Projet 8 :
« Déployez un modèle dans le cloud »

présentation du 28 janvier 2022

Plan de la présentation

1) Problématiques métier

2) Présentation des données

3) Solutions retenues

4) Traitement des images avec PySpark en local

5) Traitement des images avec PySpark dans le cloud (AWS EMR)

6) Conclusion

7) Perspectives

Problématiques de la start-up "Fruits!"

Fruits!

Contexte :
- start-up qui veut mettre en place un moteur de classification d'images de fruits.
- construction d'une première version de son architecture Big Data.
- utilisation de données disponibles sur Kaggle comme point de départ pour la POC.

Mission :
- développer dans un environnement Big Data une première chaîne de traitement des données : preprocessing et réduction de dimension.

Contraintes :
- anticiper une augmentation très rapide du volume de données après la livraison de ce projet.
- développer des scripts en PySpark.

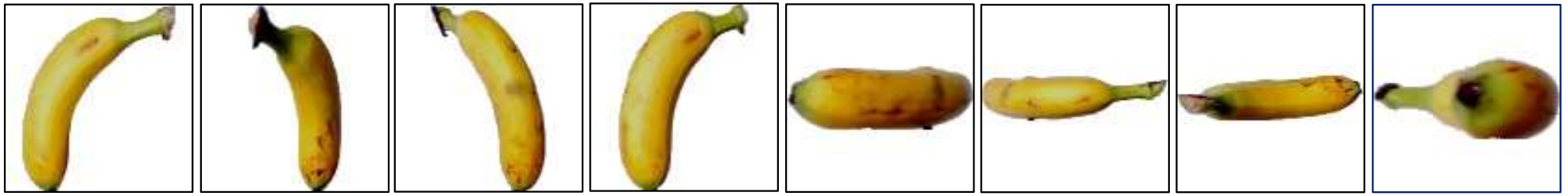
Suggestions :
- utiliser le cloud AWS pour profiter d'une architecture Big Data.

Présentation des données

Disponibles sur Kaggle : https://www.kaggle.com/moltean/fruits

~90000 images :
- 131 fruits et légumes différents,
- photos en studio avec fond blanc,
- sous différents angles,
- dimensions : 100 x 100 pixels x 3 (RGB),
- format JPEG,

Exemple :



Données utilisées pour la POC de notre projet :
- images de 10 fruits (10 dossiers) différents, soit ~5000 images.
- accès via un bucket S3.

Solutions retenues

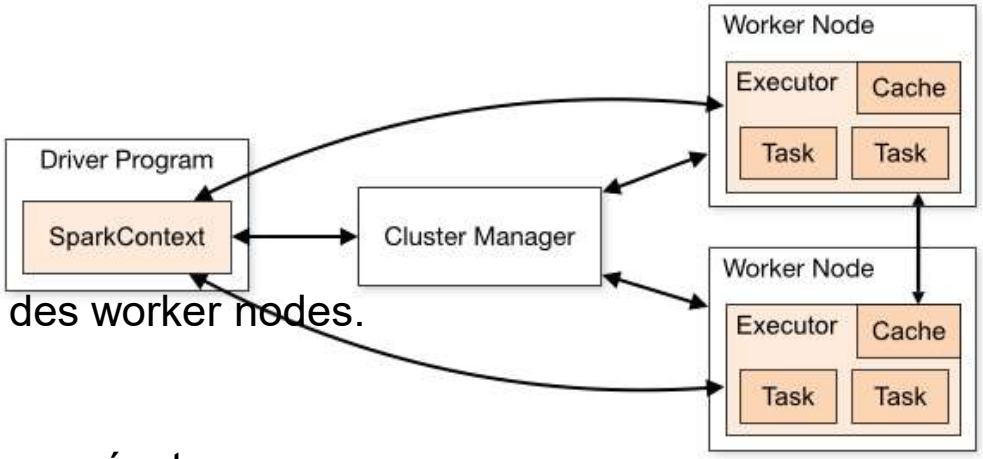
Approaches envisagées pour générer des descripteurs :

1- Techniques classiques de vision par ordinateur pour générer des descripteurs (SIFT, ORB, SURF) puis recherche de n clusters pour générer des features à n dimensions pour chaque image.

2- Transfer learning avec un réseau de neurones préentrainé.

→ Préférence pour l'approche 2 (qualité des features, vitesse de calcul, simplicité). **Choix de VGG16**.

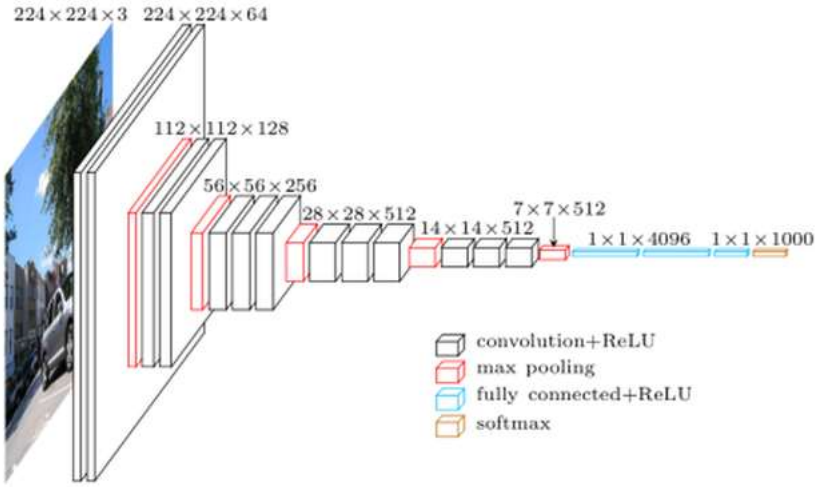
Apache Spark pour l'architecture de calcul Big Data :



- Tâches de calculs :
  - distribuées entre executors qui se trouvent sur des worker nodes.
  - réalisables en parallèle.
  - tolérance aux pannes.
- Driver program : le programme principal. Fera appel à des exécuteurs.
- Executor : process qui exécute les tâches de calcul distribué demandées par le driver.
- Cluster manager : gère les ressources du cluster, notamment les interactions entre driver et executor.
- PySpark : API Python pour Spark (langage natif : Scala).

Extraction de descripteurs par VGG-16

VGG-16 :
Réseau de neurones convolutif à 16 couches pour la classification d'images.
Réseau pré-entrainé sur 1.3 millions d'images.
Possibilité de l'utiliser en transfer learning.
Dernière couche : classifieur softmax qui prend un vecteur de dimension 4096 en entrée.



Génération du descripteur (4096 dimensions) d'une image :

1) Prepocessing de l'image (en particulier redimensionnement au format 224*224*3 pixels)
2) Calcul du descripteur via le modèle préentrainé.

Traitement des images avec PySpark

Procédé mis en œuvre, pour chaque image:

Clé (adresse) de l'image → Récupération de l'image sur S3 → Pré-processing de l'image pour l'injecter dans VGG-16 → Calcul par VGG16 de descripteurs de l'image → Obtention d'un vecteur à 4096 dimensions


```python
# fonction d'extraction de feature
def VGG16_extracteur(path, nom_image, model):
    """Transforme un fichier image en un vecteur de dimension 4096.
    Args :
    - path : chemin vers les images (chemin local ou 'S3').
    - nom_image : chemin d'accès à l'image (exemple : 'Apple Braeburn/r_173_100.jpg')
    - model : model_vgg16 ou bc_model_vgg16.
    Returns :
    - liste de dimension 4097 (4096 dimensions de VGG16 + nom_image).
    """
    # create path to the image stored locally
    if path == 'S3':
        # il faut un une instance Bucket différente sur le driver et les exécuteurs
        s3_bucket_vgg16 = boto3.resource('s3').Bucket(BUCKET_NAME)
        s3_bucket_vgg16.download_file('input/' + nom_image, '/tmp/img')
        path_nom_image = '/tmp/img'
    else:
        path_nom_image = os.path.join(path, nom_image)
    # load the image (Pillow appelé par TF)
    image = load_img(path_nom_image, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)
    # get extracted features
    im_features = model.predict(image)
    # convert to list and add nom_image
    im_features = im_features[0].tolist()
    im_features.append(nom_image)
    return im_features
```

Map/Reduce sur toutes les images :

```python
# broadcasting
bc_model_vgg16 = sc.broadcast(model_vgg16)

# job spark
resultat = sc.parallelize(megabatch_img) \
    .map(lambda img: VGG16_extracteur('S3', img, bc_model_vgg16.value)) \
    .collect()
```

Export des données collectées vers un .csv :

	dim_0	dim_1	dim_2	...	dim_4093	dim_4094	dim_4095	path
0	0	0	0	...	0.258413106	1.812143803	0	Apple Braeburn/0_100.jpg
1	0	0	0	...	0	1.141268373	0	Apple Braeburn/100_100.jpg
2	0	0	0	...	0.117657334	0.927160382	0	Apple Braeburn/101_100.jpg
3	0	0	0	...	0.293886185	0.592740119	0	Apple Braeburn/102_100.jpg
...	...	...	...	...	...	...	...	...

Traitement des images sur un PC Linux en local avec PySpark

Résultats :

- fonctionnement de Spark en *--deploy-mode local* différent de celui observé en *--deploy-mode cluster*
- contrairement au mode **cluster (ressources en conteneur)**, en local les CPUs, la RAM et la Java VM sont partagés entre tous les exécuteurs.
- **résultats non significatifs par rapport à un cluster dans le cloud** avec des images stockées à proximité.
- en local, en se limitant à un seul exécuteur → **les temps d'I/O (S3)** sont supérieurs aux temps de calcul.
- en local, en augmentant le nb d'exécuteurs → performances boostées par partage des CPUs entre tâches Spark (pas possible en cluster).
- utilisation de **Spark session** pour transformer les données → lent et fréquentes erreurs de la Java VM.

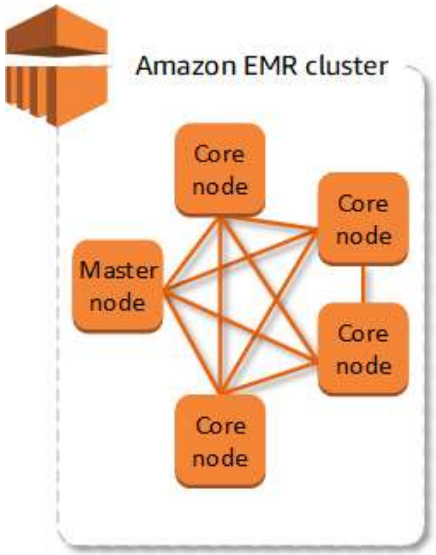
Conclusion :

- mode local utile pour **mettre au point le script PySpark.**
- **Spark session non retenue** pour le traitement dans le cloud.
- **tuning** des paramètres Spark **non extrapolable** au cluster.

Traitement des images dans le cloud (AWS EMR) avec PySpark

Présentation d'EMR (Elastic MapReduce) :

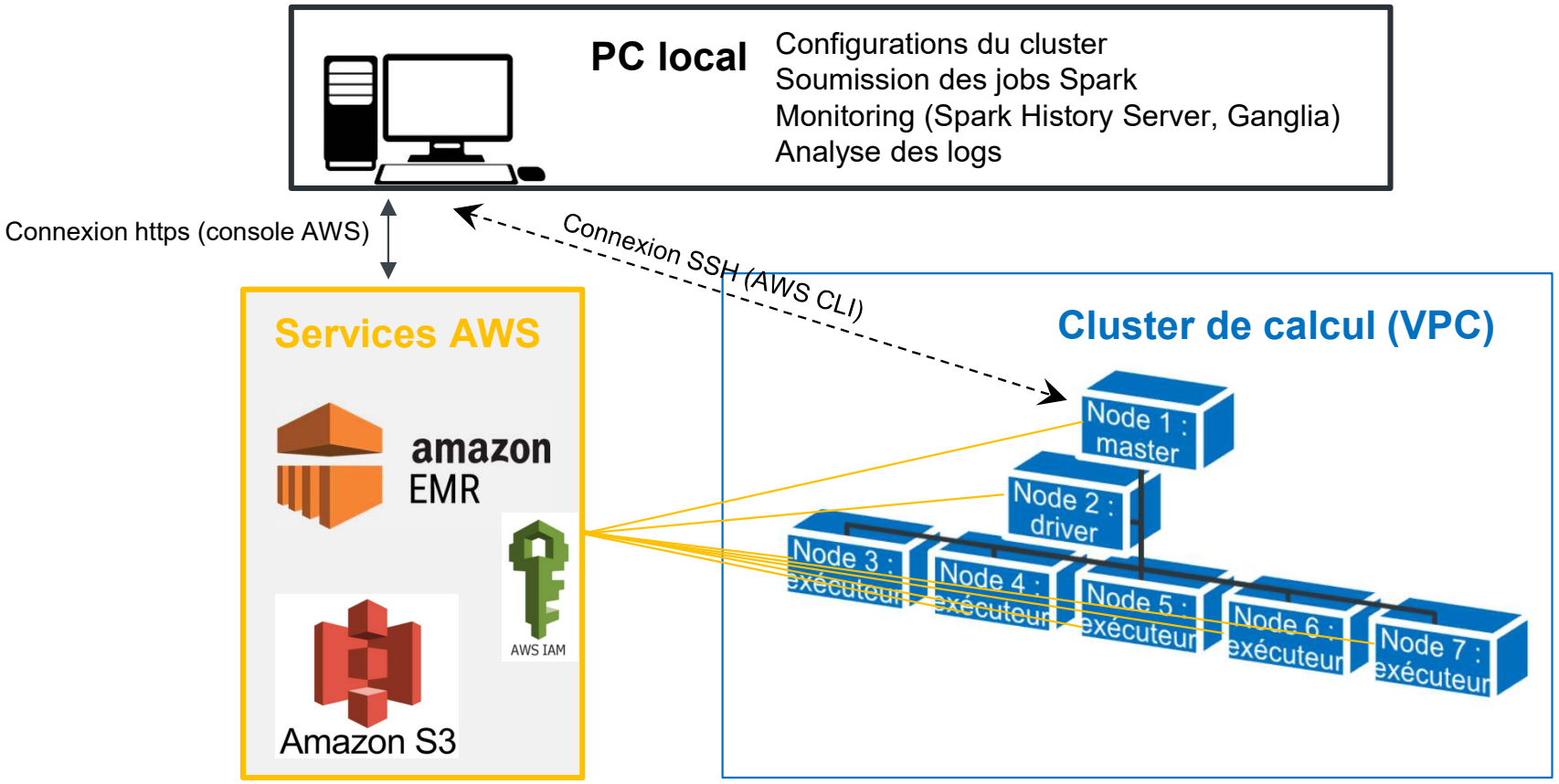
- plateforme de services pour le calcul distribué.
- simplifie l'utilisation des infrastructures de données massives, telles que Hadoop et Spark.
- évolutif à travers des instances EC2 dans le cloud AWS.



- cluster (composant principal) = ensemble de nodes (serveurs EC2), chacun ayant un rôle précis :
  - master node : effectue le suivi du statut des tâches et surveille l'état du cluster.
  - core node : exécutent les tâches dans un système de fichiers distribués (HDFS).
  - task node : facultatif, exécutent des tâches et ne stockent pas les données dans HDFS.

Traitement des images dans le cloud (AWS EMR) avec PySpark

Cluster retenu : 7 instances EC2 m5.xlarge (compromis entre : RAM / vCPUs / coût / limitations de mon compte AWS)



Traitement des images dans le cloud (AWS EMR) avec PySpark

Configuration finale du cluster :

- EMR release 6.5.0.
- 7 instances EC2 m5.xlarge (7*4 vCPUs, 7*16 Go RAM, 7*64Go SSD). 1 master et 6 cores.
- Boostraps : « pip install » boto3, pandas==1.2.5, tensorflow==2.7.0, Pillow
- Softwares pré-chargés : Hadoop v3.2.1, Spark 3.1.2, Ganglia 3.7.2.

Principaux écueils rencontrés avec EMR :

- Configuration initiale d'un cluster (certaines configurations par défaut proposées par AWS plantent !).
- Disponibilité des instances m5.xlarge (eu-west-1 > eu-west-3).

Principaux débuggages pour faire passer le code du local vers un cluster EMR :

- Affecter 10Go au conteneur du driver (au lieu des 2Go par défaut) pour lancer VGG16.
- Versionning des modules Python.
- Sérialisation du modèle VGG16 (*TypeError: can't pickle weakref objects*) : utiliser tensorflow >= 2.7.0

Traitement des images dans le cloud (AWS EMR) avec PySpark

Métriques du job Spark sur mini-clusters :

Worker nodes → Exécuteurs	vCPUs & RAM totale, par exécuteur	Nb partitions du RDD	Nb images	Durée job (s)	Commentaire
1 → 1	4 vCPUs & 16 Go	4 (défaut)	64	30	17s d'overhead au démarrage du job, puis 13s pour 64 images
			128	43	
		8 (forcé)	128	43	Pas d'impact temps du nb de partition
2 → 2		8 (défaut)	64	24	17s d'overhead au démarrage du job, puis 7s pour 64 images
			128	31	
2 → 4	2 vCPUs & 8 Go		128	32 à 52	« 6 tasks failed » en 7 jobs

→ Prévoir **16Go par exécuteur** pour éviter des « tasks failed »

Traitement des images dans le cloud (AWS EMR) avec PySpark

Job final (montée en échelle sur 4987 images) :

Worker nodes → Exécuteurs	vCPUs & RAM totale, par exécuteur	Nb images	Nb partitions du RDD	Durée job (s)	Commentaire
5 → 5	4 vCPUs & 16 Go	4987	20 (défaut)	224	2 répétitions (0 task failed)

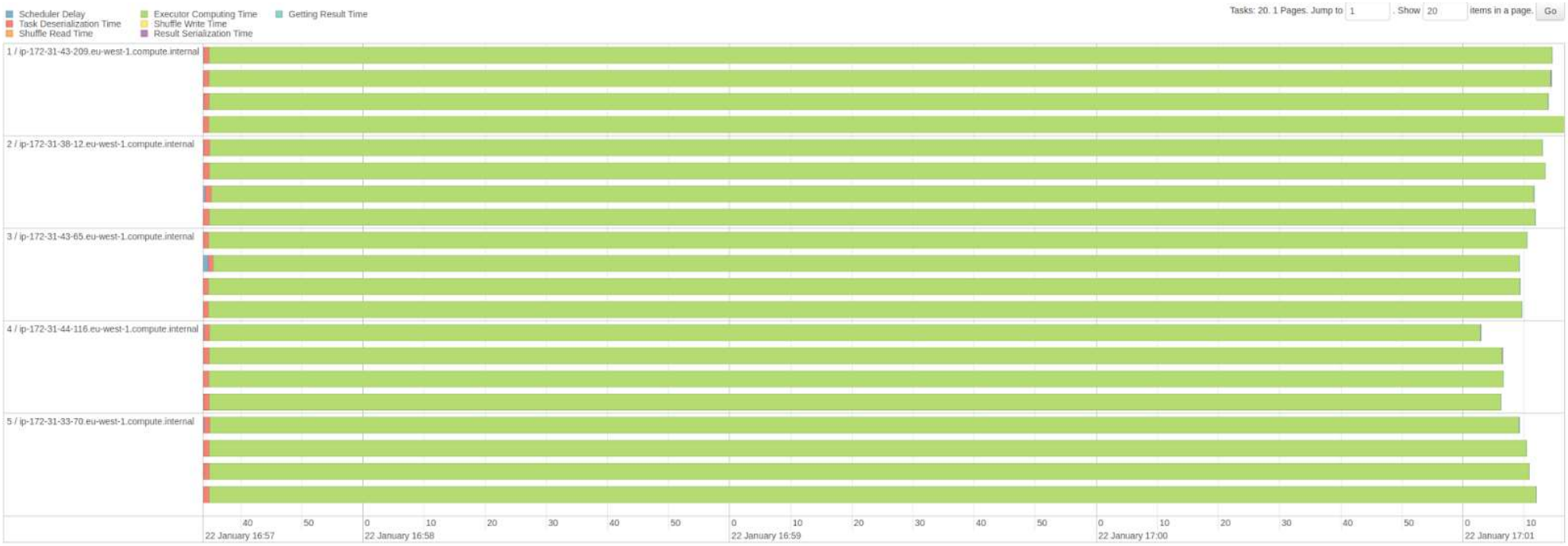
→ On reste **constant sur la vitesse de calcul par vCPU d'exécuteur** :
- 4100 images / (h.vCPU) sur le job du slide précédent (7s pour 64 images avec 8 vCPUs)
- 4400 images / (h.vCPU) sur le job de 4987 images en défalquant l'overhead de démarrage.

→ Augmentation du temps de calcul cohérente avec **variation linéaire :**
f(nb d'images, 1/(nb de vCPus exécuteur))

→ Taille du .csv de sortie du job : 161Mo.

Traitement des images dans le cloud (AWS EMR) avec PySpark

Job final (4987 images) - Spark History Server - event timeline :

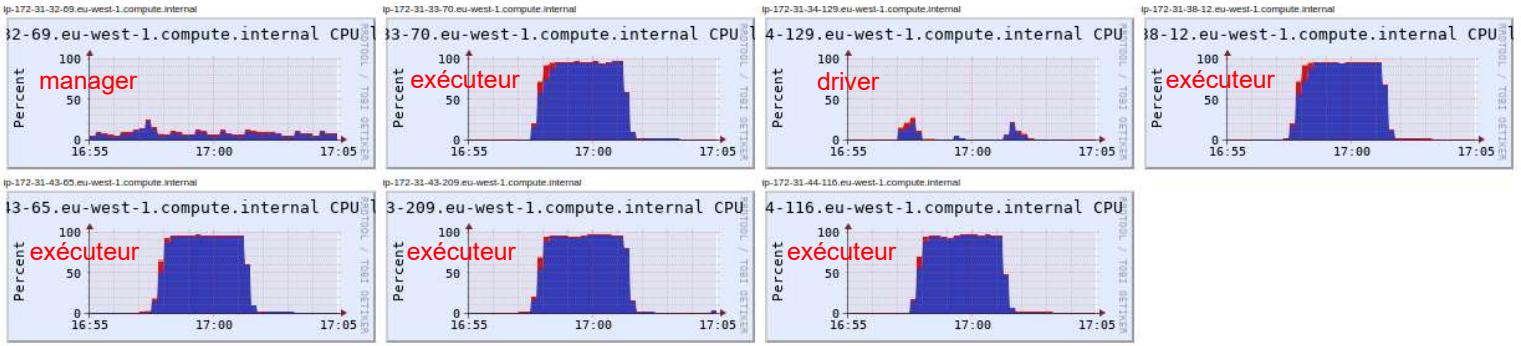


→ Peu de dispersion sur le temps de calcul des tâches
→ A surveiller : nb de partitions. Augmenter en cas de « task failed ».

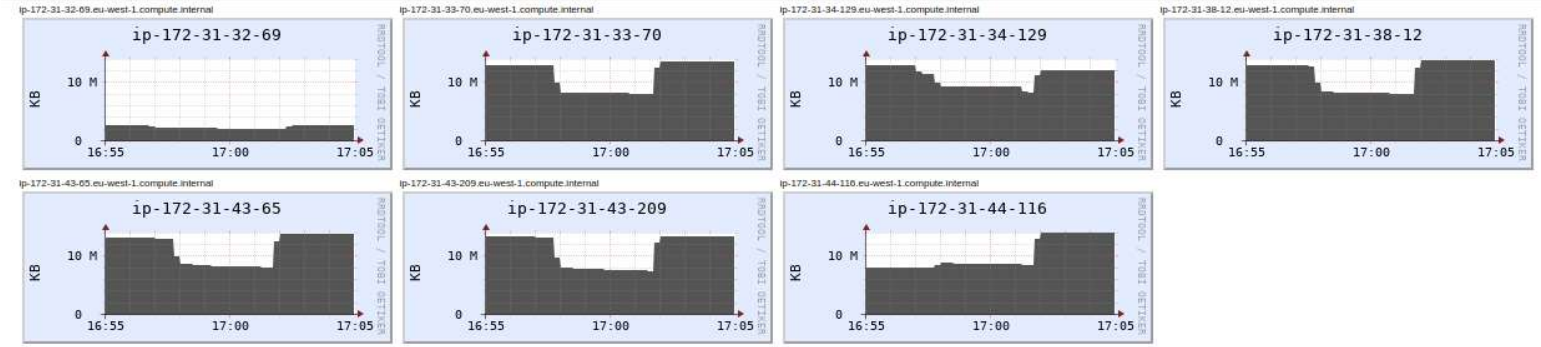
Traitement des images dans le cloud (AWS EMR) avec PySpark

Job final (4987 images) - monitoring Ganglia :

Charge des CPUs :



Charge des RAMs :



→ Possibilité de fusionner le manager et le driver sur un seul node.
→ CPUs des exécuteurs au max durant les jobs (peu d'idle dû aux I/O) → envisager p2.
→ RAM des exécuteurs : bien dimensionnée.
→ RAS sur le monitoring des I/O réseau.

Conclusion

POC obtenue **:**
- sur ~5000 images,
- avec un algorithme d'extraction basé sur VGG-16,
- dans le cloud AWS EMR,
- avec le framework Spark.

Augmentation du temps de calcul linéaire lors de la montée en échelle f(image, 1/(vCPUs))

Vitesse de calcul par vCPU d'exécuteur sur instance m5.xlarge :
- ~4400 images / (h.vCPU)
- 1.000.000 images pour 11€ (0.2€/h pour m5.xlarge).

Sur un plan personnel :
- acquisition de nouvelles compétences : Spark, PySpark, Boto3, AWS (en particulier S3, EMR).
- amélioration des compétences : Linux (tunnel SSH), Java VM.

Perspectives

Accélérer les calculs :

1) par le hardware : instances de calcul mieux adaptées telles que EC2 p2 (GPU).

2) par le code : solutions envisagées à l'instanciation de boto3 à chaque image :
- spark.read.format('Image'),
- pré-télécharger les images depuis S3 vers le VPC (task node de stockage).

Fiabiliser les calculs :

augmenter le nb de partitions du RDD.

Points durs pour une montée en échelle :

taille du fichier .csv de sortie trop gros si unique → générer un fichier .csv par fruit.

Tester images plus grande que 100*100 pixels :

code et cluster conçus pour des images de taille plus grande.

Scripts finaux

Création du cluster depuis AWS CLI :

```
aws emr create-cluster --applications Name=Hadoop Name=Hive Name=Spark Name=Ganglia --ec2-attributes '{"KeyName":"cle_irelande","InstanceProfile":"EMR_EC2_DefaultRole","SubnetI
d":"subnet-00bbdbcf537bf57b5","EmrManagedSlaveSecurityGroup":"sg-08a05f6eec6c610ac","EmrManagedMasterSecurityGroup":"sg-09d0c339df01417fd"}' --release-label emr-6.5.0 --log-uri 's
3n://aws-logs-327946743066-eu-west-1/elasticmapreduce/' --instance-groups '[{"InstanceCount":6,"EbsConfiguration":{"EbsBlockDeviceConfigs":[{"VolumeSpecification":{"SizeInGB":3
2,"VolumeType":"gp2"},"VolumesPerInstance":2}]},"InstanceGroupType":"CORE","InstanceType":"m5.xlarge","Name":"Core - 2"},{"InstanceCount":1,"EbsConfiguration":{"EbsBlockDeviceConf
igs":[{"VolumeSpecification":{"SizeInGB":32,"VolumeType":"gp2"},"VolumesPerInstance":2}]},"InstanceGroupType":"MASTER","InstanceType":"m5.xlarge","Name":"Master - 1"}]' --auto-sca
ling-role EMR_AutoScaling_DefaultRole --bootstrap-actions '[{"Path":"s3://projet8-oc/scripts-pyspark/emr_boostrap_v5.sh","Name":"Custom action"}]' --ebs-root-volume-size 100 --ser
vice-role EMR_DefaultRole --enable-debugging --auto-termination-policy '{"IdleTimeout":14400}' --name 'Cluster pour Projet 8' --scale-down-behavior TERMINATE_AT_TASK_COMPLETION --
region eu-west-1
```

Script bash du boostrap :

```
#!/bin/bash
sudo python3 -m pip install boto3 pandas==1.2.5 tensorflow==2.7.0 Pillow==8.4.0
```

Script de lancement du job PySpark depuis la CLI du node master :

```
spark-submit --deploy-mode cluster --driver-memory 10g s3://projet8-oc/scripts-pyspark/extraction_features_v5.1.py
```

Code du driver

```python
# choix de la zone selon disponibilité :
BUCKET_NAME = "projet8-oc"   # eu-west-1

# imports modules
import os
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.models import Model
from pyspark import SparkContext
import boto3
from datetime import datetime
from time import time
import pandas as pd

def timestamp():
    return datetime.now().strftime("%Y-%m-%d--%H-%M-%S")

def logger_s3(s3_bucket, logfile, log, log_to_shell=True):
    """Appends log to logfile. Uploads each update to S3 bucket.
    Args :
        - s3_bucket (Bucket).
        - logfile (string) .
        - log (string).
        - log_to_shell (bool) : si True, imprime également dans le std.out.
    Returns : /
    """
    time_log = timestamp() + " : " + log + "\n"
    if log_to_shell : print("*"*100 + "\n", time_log)
    with open("./logs/"+ logfile, 'a') as f:
        f.write(time_log)
    s3_bucket.upload_file("./logs/"+ logfile, 'logs/' + logfile)

# fonction d'extraction de feature
def VGG16_extracteur_spark(path, nom_image, model):
    """Transforme un fichier image en un vecteur de dimension 4096.
    Args :
    - path : chemin vers les images (chemin local ou 'S3').
    - nom_image : chemin d'accès à l'image (exemple : 'Apple Braeburn/r_173_100.jpg')
    - model : model_vgg16 ou bc_model_vgg16.
    Returns :
    - liste de dimension 4097 (4096 dimensions de VGG16 + nom_image).
    """
    # create path to the image stored locally
    if path == 'S3':
        # il faut un une instance Bucket différente sur le driver et les exécuteurs
        s3_bucket_vgg16 = boto3.resource('s3').Bucket(BUCKET_NAME)
        s3_bucket_vgg16.download_file('input/' + nom_image, '/tmp/img')
        path_nom_image = '/tmp/img'
    else:
        path_nom_image = os.path.join(path, nom_image)
    # load the image for keras processing
    image = load_img(path_nom_image, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)
    # get extracted features
    im_features = model.predict(image)
    # convert to list and add nom_image
    im_features = im_features[0].tolist()
    im_features.append(nom_image)
    return im_features
```

```python
# start logging
os.makedirs(os.path.join(os.getcwd(), 'logs/'), exist_ok=True)
logfile = "Log for job started " + timestamp() + ".txt"
logger_s3(s3_bucket, logfile, "logging starts")

# liste des fruits dans le répertoire "input/" : megabatch_img
all_fruits = s3_bucket.objects.filter(Prefix="input")
megabatch_img = []
for obj in all_fruits:
    megabatch_img.append(obj.key.lstrip("input/"))
del megabatch_img[0]  # suppression d'un objet non pertinent

# load model VGG16
model_vgg16 = VGG16()
# remove the output layer
model_vgg16 = Model(inputs=model_vgg16.inputs, outputs=model_vgg16.layers[-2].output)

# Spark context : le SparkConf est supprimé car géré via le spark-submit
sc = SparkContext()

# logger des paramètres du spark context lancé
logger_s3(s3_bucket, logfile, "Configuration de Spark :" + str(sc.getConf().getAll()))

# broadcasting
bc_model_vgg16 = sc.broadcast(model_vgg16)

# logging
logger_s3(s3_bucket, logfile, "spark job starts")
t0 = time()

# job spark
resultat = sc.parallelize(megabatch_img) \
    .map(lambda img: VGG16_extracteur_spark('S3', img, bc_model_vgg16.value)) \
    .collect()

# logging
logger_s3(s3_bucket, logfile, f"spark job has ended (duration of {round(time() - t0, 1)}s) - output to S3 starts")

# output
df_output = pd.DataFrame(resultat)
df_output.columns = [f'dim_{i}' for i in range(4096)] + ['path']
file_name = f'df_output_{len(df_output)}_fruits.csv'
local_path = os.path.join(os.getcwd(), file_name)
df_output.to_csv(path_or_buf=local_path)
s3_bucket.upload_file(local_path, 'output/' + file_name)

# end logging
logger_s3(s3_bucket, logfile, "output to S3 has ended")
```