Déployez un modèle dans le cloud

Sommaire

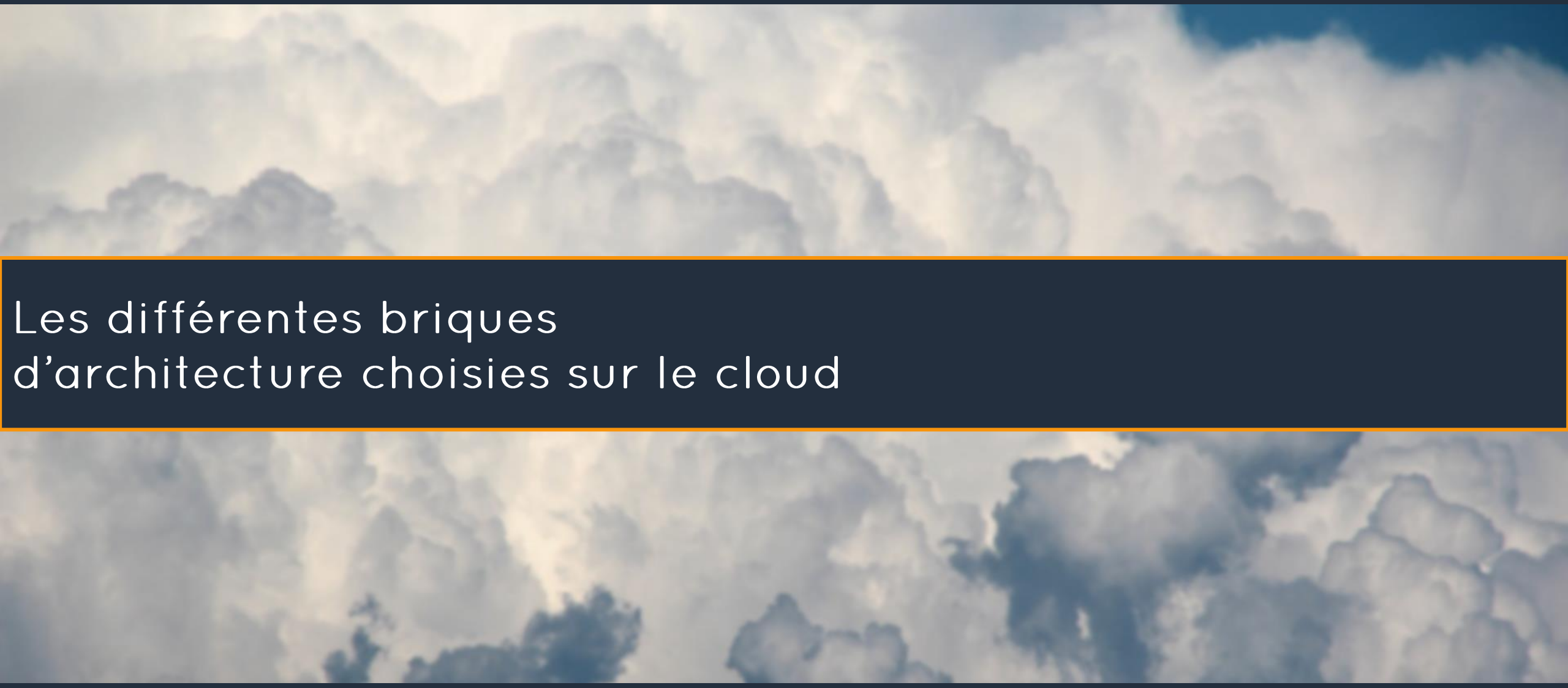
Les différentes briques d'architecture choisies sur le cloud

Leur rôle dans l'architecture Big Data

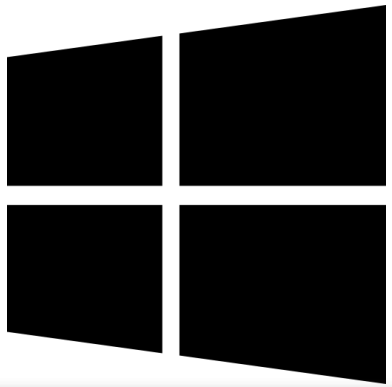
- Définition du Big Data
- Adaptabilité de la structure choisie (mise à l'échelle)

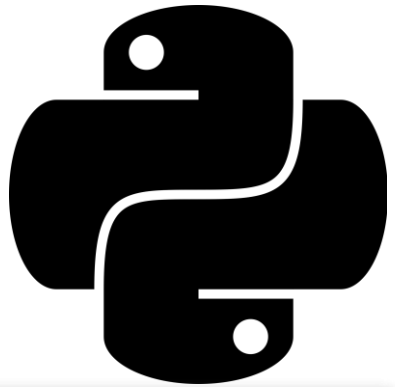
Les étapes de la chaine de traitement

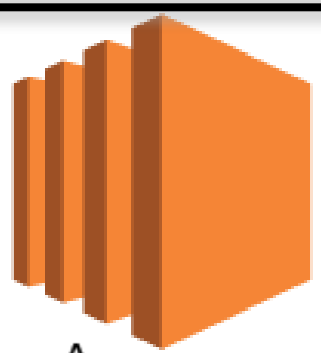
- Redimensionnement des images
- Chargement des données dans un DF PySpark
- Vectorisation et réduction de dimensions
- Mise en place d'un réseau de neurone avec Transfer Learning

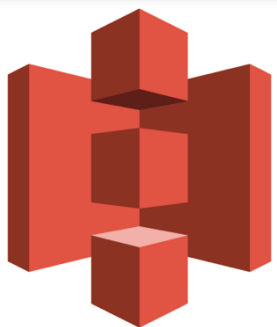


Les différentes briques d'architecture choisies sur le cloud






Amazon
EC2


Amazon S3


ubuntu


APACHE
Spark

Amazon Elastic Cloud Computing

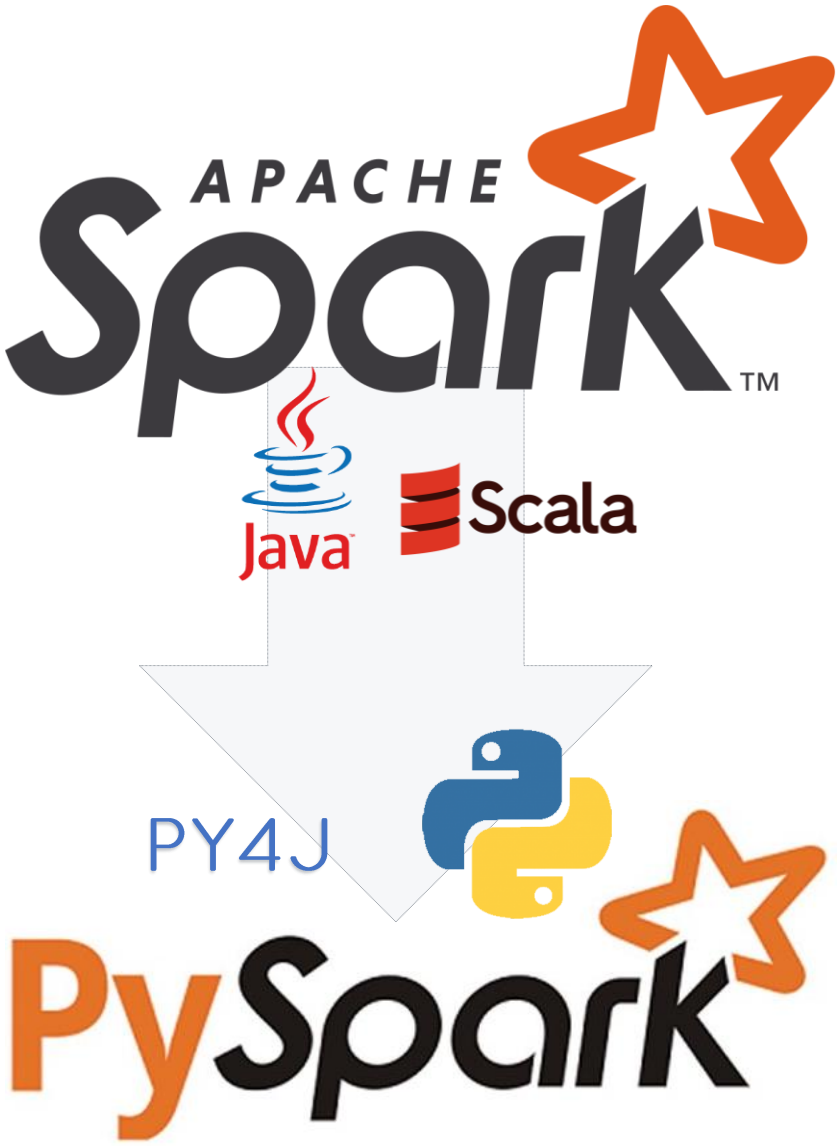
EC2 est un service qui met à disposition un outil de calcul à **capacité évolutive** adapté à nos besoins.

Celui-ci peut **automatiquement** augmenter sa puissance de calcul si nécessaire, afin de **gérer une surcharge d'activité** temporaire (nombre de requête pour un site web ou besoin de puissance de calcul pour une application), ou la diminuer dans le cas contraire, ce afin de **minimiser les coûts**. (pay as you go)

Amazon
Simple Storage Service

Est un outil de stockage à **capacité évolutive** ayant des fonctionnalités de classement, d'accessibilité avancées, de versionning...

L'objet principal est un **bucket** (sceau) dans lequel on place nos fichiers. Le sceau n'a **pas de limite** de taille ou de restriction concernant le nombre d'objet qu'il peut contenir.
On peut accéder aux buckets via les autres services Amazon, notament les instances EC2.

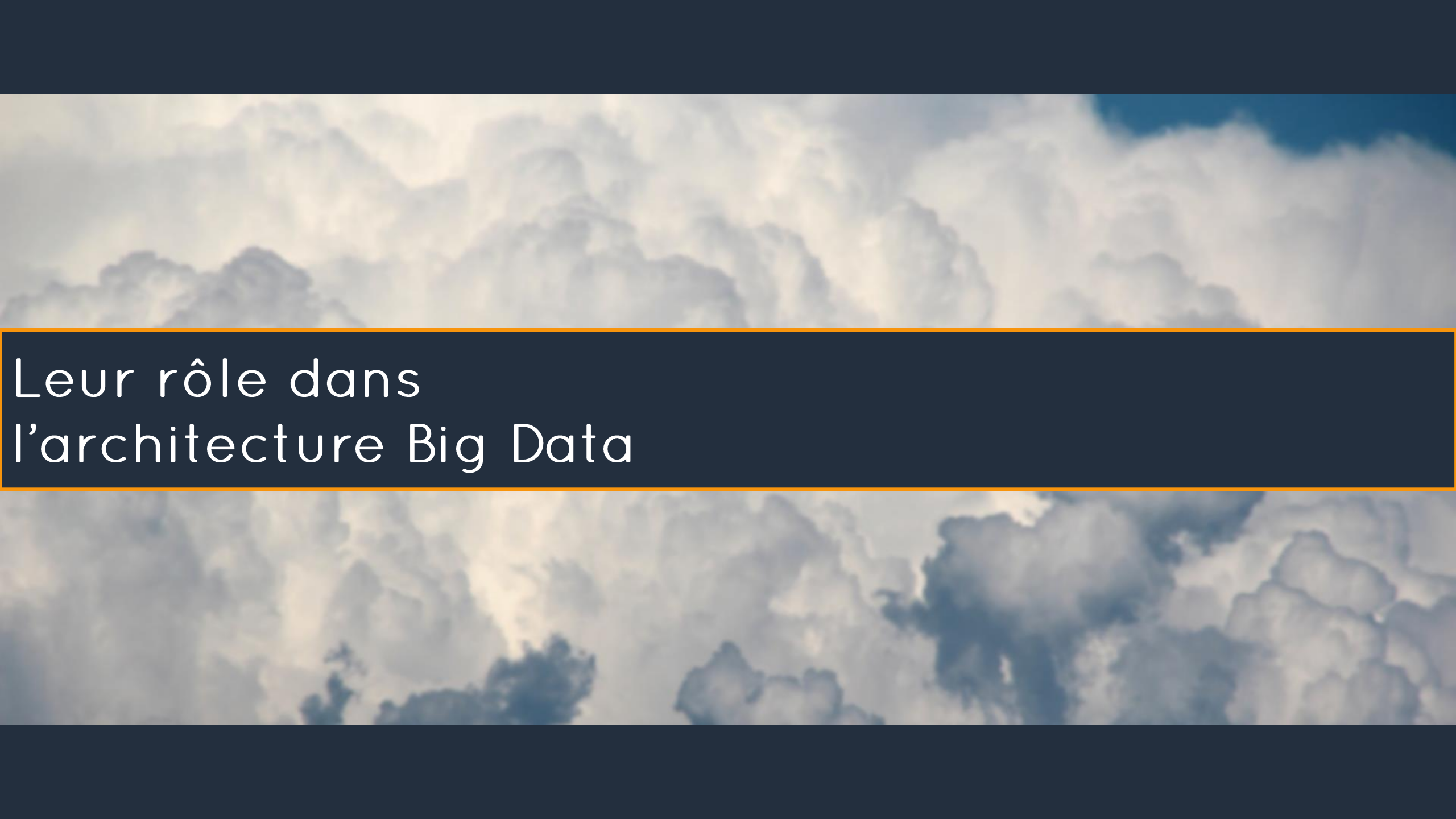


SPARK

est un logiciel **open source** qui permet de faire du **calcul distribué** afin de permettre de l'analyse de données à **grande échelle**.

PYSPARK

est l'**API PYTHON** pour SPARK, dont le langage natif est SCALA, qui met à disposition des **librairies compatibles** avec ce langage.



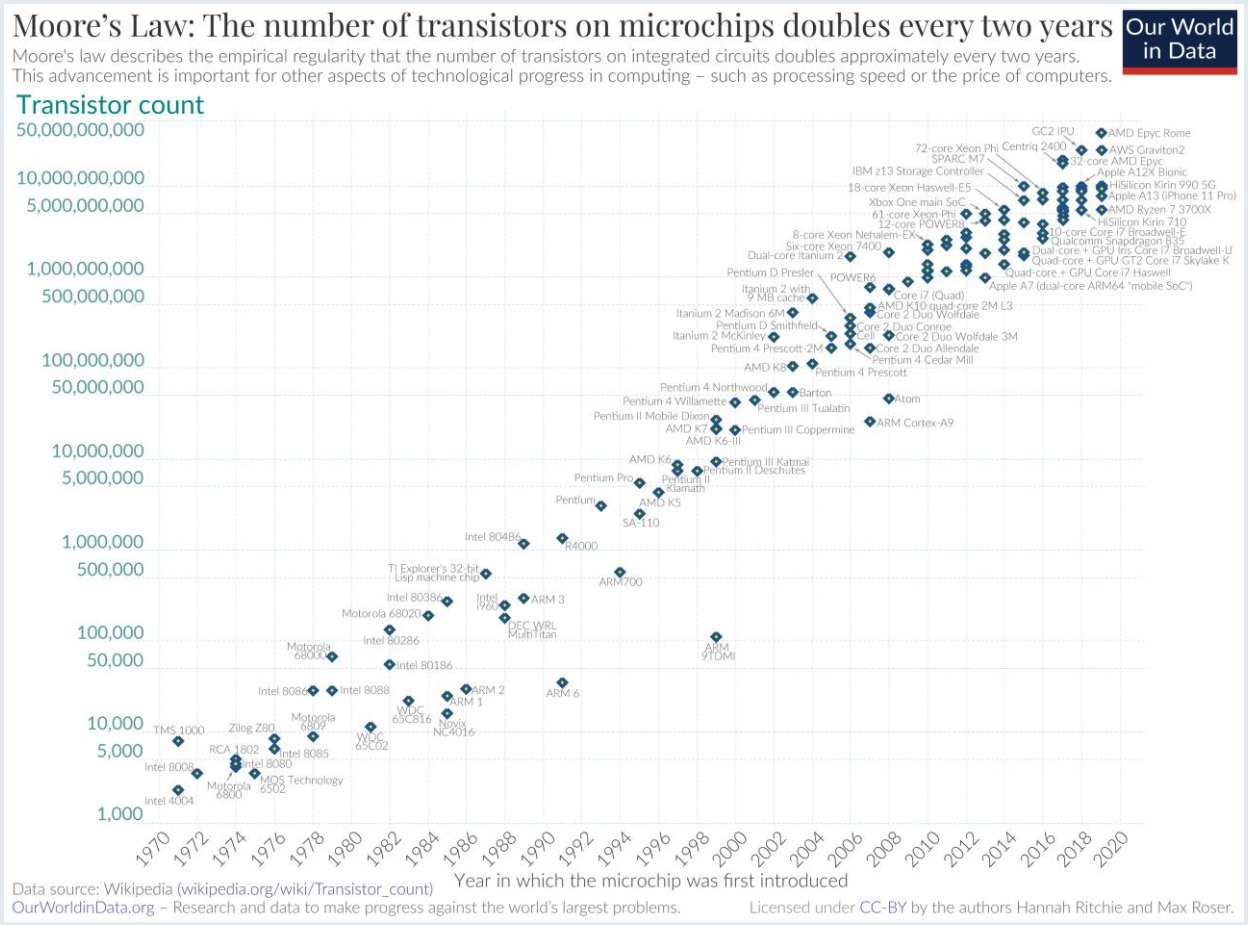
Leur rôle dans l'architecture Big Data

Qu'est ce que le Big data ?

Traditionnellement, le Big Data fait référence à des **jeux de données trop grands** pour être pris en charge par un **ordinateur seul**, car demandant trop de **puissance de calcul** pour être manipulé.

Cependant, au vu de la multiplication de la puissance de calcul des ordinateurs, cette **notion est mouvante**.
La plupart des ordinateurs d'aujourd'hui peuvent tout à fait manipuler un jeu de donnée de 1G, ce qui n'était pas le cas il y a 10 ans.

Aussi, en 2018, une nouvelle définition se réfère plutôt à l'architecture disant que **relève du Big Data les problématiques qui ne peuvent se résoudre que par des outils de parallélisation informatiques** (parallel computing tools).



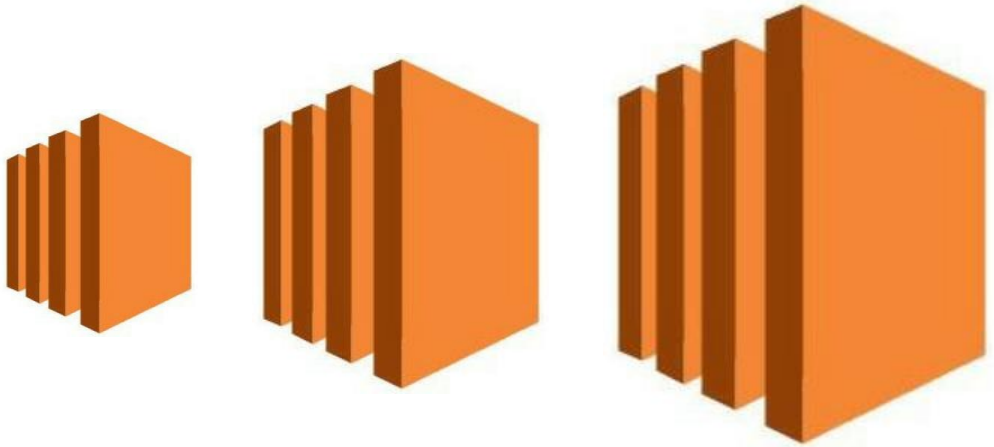
La loi de Moore relatant du **doublement** de la puissance de calcul des ordinateurs tous les **2 ans**.

Adaptabilité de la structure choisie (mise à l'échelle)

Comme évoqué précédemment, les outils AWS utilisés sont à **capacité évolutive**.

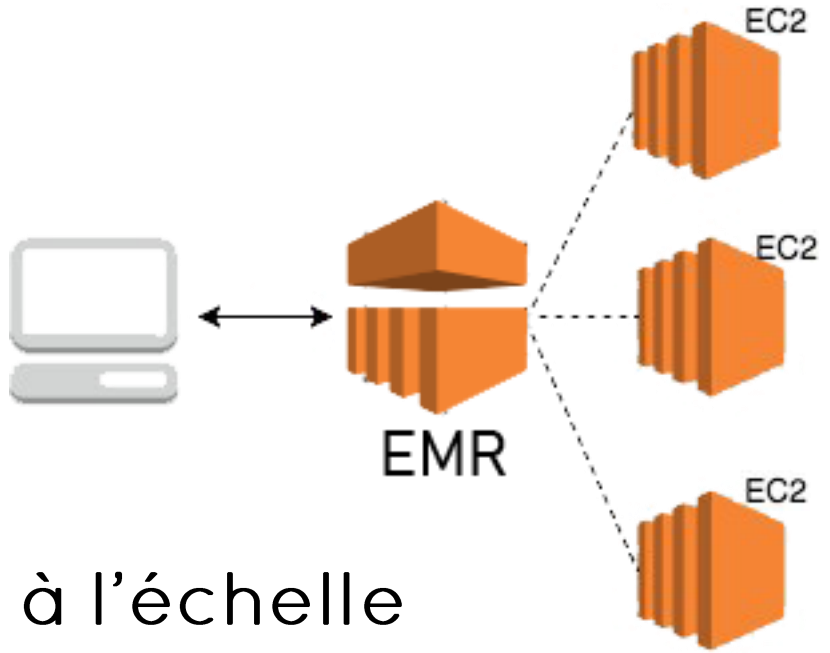
On a vu que les sceaux **S3 sont sans limite** et adaptent leur taille à leur contenu.

Pour les **instances EC2**, c'est également le cas. Pour augmenter les capacités de calcul de notre machine, il y a au moins **deux approches possibles**.



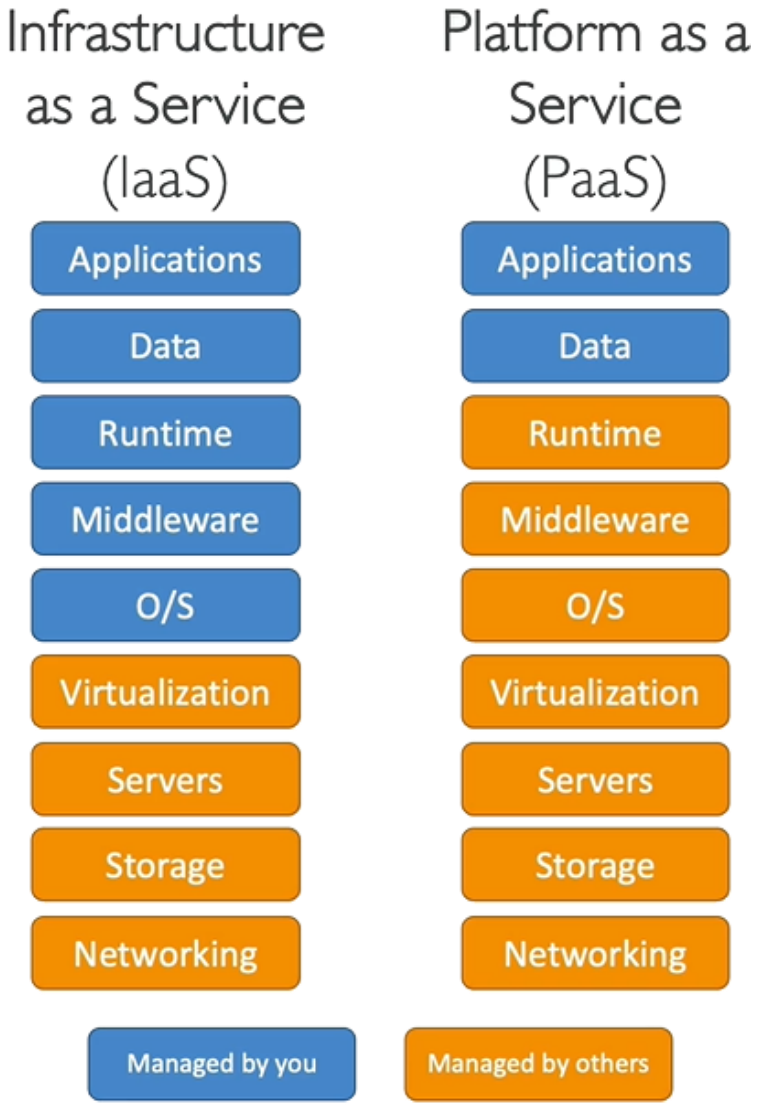
Mise à l'échelle verticale

Dans cette approche on choisie une machine + ou – performante en fonction de nos besoins.



Mise à l'échelle horizontale

Création d'une nouvelle structure avec un **EMR** (ElasticMapReduction) qui gère des instances EC2 (Auto Scaling et équilibrage de charges)



Différence entre les deux architectures :

EMR est un PaaS, et donc une architecture basée sur ce service sera moins paramétrable.
Cela est particulièrement vrai si le cluster est amené à durer dans le temps et qu'il faut **mettre à jour** l'O/S (Ubuntu) ou un des middleware (Java).

EMR bénéficie depuis juin 2020 et EMR 5.30 de l'**Auto Scaling**, ainsi il est possible d'adapter la **taille** du système choisis en **fonction des besoins**.

Les étapes de la chaîne de traitement

La problématique

Fruits!

Fruits
est une start-up qui veut dans un premier temps mettre au point un **moteur de classification** des images de fruits.

Notre mission est de développer une première chaine de traitement comprenant le **preprocessing**
et une étape de
réduction de dimensions
en
PYSPARK.

Les données

kaggle

Le jeu de données est en accès libre sur Kaggle.

Fruits 360

Il comprend **90483** images réparties comme tel :

- un jeu d'entrainement de 67692 images
- un jeu de test de 22688 images

Il est divisé en **131 classes**.

Taille des images : **100x100 px**.

Redimensionnement des images

L'idée est de préparer les données pour l'algorithme **VGG-16**.

Pour ce faire, nous devons les **redimensionner** puisque cet algorithme accepte en entrée des vecteurs d'images de **224x224 px**.

Nous utilisons **OPEN-CV**.

Comme nous utilisons l'algorithme en Transfer Learning (en gardant les poids pré-entrainé), nous n'avons pas besoin de faire de **Data-augmentation**.

```
: image_resized.size
: (224, 224)
```

```
image_originale.size
```

```
: image_resized
```

```
(100, 100)
```

```
:
```

```
image_originale
```

Chargement des données dans un dataframe PySpark

Contrairement au Dataframe Pandas, le Dataframe PySpark supporte les **calculs distribués**.

Afin de limiter les coûts de développement, nous sélectionnons un **échantillon** des données comprenant **238 images de 6 fruits** différents.

Les images précédemment redimensionnées sont chargées avec succès dans le **DF PySpark**.

```
path = new_train_directory +'/'+ train_names[0]
df = spark.read.format('image').load(path)
df = df.withColumn('path', input_file_name())
df = df.withColumn('name', lit(train_names[0]))
for i in range(len(train_names)-1):
    path = new_train_directory +'/'+ train_names[i+1]
    temp = spark.read.format('image').load(path)
    temp = temp.withColumn('path', input_file_name())
    temp = temp.withColumn('name', lit(train_names[i+1]))
    df = df.union(temp)
```

```
df.count(), len(df.columns)
```

```
(238, 3)
```

```
df.printSchema()
```

```
root
 |-- image: struct (nullable = true)
 |    |-- origin: string (nullable = true)
 |    |-- height: integer (nullable = true)
 |    |-- width: integer (nullable = true)
 |    |-- nChannels: integer (nullable = true)
 |    |-- mode: integer (nullable = true)
 |    |-- data: binary (nullable = true)
 |-- path: string (nullable = false)
 |-- name: string (nullable = false)
```

```
df.show(5)
```

```
+--------------------+--------------------+----+
|               image|                path|name|
+--------------------+--------------------+----+
|{file:///home/pys...|file:///home/pysp...|Pear|
|{file:///home/pys...|file:///home/pysp...|Pear|
|{file:///home/pys...|file:///home/pysp...|Pear|
|{file:///home/pys...|file:///home/pysp...|Pear|
|{file:///home/pys...|file:///home/pysp...|Pear|
+--------------------+--------------------+----+
only showing top 5 rows
```

Vectorisation et réduction de dimension

Comme exposé précédemment, l'algorithme **VGG-16** prend en entrée des **vecteurs issues d'images**.

Ainsi, nous devons convertir les images redimensionnées en **vecteurs**.

Pour ce faire, nous utilisons les fonctions **DenseVector** et **VectorUDT** de la **librairie ML** de PySpark.

Nous en profitons pour faire une réduction de dimensions via une **analyse des composantes principales** depuis la même librairie PySpark ML.

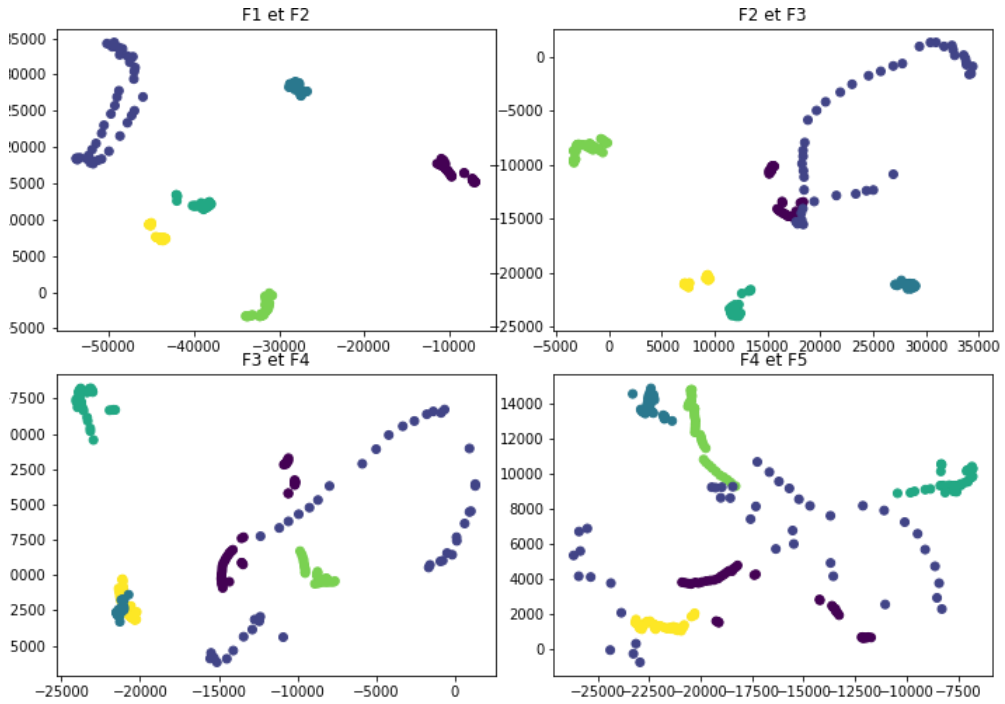
```
ImageSchema.imageFields
['origin', 'height', 'width', 'nChannels', 'mode', 'data']

img2vec = F.udf(lambda x: DenseVector(ImageSchema.toNDArray(x).flatten()), VectorUDT())

df = df.withColumn('vecs', img2vec("image"))

df.show(5)
```

```
+--------------------+--------------------+----+--------------------+
|               image|                path|name|                vecs|
+--------------------+--------------------+----+--------------------+
|{file:///home/pys...|file:///home/pysp...|Pear|[255.0,255.0,255....|
|{file:///home/pys...|file:///home/pysp...|Pear|[255.0,255.0,255....|
|{file:///home/pys...|file:///home/pysp...|Pear|[255.0,255.0,255....|
|{file:///home/pys...|file:///home/pysp...|Pear|[255.0,255.0,255....|
|{file:///home/pys...|file:///home/pysp...|Pear|[255.0,255.0,255....|
+--------------------+--------------------+----+--------------------+
only showing top 5 rows
```



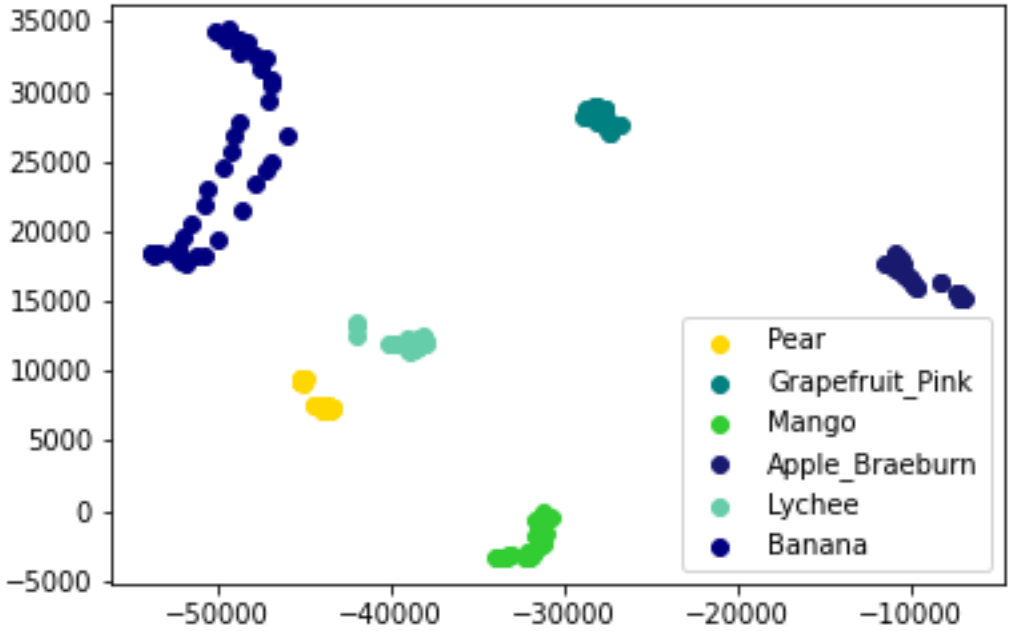
Représentation graphique sur le premier plan factoriel

La variance s'explique à
74% par les plans **factoriels 1 et 2**.

On voit sur le graphique ci-contre que l'analyse des composantes principales a permis de clairement **distinguer les catégories de fruits** présentées.

On en déduit que la **vectorisation** s'est bien passée.

Représentation graphique sur F1 et F2




```
model.explainedVariance
```

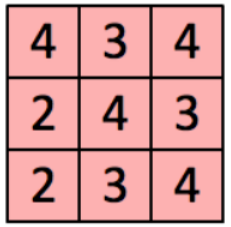
```
DenseVector([0.4609, 0.2789, 0.1261, 0.0739, 0.0601])
```

Mise en place d'un CNN avec Transfer Learning

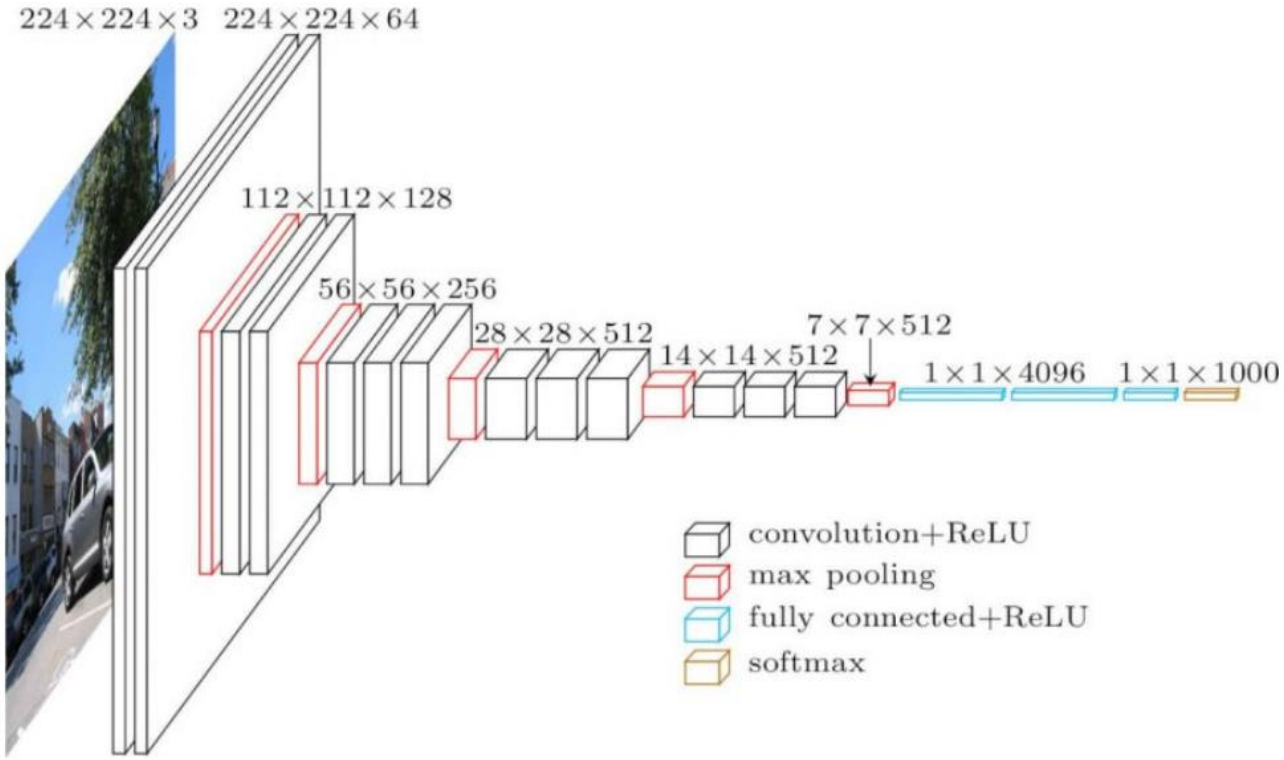
Nous allons utiliser le VGG-16 avec Transfer Learning.

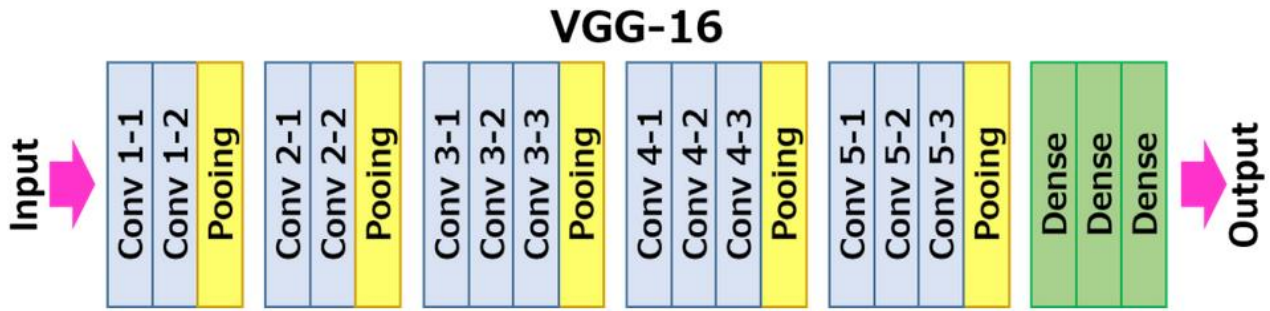
Cet algorithme initialement présenté en 2014 est encore une référence aujourd'hui.

Image



Convolved Feature





Nous importons le modèle avec **les poids entrainés** et non ré-entrainable sans la dernière couche de neurones.

Nous rajoutons une **dernière couche** avec une fonction **'Softmax'** et autant de sorties possibles que nous avons de catégories.

Cette dernière couche est prête à être entrainée.

```
: cnn_tl.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
flatten (Flatten)            (None, 25088)             0
dense (Dense)                (None, 6)                 150534
=================================================================
Total params: 14,865,222
Trainable params: 150,534
Non-trainable params: 14,714,688
_________________________________________________________________
```

CONCLUSION

Les solutions proposées par AWS permettent de mettre en place des architectures Big Data de manière **relativement simples**.

De plus la dynamique de ces environnements, **en constante évolution** augure des fonctionnalités à la fois plus complètes et plus simples.

Sachant que le **Big Data englobe de plus en plus de champs**, ces outils vont s'avérer de plus en plus **indispensables** dans un futur proche.