per crear un superusuari així no tenim problemes de permisos, tot i que en donar permisos d'execució de lambdes i d'escriptura i lectura en S3 n'hi hauria suficient.

2. Creem un repositori a s3.

- Creem la següent estructura de directoris

<input type="checkbox"/>	Name 
<input type="checkbox"/>	 Input_Files
<input type="checkbox"/>	 Output_Files
<input type="checkbox"/>	 Split_Files_CW
<input type="checkbox"/>	 Split_Files_WC
<input type="checkbox"/>	 Treated_Chunks

- **Input_Files:** Conté els fitxers d'entrada a tractar.
- **Output_Files:** Conté els fitxers de resultat. Aquests fitxers contenen el diccionari generat en python en format txt.
- **Treated_Chunks:** És la carpeta d'Output de la fase de Map. Els resultats parcials es guarden aquí. Cada fitxer és un diccionari del tros del fitxer processat pel mapper.
- **Split_Files_CW i Split_Files_WC:** És la carpeta d'output de la fase d'split. Segons la funcio escollida (WC o CW) l'split guardarà en una o altra carpeta.

3. Creem les nostres funcions lambda:

- **split:** Divideix el fitxer d'entrada:

```
import boto3
import botocore
import os
import codecs
```

```
# aquesta funcio rep com a parametre la funcio a executar [WC|CW], el nombre de
mappers i el nom del fitxer a tractar.
# Llegeix de Input_Files/event.nomfitxer, divideix el fitxer en event.nummappers i
guarda cada tros a la carpeta Treated_Chunks_event.funcio/numfitxer.txt
# això causara l'activació de les funcions mapper corresponents en escriure a la
carpeta concreta.
```

```
def lambda_handler(event, context):
    # Obtenim host client
    s3_client = boto3.client('s3', aws_access_key_id='???' ,
                              aws_secret_access_key='???')

    # Download the file from S3
    s3_client.download_file('mapreducesd', 'Input_Files/'+event.get("filename"),
                              '/tmp/input.txt')

    # Count lines
    num_lines = sum(1 for line in open("/tmp/input.txt"))

    # Calculate the size of the chunks (adding 1 could be a problem for small files
    and many mappers, but we need it since it's a non-real operation)
    splitLen = (num_lines / int(event.get("nummappers"))) + 1
```

```

count = 0
at = 0
dest = None
input = open('/tmp/input.txt', 'r+')
for line in input:
    if count % splitLen == 0:
        if dest: dest.close()
        at += 1
        dest = open('/tmp/'+str(at)+'.txt', 'w+')
    dest.write(line)
    count += 1

dest.close()
# upload chunks
for num in range(1, at + 1):
    s3_client.upload_file('/tmp/'+str(num)+'.txt', 'mapreducesd',
'Split_Files_'+event.get("function")+'/o'+str(num)+'.txt')

# Now mappers are running
return 0

```

- **map:** Aplica una funció sobre un tros del fitxer d'entrada. Nosaltres hem considerat la funció aplicada inherent a la funció lambda, pel que tenim una funció lambda per cada funció que volem aplicar. En aquest cas, podem aplicar l'algoritme de comptar aparicions de totes les paraules (WC) o de comptar el nombre de paraules total (CW).

- **mapWC:**

```

import boto3
import botocore

# Aquesta funcio te com a trigger la creacio d'un fitxer en la carpeta
Split_Files_WC.
# La funcio rep aquest event com a parametre, que conte la ruta al fitxer que ha
causat l'activacio del trigger.
# Aquest mapper executa la funcio WC i salva el resultat a Treated_Chunks

def lambda_handler(event, context):
    # Obtenim host de s3
    s3_client = boto3.client('s3', aws_access_key_id='???',
aws_secret_access_key='???')

    # Guardem nom en una variable
    REMOTE_PATH = event.get("Records")[0].get("s3").get("object").get("key")
    NUM_FOLDERS = len(REMOTE_PATH.split('/'))
    NAME = REMOTE_PATH.split('/')[NUM_FOLDERS-1]

    # Download the file from S3 and obtain plain text
    s3_client.download_file('mapreducesd', REMOTE_PATH, '/tmp/input.txt')
    text = open('/tmp/input.txt', 'r').read()
    result = {}

    #WC
    text = text.translate(None, "-?!.,:;()\"").lower() # deleting trash characters
    for line in text.split("\n"):
        for word in line.strip().split():
            if word in result:
                result[word] = result.get(word)+1
            else:
                result[word] = 1

```

```

# Save result into /Treated_Chunks/$(event.filename())
with open('/tmp/output.csv', 'w') as f:
    for key, value in result.items():
        f.write('%s:%s\n' % (key, value))

# Upload result into Treated_Chunks folder
s3_client.upload_file('/tmp/output.csv', 'mapreducesd', 'Treated_Chunks/o'+NAME
)

# Auto-clean
s3_client.delete_object(Bucket='mapreducesd', Key=REMOTE_PATH)

return 0

```

- **mapCW:** Funciona igual que l'altre mapper, però duu a terme una funció diferent.

```

#WC
result={"counter":0}
for line in text.split("\n"):
    for word in line.strip().split(): # aqui no fa falta eliminar caracters
brossa ja que nomes comptem ocurrences
        result["counter"] = result["counter"]+1

```

- **reduce:** Fusiona els fitxers de resultat en un de sol.

```

import boto3

def lambda_handler(event, context):

    # Obtain host
    s3 = boto3.client('s3', aws_access_key_id='???', aws_secret_access_key='???')

    # Data output structure
    mappers_output = []
    result = {}

    # Getting all files in the output directory
    event.get("Records")[0].get("s3").get("object").get("key").split("/")[-1] = WORKING_DIR

    resp = s3.list_objects(Bucket='mapreducesd', Prefix=WORKING_DIR+'/o')
    print resp
    filenames = resp.get('Contents')

    #Fem la comprovacio de que ja estan tots els resultats del maper en el
    directori mapper_output/
    if filenames:
        print 'mappers havent finished; going to sleep'
        return 1

    # if not, mappers have finished; so list files in Treated_Chunks/

    # if mappers have finished but files does not contain filenames (the directory
    is still empty)
    # then synchronization problem (due to the lack of consistency)
    files = s3.list_objects(Bucket='mapreducesd',
    Prefix='Treated_Chunks/o').get('Contents')
    filenames = []
    # append filenames
    for file in files:
        print file
        filenames.append(file['Key'])

```

```

# Read and append all results from all mappers:
for file in filenames:
    print file #TEST
    s3.download_file('mapreducesd', file, '/tmp/input.txt')
    # Reading a dictionary from a file
    with open('/tmp/input.txt', 'r') as f:
        dict = {}
        text = f.read()
        for line in text.split('\n'):
            values = line.split(':')
            if values[0] != '':
                dict[values[0]] = int(values[1:len(values)][0])
        mappers_output.append(dict)

print mappers_output
#Reduce function:
for hashes in mappers_output:
    for key in hashes.keys():
        if key in result:
            result[key] = result[key]+[hashes[key]]
        else:
            result[key] = [hashes[key]]
for word in result.keys():
    if len(result[word]):
        result[word] = sum(result[word])

#Saving the result in S3
with open('/tmp/output.csv', 'w') as f:
    for key, value in result.items():
        f.write('%s:%s\n' % (key, value))

s3.upload_file('/tmp/output.csv', 'mapreducesd', 'Output_Files/out.txt')

# Autoclean
for file in filenames:
    print file
    s3.delete_object(Bucket='mapreducesd', Key=file)

return 0

```

4. Creem events per a dur a terme la sincronització entre la fase de split i map i entre la fase de map i reduce. Aquests events permetran activar les funcions de la següent fase un cop la fase actual hagi acabat.

- **split:** No necessita events. És la funció “pare” ja que és la que rep els paràmetres i la primera en ser cridada, causant l’activació de la resta de funcions.
- **map:** Aquesta funció s’activa per event. Cada tipus de map, mapCW i mapWC, “escolta” l’event de creació de qualsevol fitxer procedent de la carpeta Split_Files_CW o Split_Files_WC, respectivament. D’aquesta manera, un cop l’split ha dut a terme la seva funció i ha pujat els fitxers partits a la carpeta corresponent (en funció del paràmetre “function” rebut per la funció split), s’activa un map per cada fitxer creat en aquell directori. Cada mapper activat per event rep per paràmetre el fitxer que l’ha activat, de manera que ja sap quin és el fitxer que aquell mapper concret ha de tractar.

Un cop el map ha pujat el fitxer tractat a la carpeta Treated_Chunks, elimina el fitxer front de la carpeta Split_Files_”Function”, causant l’activació del reducer.

- **reduce:** Aquesta funció escolta dos events possibles, però en cada execució només un dels tipus d’event serà activat. La funció reduce escolta events d’eliminació de fitxers de la carpeta Split_Files_”Function”. En aquest cas doncs tenim dos events: un que escolta eliminació de fitxers de la carpeta Split_Files_WC i un altre de la carpeta Split_Files_CW.

D’aquesta manera, cada cop que s’esborra un fitxer (degut a que un mapper ha acabat la seva feina) el reducer llista els fitxers del directori que ha produït l’event. Si troba algun fitxer aborta el pas de reduce, sinó, vol dir que el pas de map ja ha acabat, per tant, comença el pas de reduce. Per tant, el reducer serà activat tants cops com mappers hi han, però només en l’última activació es durà a terme el pas de reduce.

5. Donem permisos a les lambdes descrites en el pas anterior per a ser executades a través d’event.

6. Donem permisos a les funcions lambda per a ser monitoritzades pel servei CloudWatch. Això ens serà útil ja que l’activació per events no permet rebre l’output de la lambda directament, sinó que hem d’utilitzar un servei com CloudWatch per a veure els logs d’output de la funció

7. Pugem els fitxers de prova. Hem utilitzat els fitxers de la primera fase per a dur a terme la validació de la solució, però per a l’apartat resultats utilitzarem fitxers amb diferents repeticions del fitxer Sherlock.txt, que conté totes les novel·les de Sherlock Holmes. Concretament 5, 10, 50 i 100 repeticions del mateix.

8. A partir d’aquests pas la configuració serveix per a poder invocar funcions des de la nostra màquina local. Primer instaleu la llibreria boto3.

```
$ pip install boto3
```

9. Configurem el client instal·lant aws a la màquina local i executant la comanda de configuració. Això ens permetrà crear instàncies dels clients per als diferents serveis d’AWS sense haver d’introduir user i pass cada cop que ho fem.

```
$ pip install awscli --user
```

```
$ aws configure
AWS Access Key ID [None]: ???
AWS Secret Access Key [None]: ???
```



```
Default region name [None]: us-east-2
Default output format [None]: json
```

10. Creem el programa “master” que s’encarrega de cridar la funció “primaria”. Com hem utilitzat events per a que la resta de lambdes es vagin executant només cal cridar a aquesta funció.

```
import boto3
import json
import os
import time
import sys

# Creating json data with arguments
payload = {
    "filename": sys.argv[1],
    "nummappers": sys.argv[2],
    "function": sys.argv[3]
}

if __name__ == "__main__":

    # Create json file
    payload = json.dumps(payload)

    # Create clients
    s3_client = boto3.client('s3')
    lambda_client = boto3.client('lambda')

    # check file available
    fitxers = s3_client.list_objects(Bucket='mapreducesd', Prefix= 'Input_Files/'
).get( 'Contents')
    filenames = []

    for file in fitxers:
        filenames.append(file['Key'].split('/')[1])

    # --> If not exists, abort
    if sys.argv[1] not in filenames:
        print 'Input file is not in the Input folder! Aborting...'
        sys.exit(1)

    # Get working directory
    wd = os.path.dirname(os.path.realpath(__file__))

    # Clean Output files in local machine
    os.system('rm -f '+wd+'/Out.txt')

    # try to delete output object in server
    try:
        s3_client.delete_object(
            Bucket='mapreducesd',
            Key='Output_Files/out.txt')
    except botocore.exceptions.ClientError:
        print 'Output folder clean.'

    # measure begin time
    begin = time.time()

    # invoke split function
    response = lambda_client.invoke(
        FunctionName='Split',
        InvocationType='Event',
        LogType='Tail',
```

```

        Payload=payload
    )

    # wait until file is available
    numfiles = None
    while numfiles is None:
        numfiles = s3_client.list_objects(Bucket='mapreducesd',
        Prefix='Output_Files/out.txt').get('Contents')
        print 'Result is not available yet. '+str(time.time()-begin)+' seconds
transcurred'

    os.system("clear")
    print 'Result Found!'

    # measure end time
    end = time.time()

    # download file
    response = s3_client.download_file(
        'mapreducesd',
        'Output_Files/out.txt',
        wd+'/Out.txt')

    # calcule RoundTrip_Time
    RoundTrip_time = end - begin

    # Show results file
    print open( wd+'/Out.txt', 'r').read()

    # show transcurred time
    print 'MapReduce RoundTrip time is '+str(RoundTrip_time)+' segons'

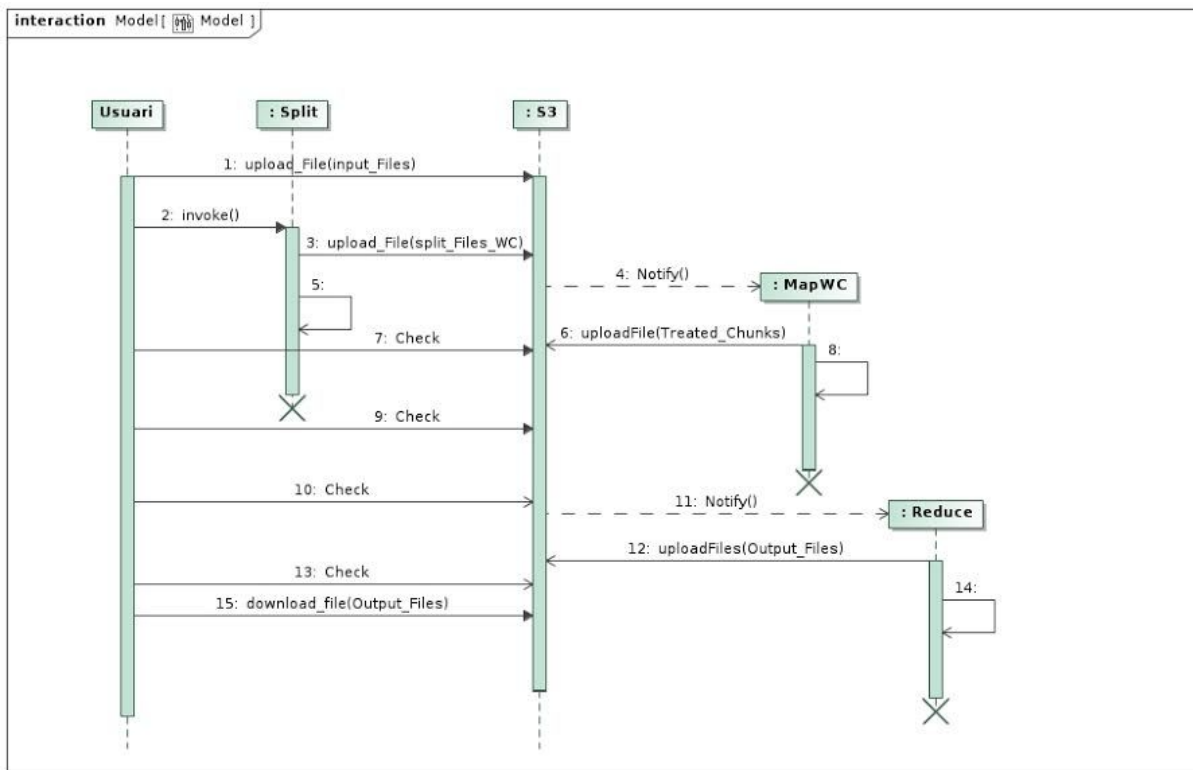
    # Salvem resultats de l'execucio
    os.system('echo \"+sys.argv[1]+';'+ sys.argv[2]+';'+
sys.argv[3]+';'+str(RoundTrip_time)+'\" >> result.csv')

    # Borrem fitxer Out.txt
    os.system("rm -f "+wd+'/'+'Out.txt')

```

Funcionament

En aquest apartat explicarem el que succeeix en un procés normal d'execució fent especial menció als mecanismes de sincronització entre fases. Aquesta informació pot trobar-se resumida en el següent diagrama:



En detall l'explicació de cada pas:

1. Desde el programa master creem un fitxer json amb el paràmetres. Els paràmetres es defineixen a través d'una estructura diccionari:
 - Nombre de mappers (i de trossos de fitxers).
 - Funció a executar. Aquest valor pot ser [WC|CW].
 - Nom del fitxer d'input. Només cal posar el nom del fitxer de la carpeta Input_File escollit.
2. El programa master invoca la funció split.
3. El programa master posa el cronometre en marxa i es queda en bucle esperant a que aparegui el fitxer Output_Files/out.txt, que és el fitxer de sortida del reduce.
4. Paral·lelament al pas 3, el fitxer es dividit i cada tros es pujat a la carpeta "Split_Files_WC" o "Split_Files_CW" depenent de la funció seleccionada com a paràmetre.
5. La creació d'objectes en aquestes carpetes provoca la invocació de les funcions "MapWC" or "MapCW", depenent de la carpeta on s'ha creat el fitxer.
6. La funció de map es invocada per cada nou fitxer creat. Cada funció de map rep l'event que l'ha activat com a paràmetre, el qual conté el path al fitxer que ha causat la invocació de la lambda.
7. Cada map aplica la seva funció de transformació al seu tros. Cada tros tractat es pujat al directori Treated_Chunks. Després, el tros original que contenia les dades

sense tractar es esborrat. Això permet unes execucions més netes i (barates) i provoca la invocació del reducer.

8. Cada delete a la carpeta "Split_Files_FUNCTION" provoca la invocació del reducer.
9. Cada invocació del reducer comprova que no hi hagin fitxers a la carpeta "Split_Files_FUNCTION". Si el directori està buit el pas de reduce comença, sinó, la funció reduce aborta sense fer res.
10. Un cop s'ha fet el pas de reduce es guarda el fitxer out.txt de resultat al directori Output_Folder. La creació d'aquest fitxer provoca que el programa master es desbloquegi (pas 3) i continuï la seva execució.
11. El master para el cronòmetre, descarrega el fitxer out.txt, el mostra per pantalla i salva els arguments d'execució i la mesura de temps.

El master també duu a terme diferents operacions de clean per a facilitar execucions posteriors.

Decisions de Dissenys

Utilització d'events per a la sincronització

Degut a que Amazon ja ofereix aquest mecanisme de sincronització amb S3 ens vam decidir a utilitzar-ho degut a que utilitzar un servei extern per a la sincronització podria augmentar de manera notable l'overhead del procés Map-Reduce.

Els reducers escolten events de "delete" en lloc de "create"

Aquesta estratègia es podria haver canviat i fer que el reducer escoltés la creació de fitxers en el directori Treated_Chunks. D'igual manera, si el reduce s'activa quan hi ha els n fitxers, els reducers es despertaran n-1 cops i només l'últim reduce serà el que realment executi la funció. Per tant, en termes d'eficiència és el mateix una estratègia que l'altra. Tot i així, fer que el pas de reduce comenci quan hi han n fitxers en el directori implica que se li ha de passar d'alguna manera aquest paràmetre n al reducer. Per tant, és més òptim que el reduce comenci aquesta fase quan un directori és buit en lloc de amb n fitxers, ja que la primera estratègia no implica aquest paràmetre.

Un directori Split_Files_[Function] per a cada funció

Nosaltres hem considerat la funció aplicada inherent a la funció lambda, pel que tenim una funció lambda per cada funció que volem aplicar. En aquest cas, podem aplicar l'algoritme de comptar aparicions de totes les paraules (WC) o de comptar el nombre de paraules total (CW). Aquest approach és opcional, podríem haver implementat una funció lambda map que apliqués una funció d'un *pool* de funcions segons el paràmetre rebut. L'eficiència a priori és la mateixa. Hagués sigut interessant aquesta funcionalitat si tinguéssim més funcions per triar.

Split

Igual que en la fase anterior, la funció split parteix en el nombre de fitxers desitjat. Això automatitza el procés i ens ha permès obtenir els resultats més ràpidament, ja que llavors el nombre de funcions map es pot parametritzar. Per així dir-ho la funció split actua com a interfície de tot el procés de Map-Reduce.

Mesura de temps

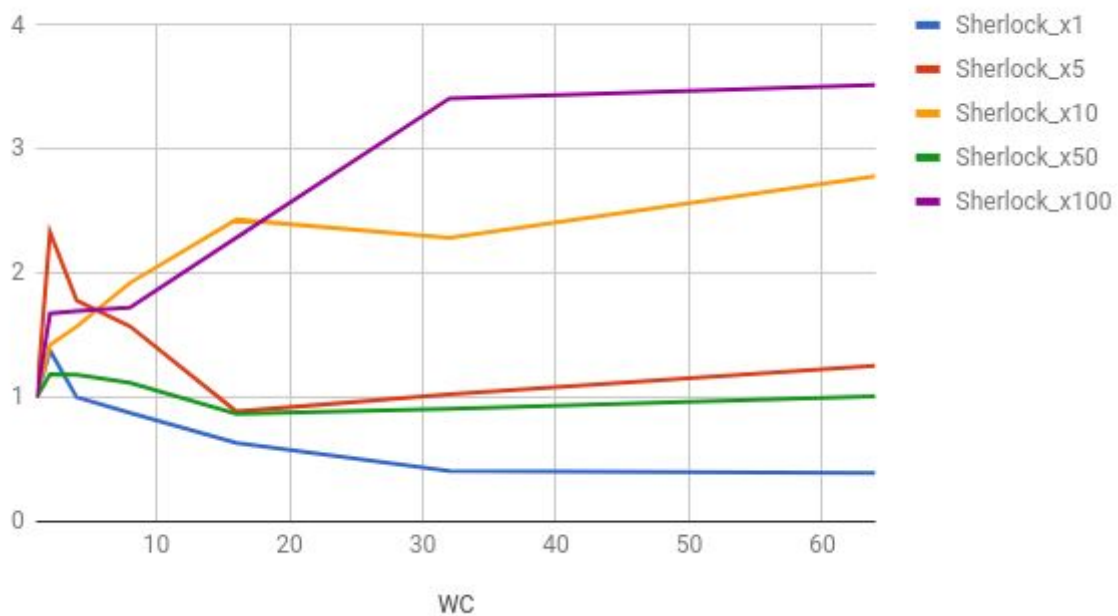
En la fase anterior vam mesurar el temps del procés Map-Reduce estrictament. Com ja hem vist a la teoria, cal utilitzar la mesura de temps en la mateixa màquina, però per la topologia

que tenim en aquesta pràctica no podem fer la mesura de temps d'aquesta forma i només mesurar el temps transcorregut a la fase de MapReduce. Per tant, calcularem el RoundTrip time, que en aquest cas serà el temps transcorregut desde que s'invoca la lambda fins que apareix el fitxer out.txt en el directori Output_Files. Això implica que la millora per utilitzar més maps pot ser menys notable degut a que també estem mesurant overhead.

Decisions de Dissenys

Word-count

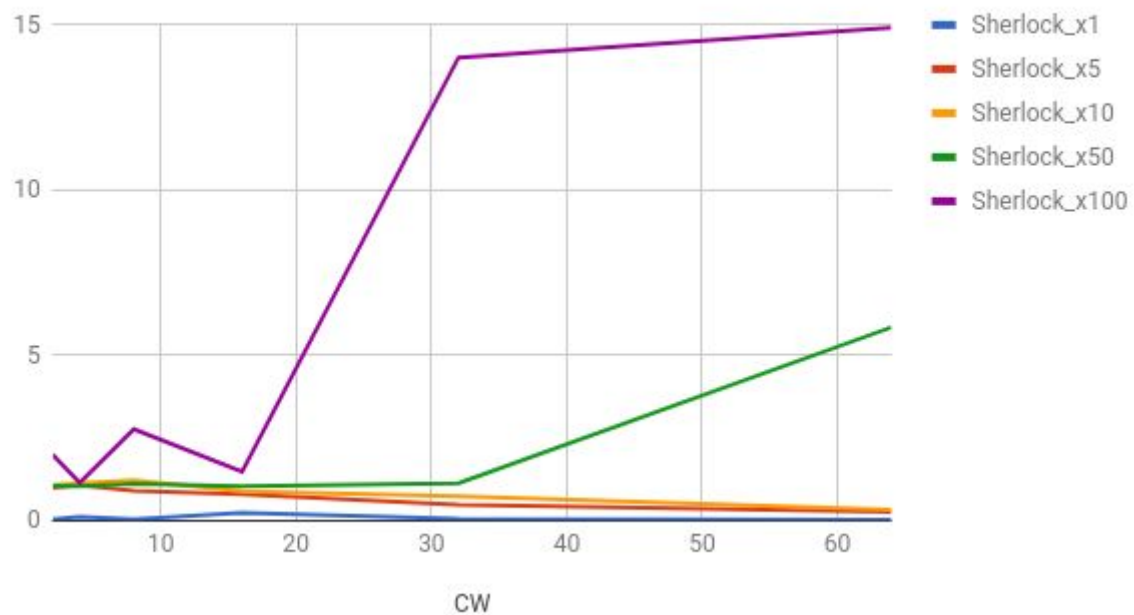
WC Speed-ups



Podem observar una tendència visible: a major quantitat de dades, major és l'speed-up, arribant a pics de 3,5 per a cent repeticions del fitxer Sherlock.

Count-words

CW Speed-ups



L'speed-up amb el fitxer de 100 és enorme. La serie de cinquanta repeticions també arriba a un speed-up de fins a 5 a mesura que augmenten el mappers.

Validació

Per a validar la nostra solució s'han utilitzat els fitxers subministrats per a aquest fi, sent aquests el llibre del Quixot, La Bíblia i Sherlock Holmes. El problema de tenir fitxers tant grans és que l'output generat és enorme; per això hem dut a terme diferents jocs de proves amb textos curts que nosaltres mateixos hem inventat.

L'output en aquests casos ha sigut satisfactori i coherent amb l'esperat.

D'igual forma, per a generar les dades de l'apartat de resultat hem utilitzat un script senzill Bash per a iterar en l'execució del programa master.py en funció de diferents funcions, fitxers i nombre de maps.

Discussió

En aquesta fase la tendència és molt més clara del que ja es podia veure en els gràfics de la fase anterior.

Amb una sola repetició del Sherlock podem veure com l'speed-up és proper a 0. L'overhead generat per partir el fitxer en 64 trossos, pujar-lo al núvol i activar els 64 mappers resulta molt costós en temps.

La tendència pot explicar-se tenint en compte l'overhead que suposa treballar en distribuït. Si la quantitat de dades amb les que treballem és petita, cal fragmentar el fitxer en més trossos i cal obrir més connexions per a contactar amb els mappers, afegint més temps de retard i més overhead de dades. En aquest temps, la versió seqüencial és capaç de processar tot el fitxer completament, és a dir, tasques amb aquesta quantitat de dades no són susceptibles a ser paral·lelitzades amb aquest mètode, degut a que amb una sola màquina en tenim suficient.

En el CW podem observar que les series amb poques repeticions es veuen molt més penalitzades per l'overhead que la funció WC. Podem atribuir aquest fet a que la funció CW és més senzilla i es fa en menys poc temps, ja que en local la funció s'executa en un cop molt més ràpid.

Conclusions

Map-Reduce, juntament amb els serveis Cloud, ens permeten implementar una arquitectura per a tractar grans volums de dades de manera paral·lela en un sistema distribuït sense que nosaltres haguem de disposar de les màquines per a dur a terme el processament.

L'speed-up per aquest mètode augmenta a mesura que augmenta la quantitat de dades tractar en proporció al nombre de mappers. D'aquesta manera evitem que l'overhead ocupi un percentatge important del nostre processament, augmentant la velocitat de tractament de les dades.

A més, aquest speed-up es veu penalitzat per la complexitat de l'algorisme.

