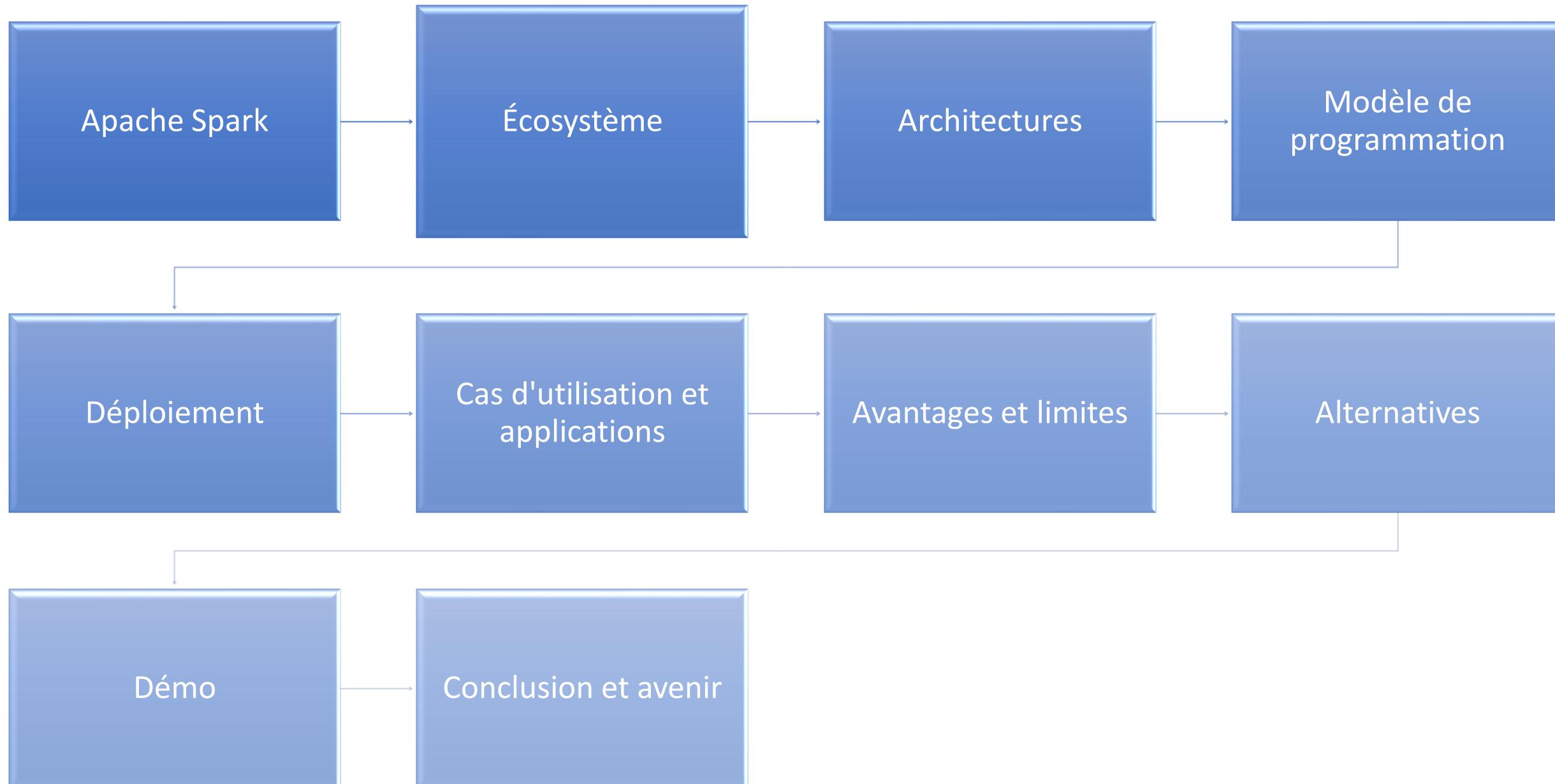
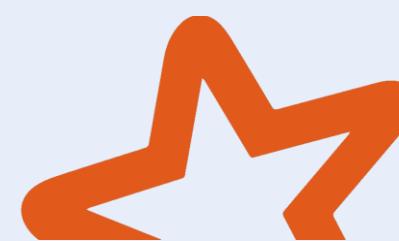




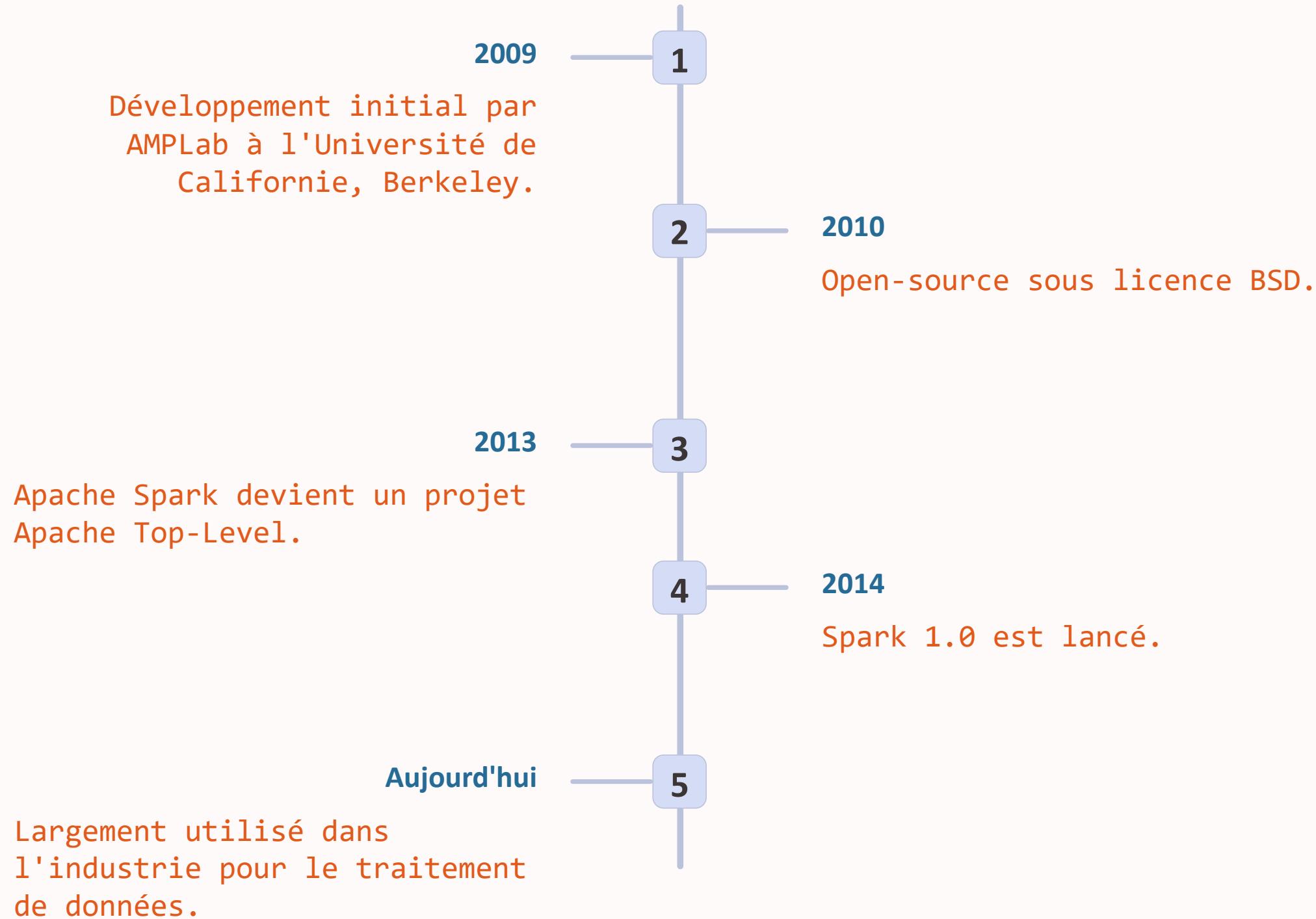
Par : Mouhamadou Adji BARRY, DIT/Atos

Encadreur : Ibrahima Oumar LY, Data Engineer





# Histoire et Évolution d'Apache Spark

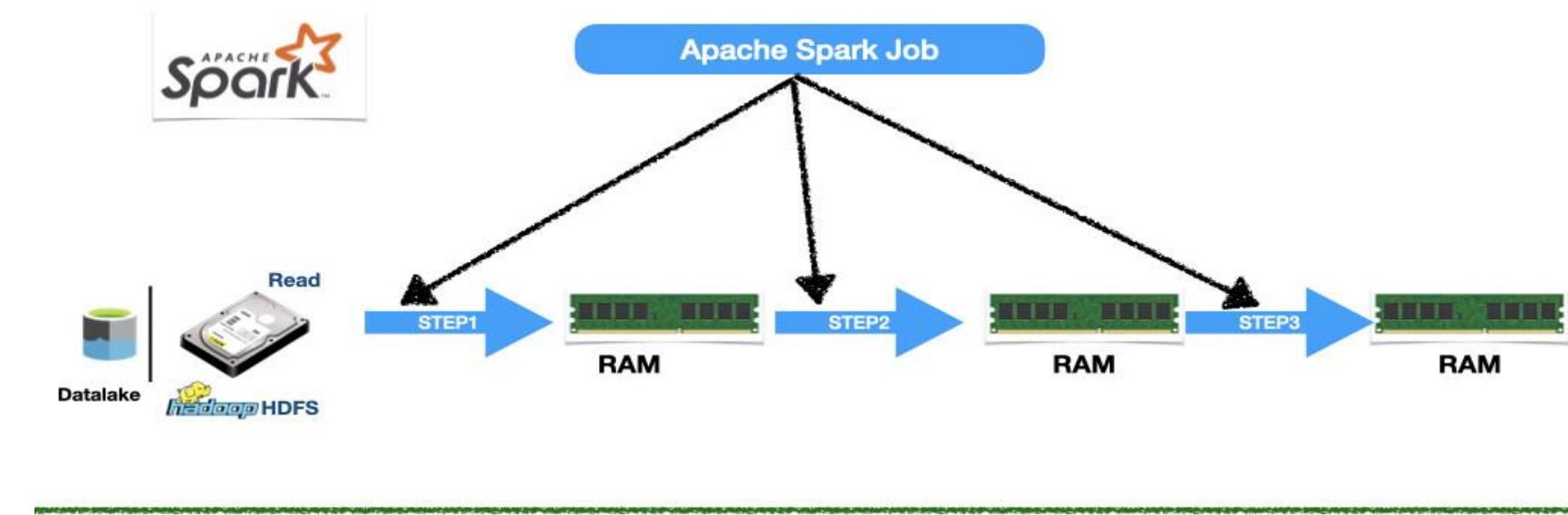




# Caractéristiques Clés d'Apache Spark

## 1 Rapidité

Traitements en mémoire très rapides.

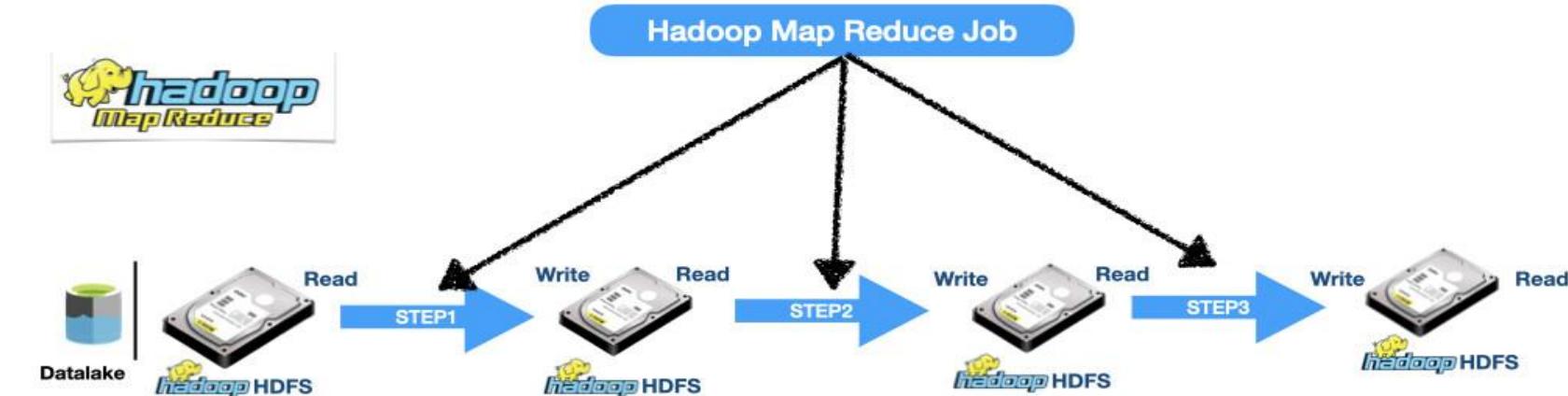


## 3 Facilité d'Utilisation

Large gamme d'applications (batch, streaming, ML, graphes).

## 4 Écosystème Complet

Intégration facile avec Hadoop, Cassandra, HBase, etc.





# L'Écosystème de Spark

## Spark SQL

Traitements de données structurées

## Spark Streaming

Traitements de flux de données

## GraphX

Traitements de graphes

## Mlib

Bibliothèque de machine learning

## Spark Core

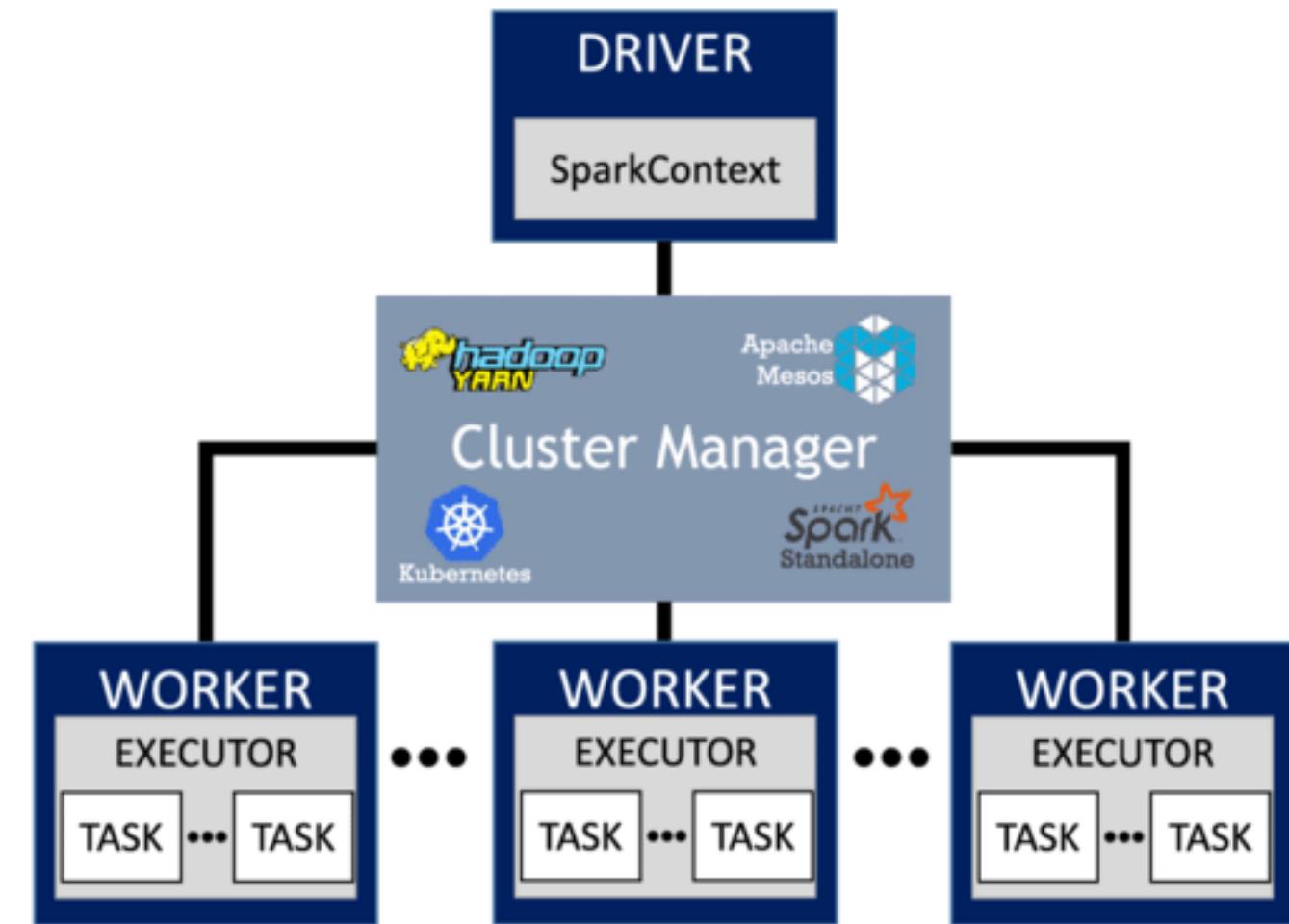
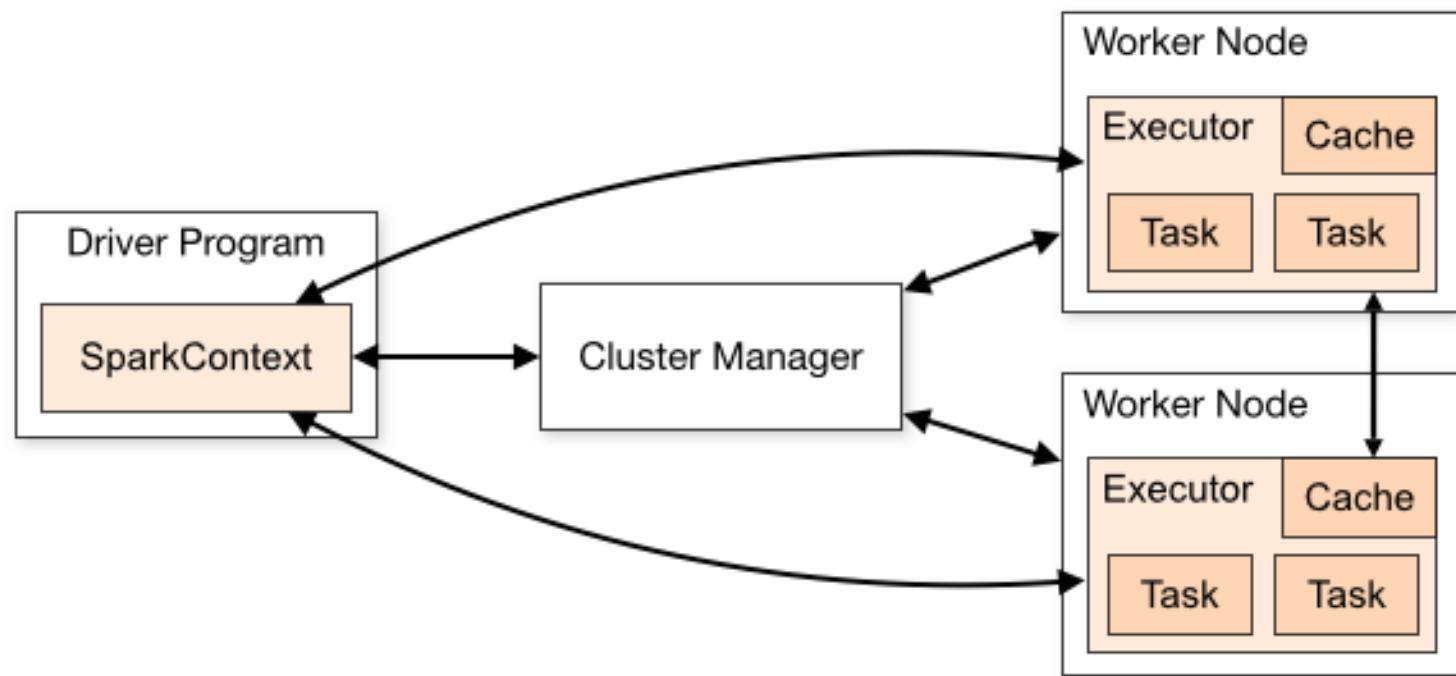
Moteur d'exécution





# Architectures de Spark

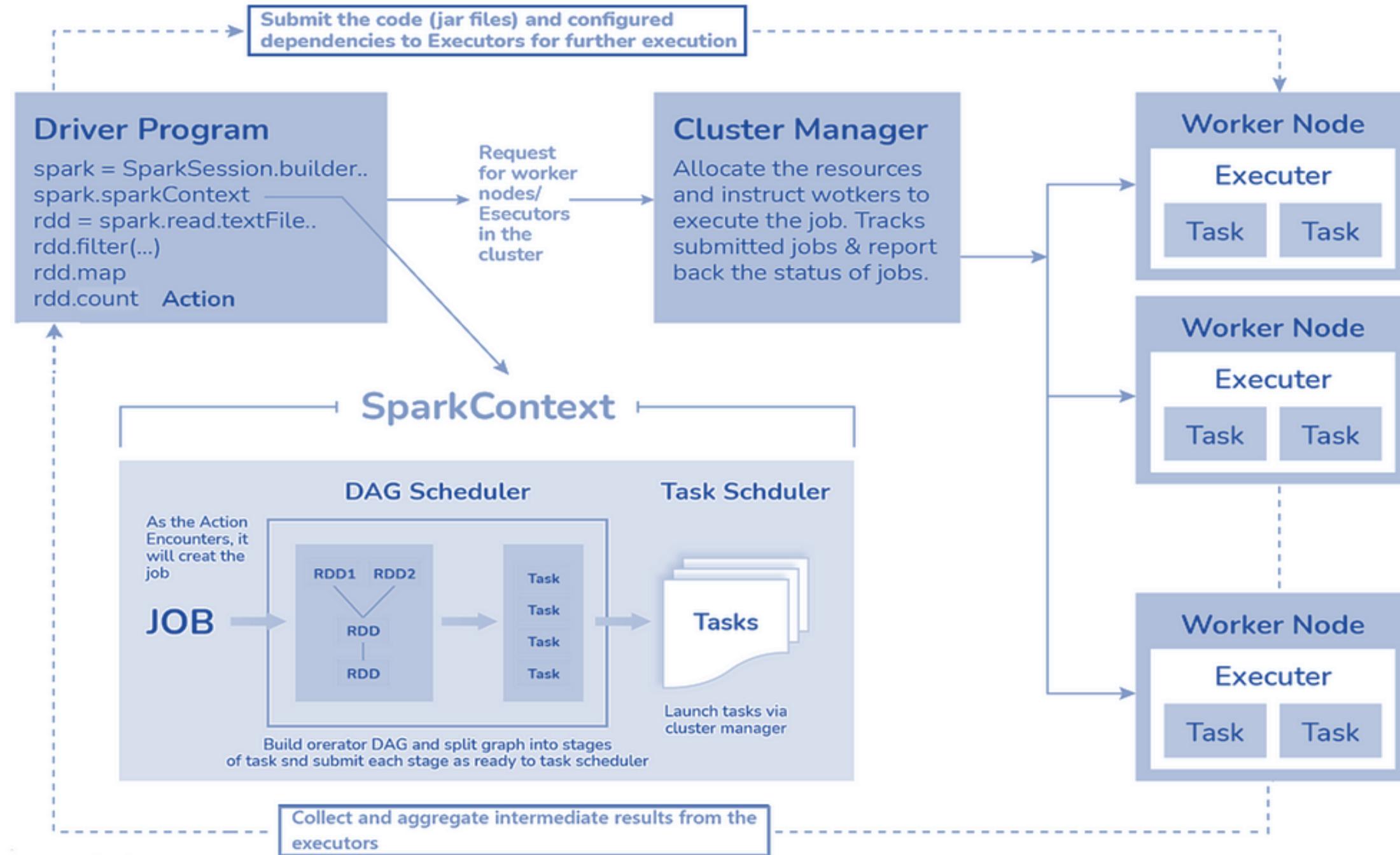
## Composants

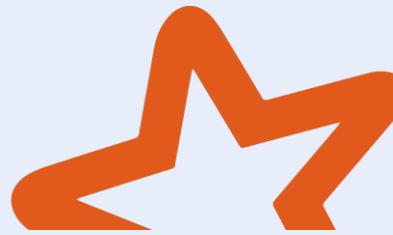




# Architectures de Spark

## Flux d'Exécution

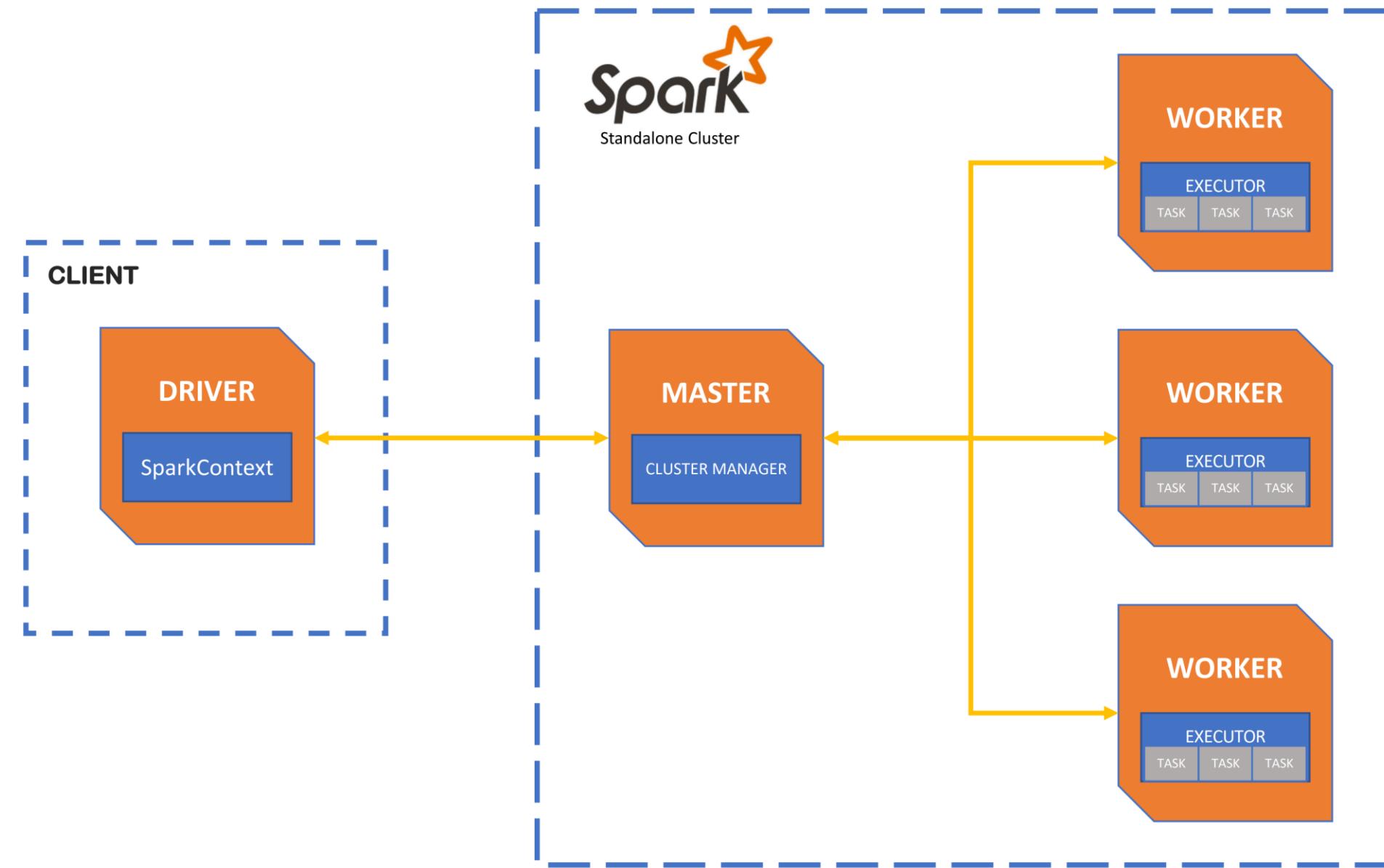


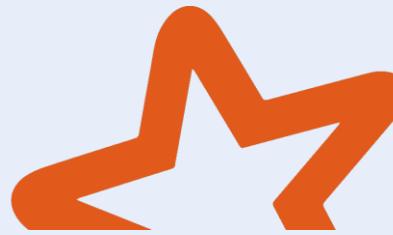


# Architectures de Spark

## Modes de Cluster

Local

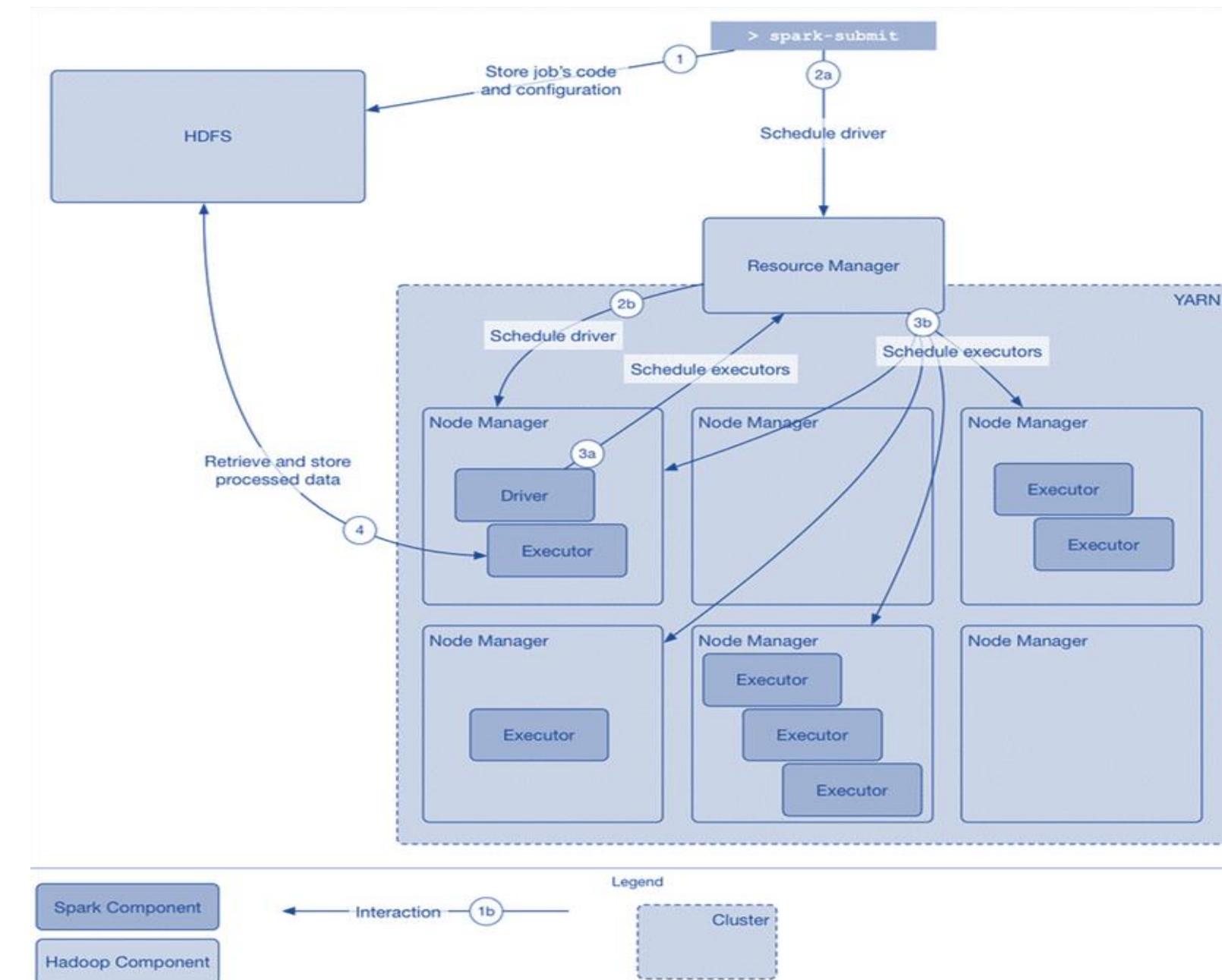




# Architectures de Spark

## Modes de Cluster

YARN

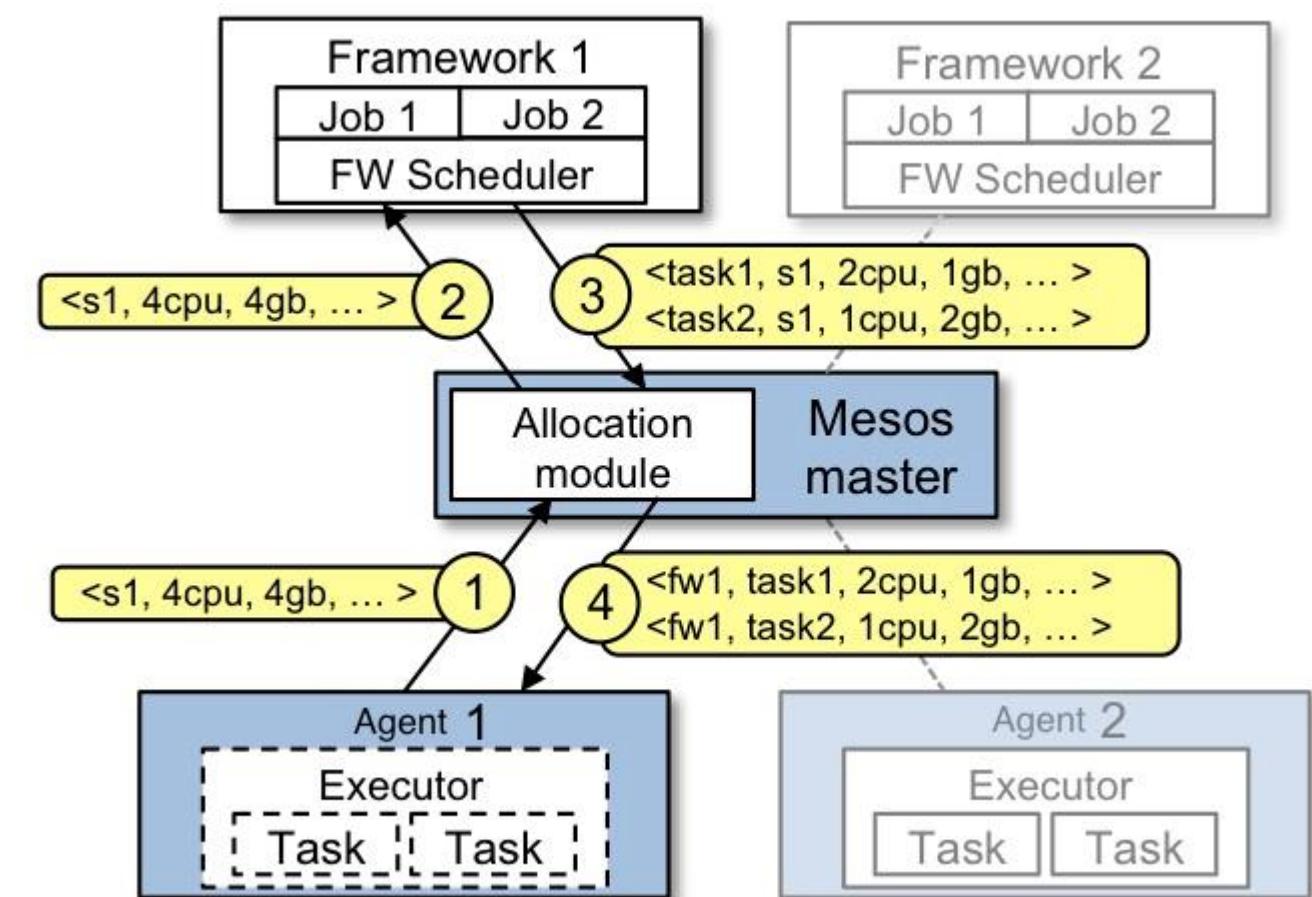
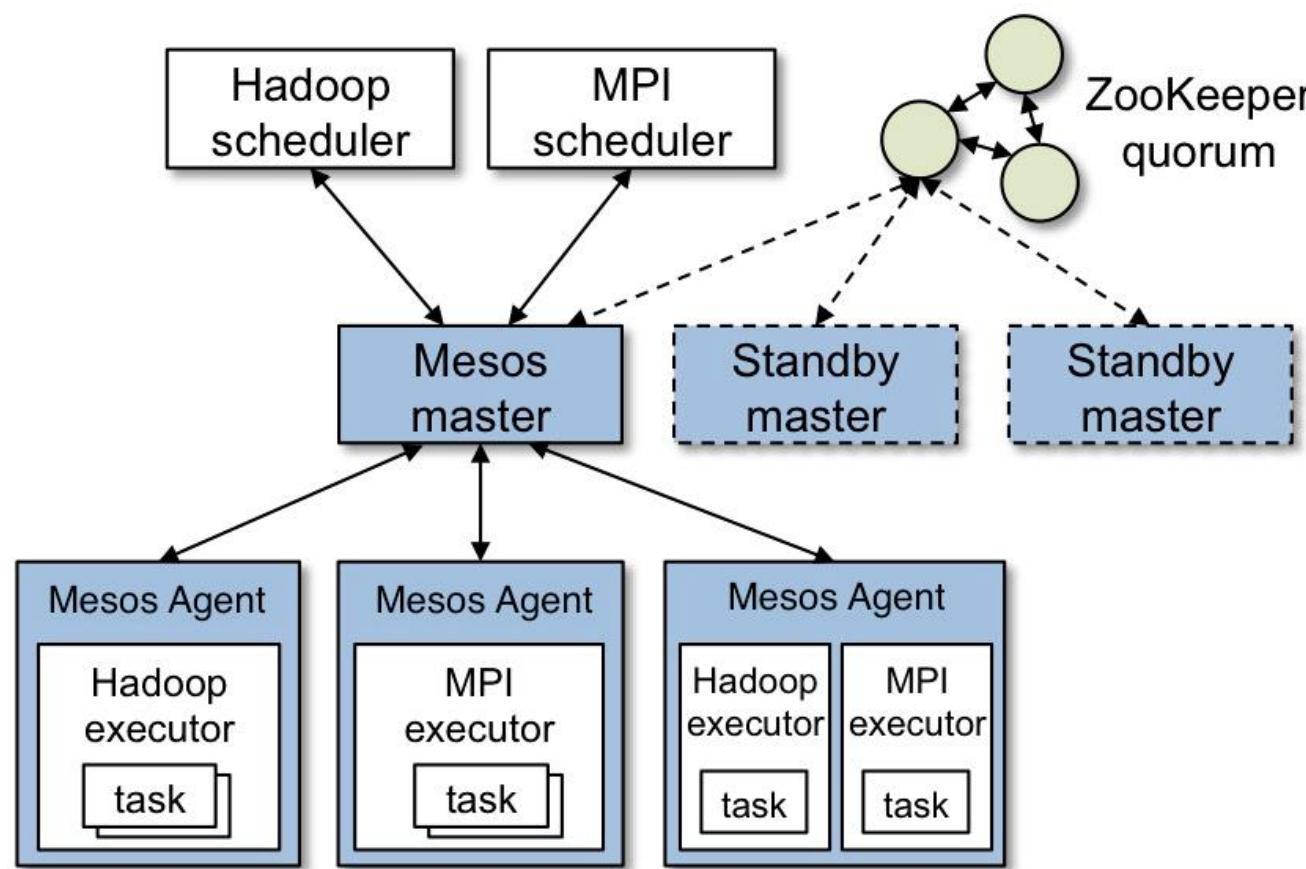




# Architectures de Spark

## Modes de Cluster

### Mesos

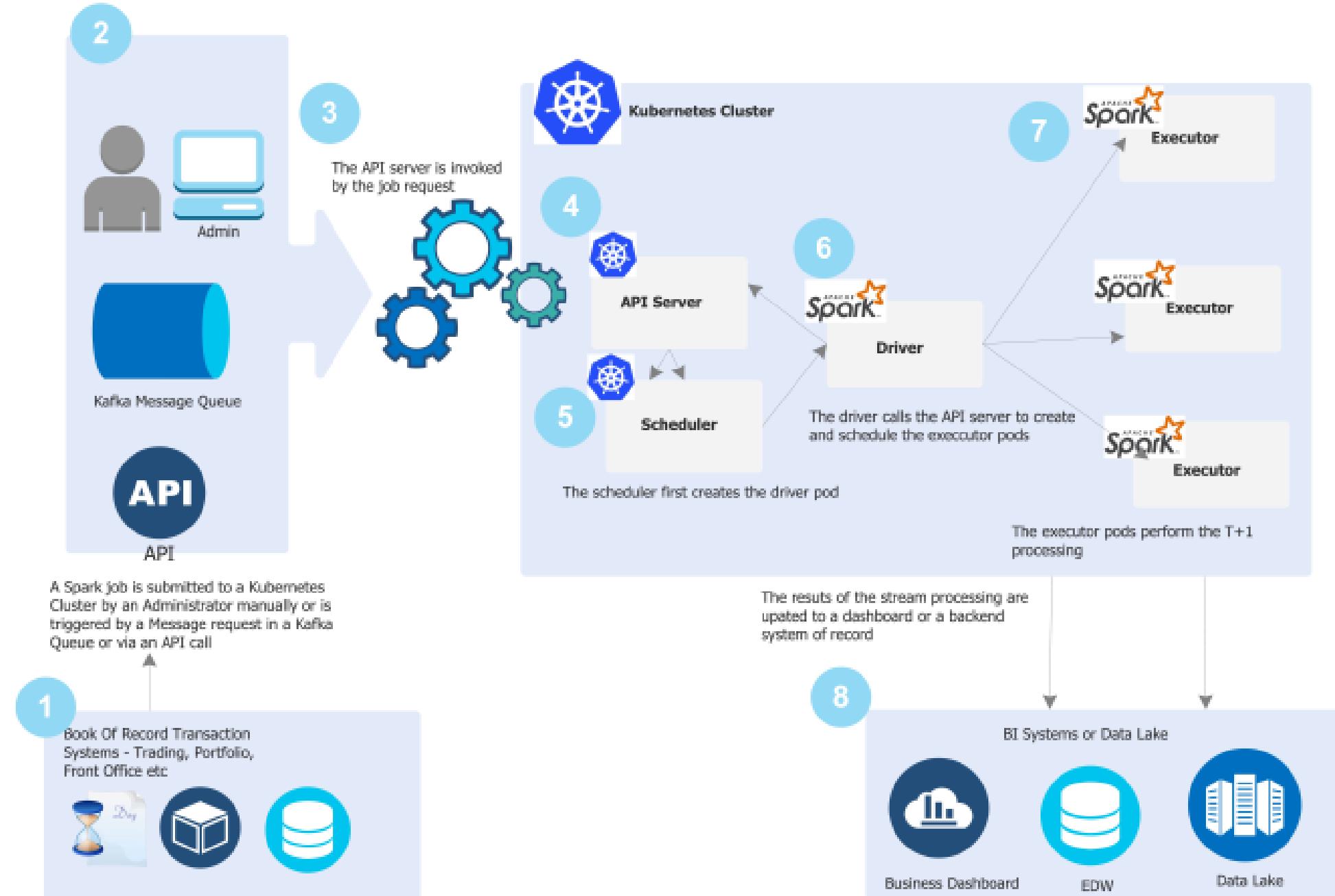


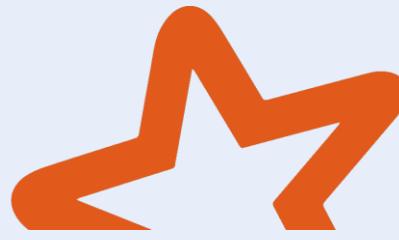


# Architectures de Spark

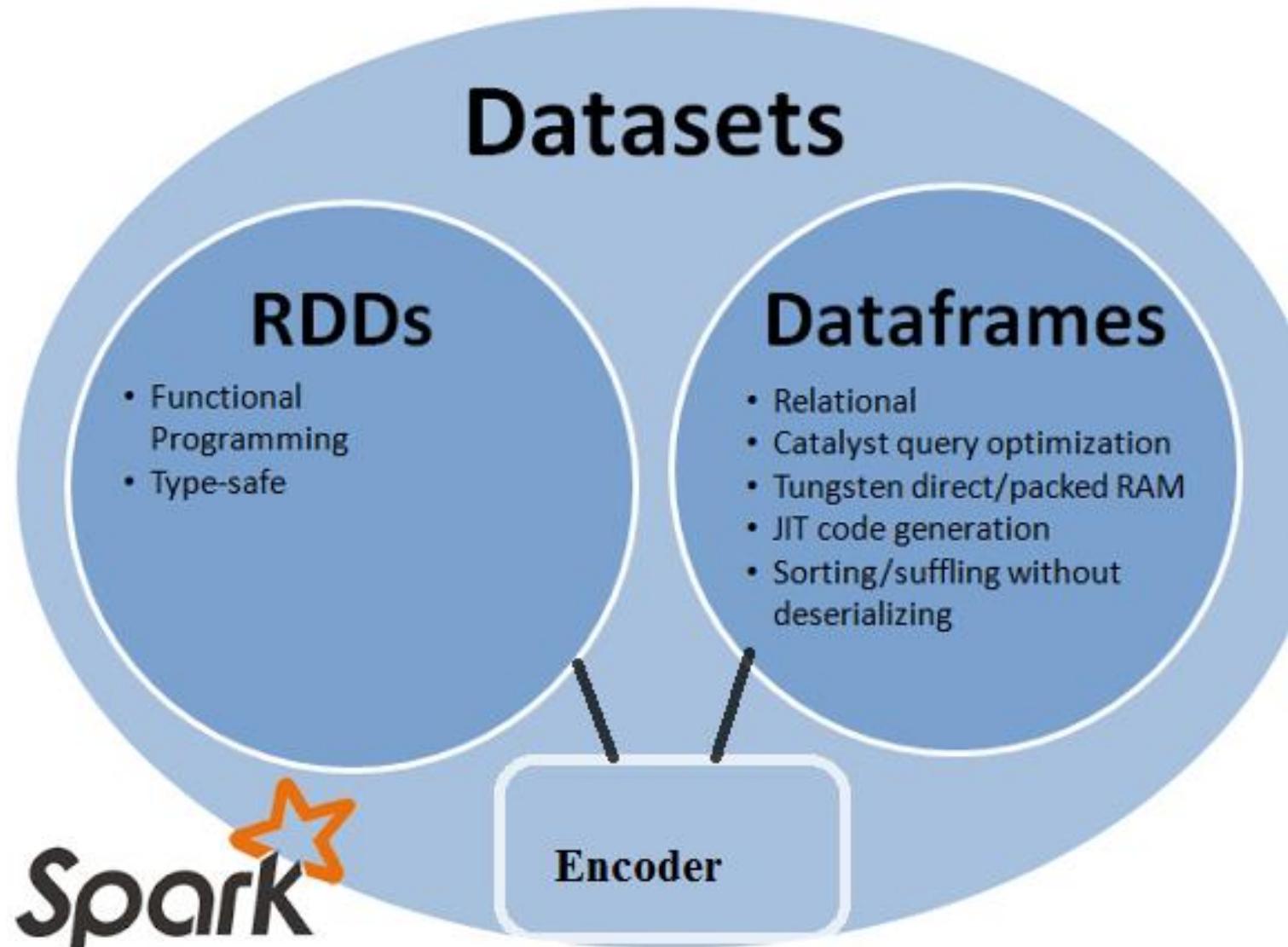
## Modes de Cluster

Kubernetes





# Modèle de Programmation de Spark



Transformations	Actions
<code>map (func)</code> <code>flatMap(func)</code> <code>filter(func)</code> <code>groupByKey()</code> <code>reduceByKey(func)</code> <code>mapValues(func)</code> <code>sample(...)</code> <code>union(other)</code> <code>distinct()</code> <code>sortByKey()</code> ...	<code>reduce(func)</code> <code>collect()</code> <code>count()</code> <code>first()</code> <code>take(n)</code> <code>saveAsTextFile(path)</code> <code>countByKey()</code> <code>foreach(func)</code> ...



## RDDs (Resilient Distributed Datasets)

### Spark 1.0

**Méthodes de transformation :**  
**map(), filter() et reduce()**

**Les méthodes d'action :**  
**collect() et saveAsObjectFile().**

#### Exemple RDD

```
rdd.filter(_.age > 21) // transformation  
    .map(_.last)// transformation  
    .saveAsObjectFile("under21.bin") // action
```



## DataFrames

### Spark 1.3.

**Concept de schéma pour décrire les données**

**Plan de requête relationnel**

#### Exemple en style SQL

```
df.filter("age > 21");
```

#### Limitations :

Comme le code fait référence aux attributs de données par leur nom, le compilateur ne peut pas détecter d'erreurs. Si les noms des attributs sont incorrects, l'erreur ne sera détectée qu'au moment de l'exécution, lors de la création du plan de requête.



## Dataset

Exemple en style SQL

```
dataset.filter(_.age < 21);
```

## Spark 1.6

### Encodeurs

### Beans

Evaluations diff. entre DataFrame & DataSet :



SQL

DataFrames

Datasets

Syntax  
Errors

Runtime

Compile  
Time

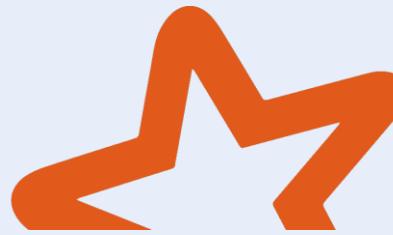
Compile  
Time

Analysis  
Errors

Runtime

Runtime

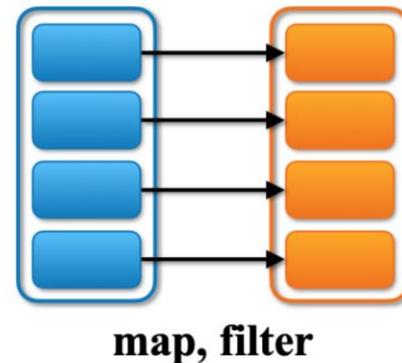
Compile  
Time



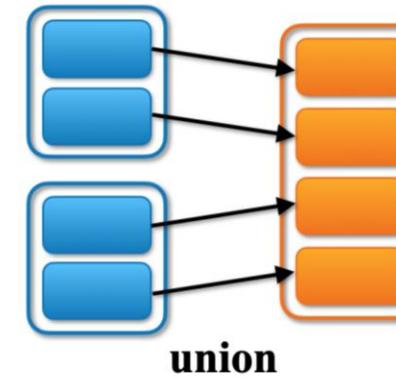
# Modèle de Programmation de Spark

## Transformations

Narrow transformations

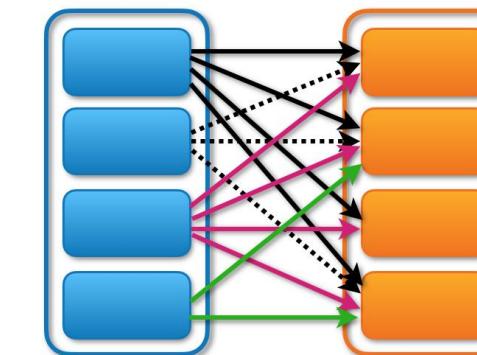


**map, filter**

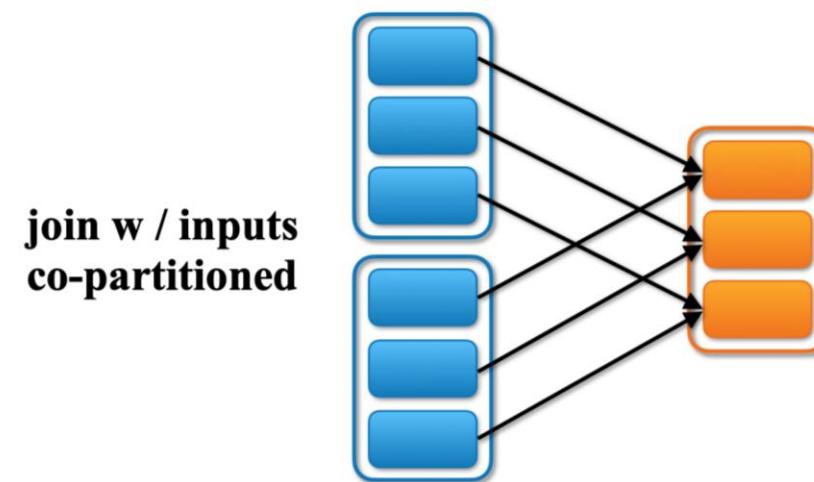


**union**

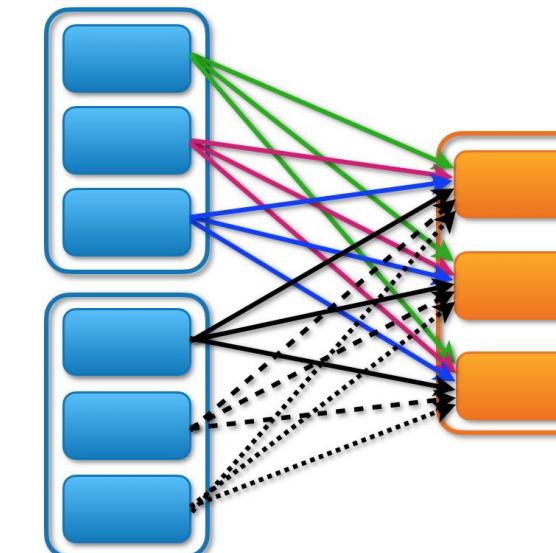
Wide transformations



**groupByKey**



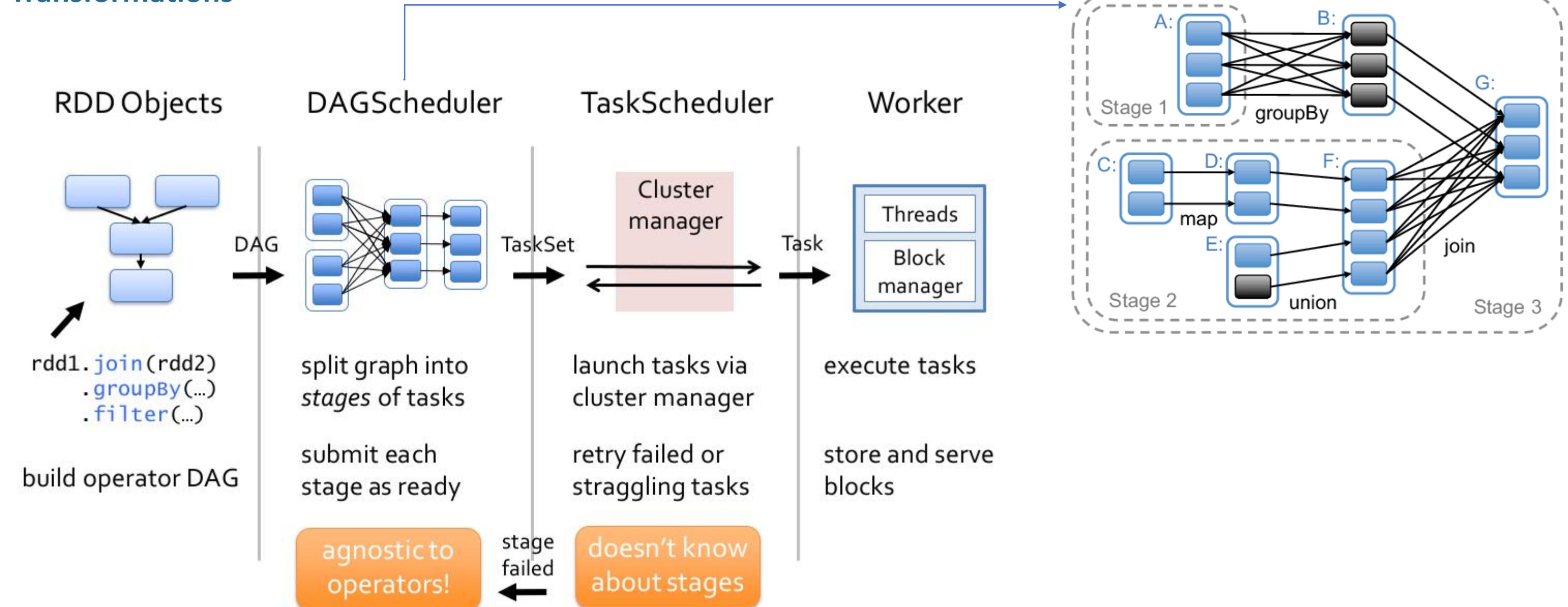
**join w / inputs  
co-partitioned**



**join w / inputs not  
co-partitioned**

# Modèle de Programmation de Spark

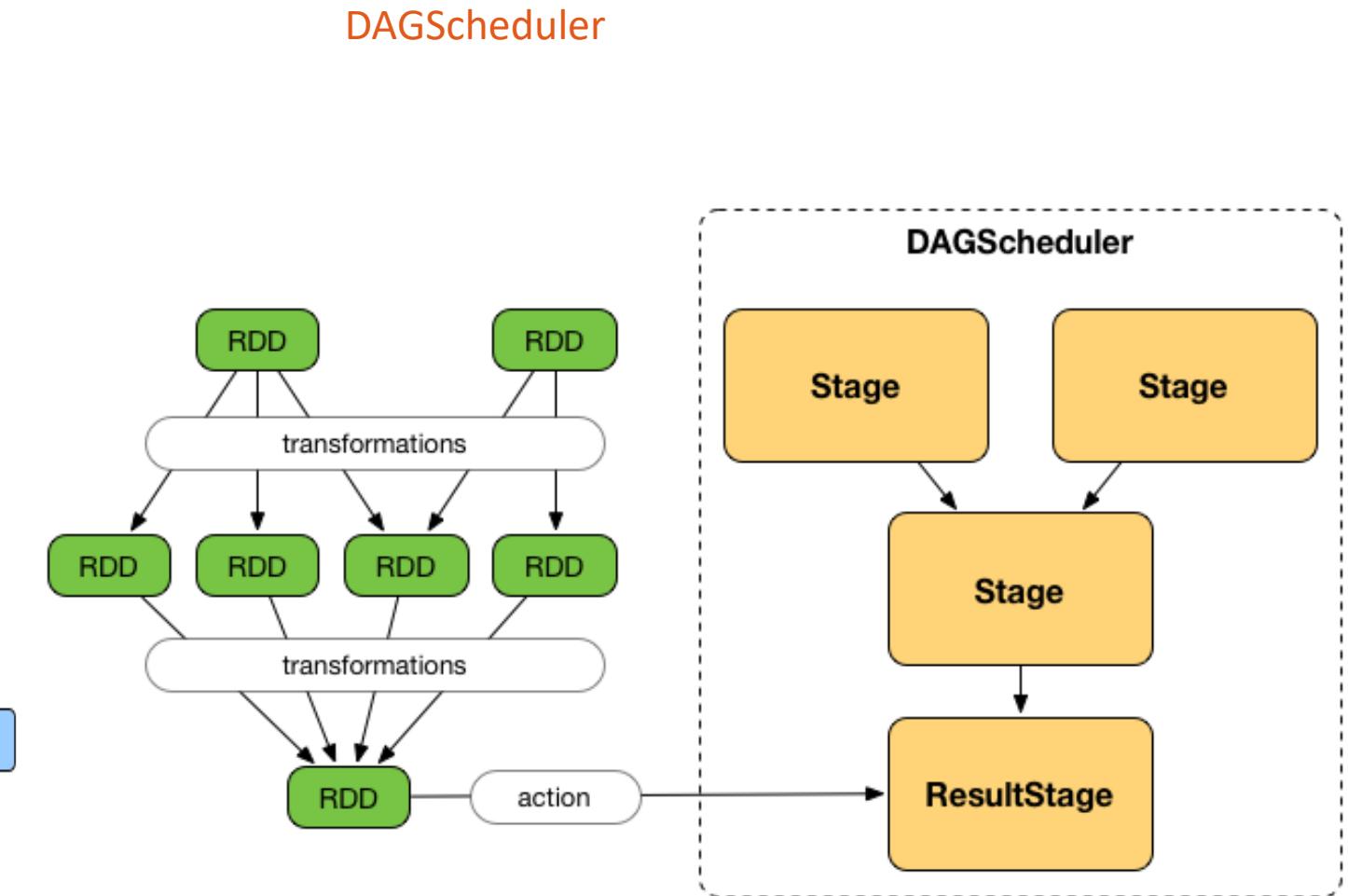
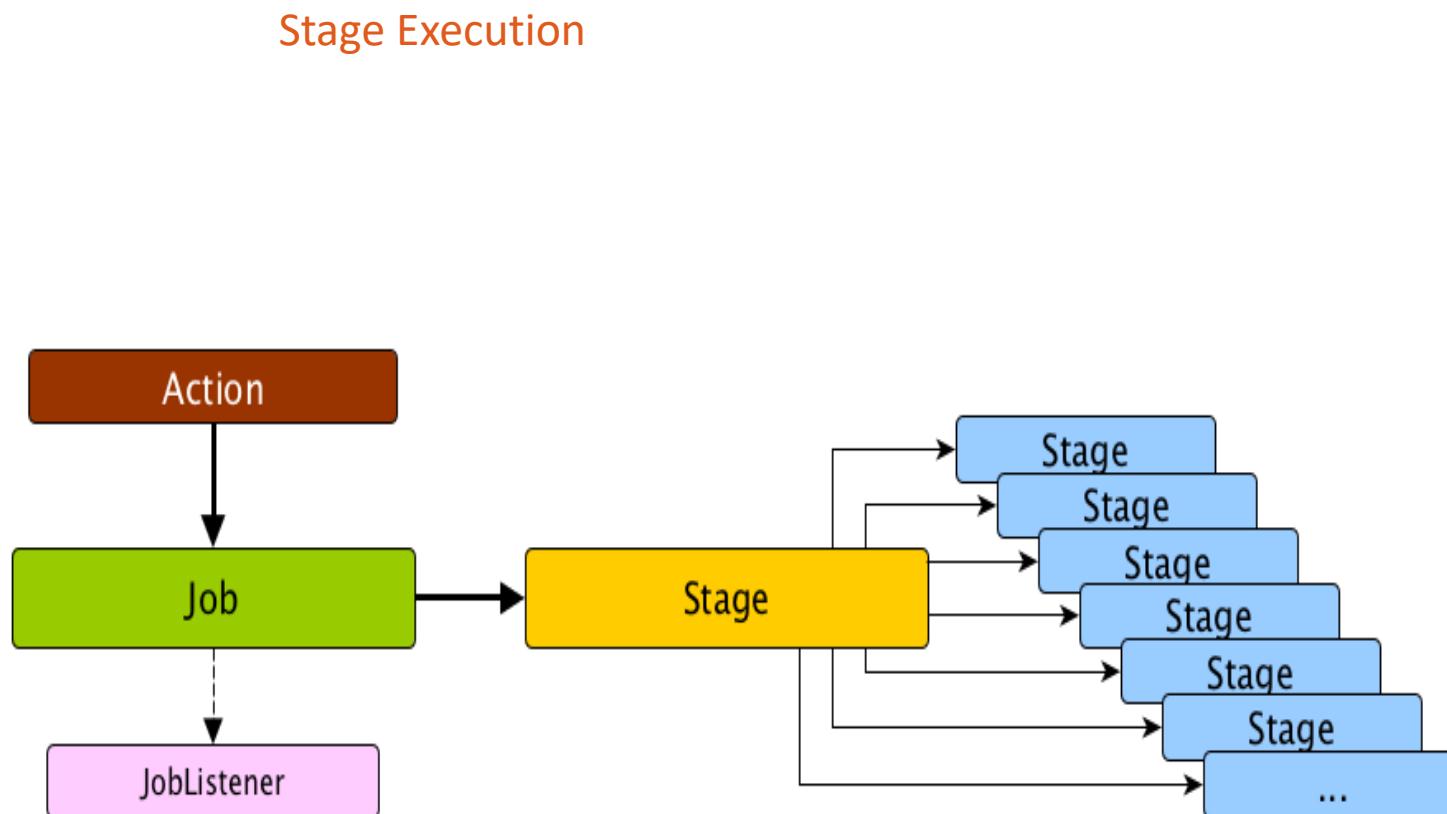
## Transformations





# Modèle de Programmation de Spark

## Actions

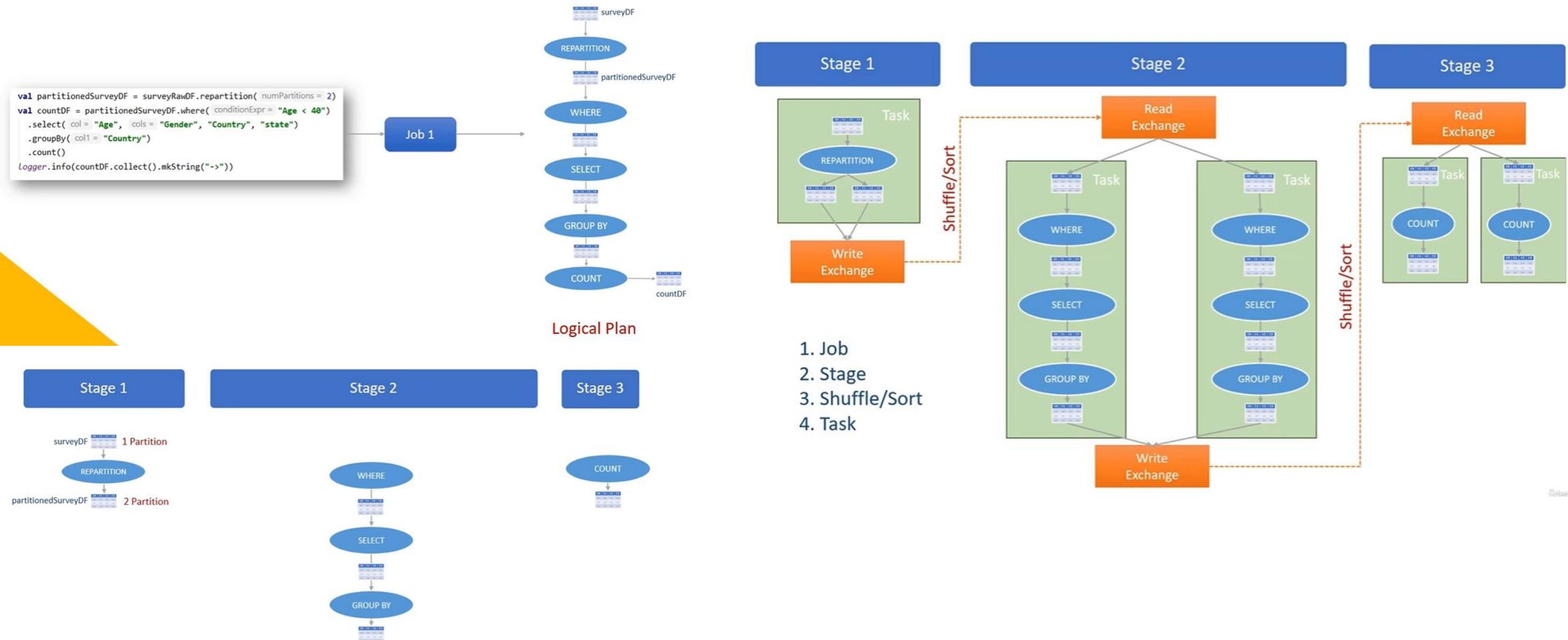




# Modèle de Programmation de Spark

## Actions

### Plan logique 1 & 2

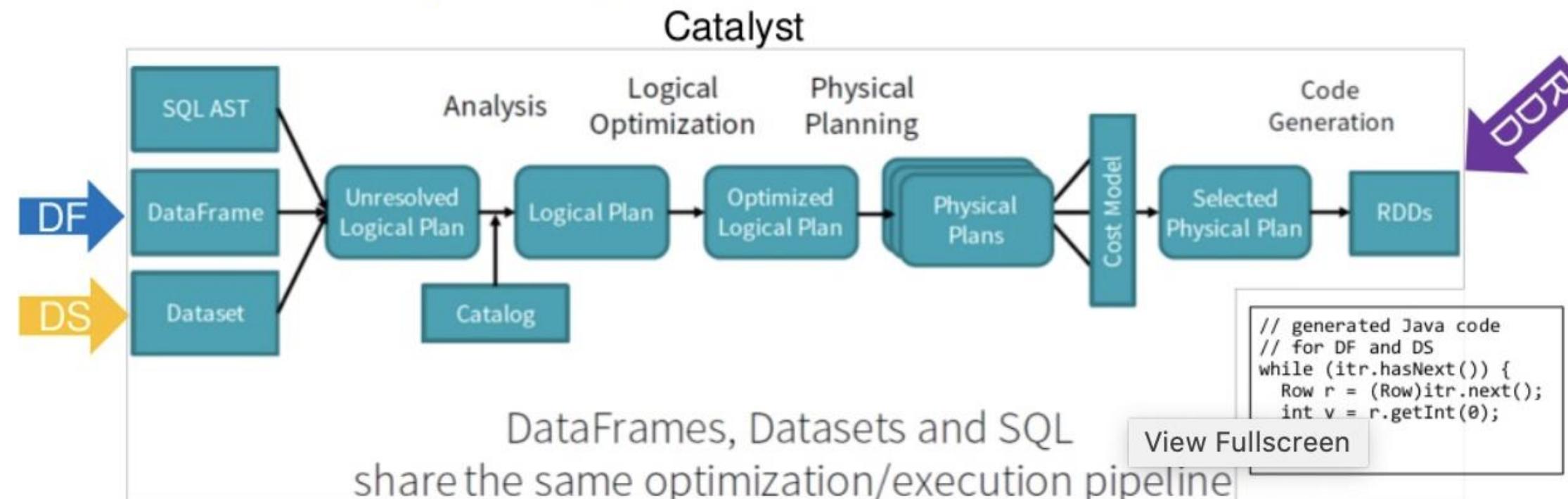




# Modèle de Programmation de Spark

## Synthèse

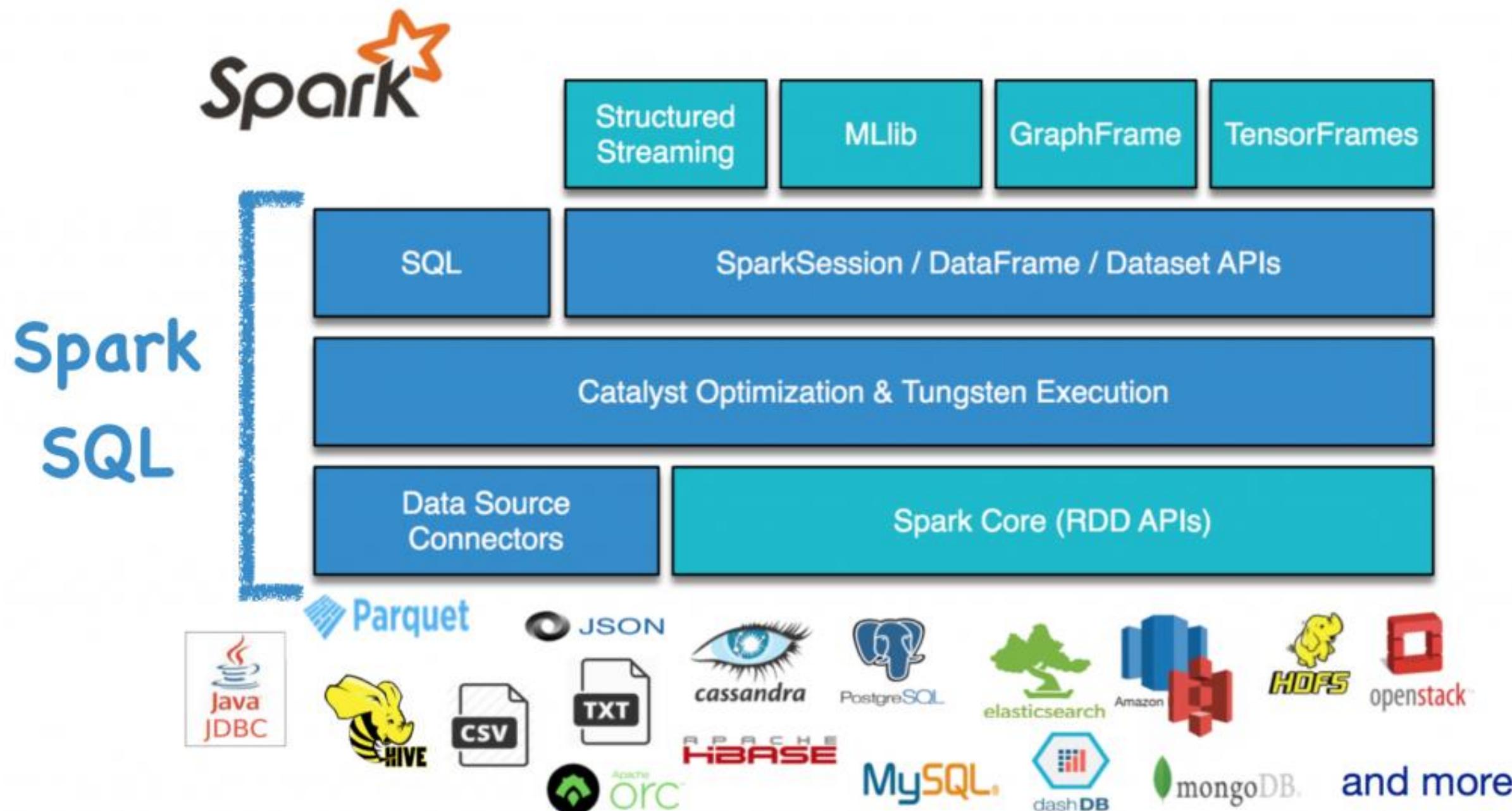
## How DF, DS, and RDD Work

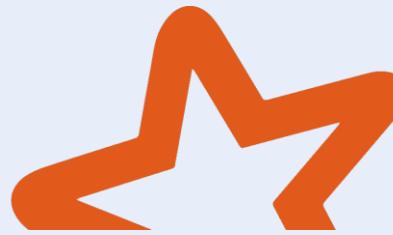


```
// select specific fields from the Dataset, apply a predicate using the where() method, convert to an RDD, and show first 10 RDD rows  
val deviceEventsDS = ds.select($"device_name", $"cca3", $"c02_level").where($"c02_level" > 1300)
```

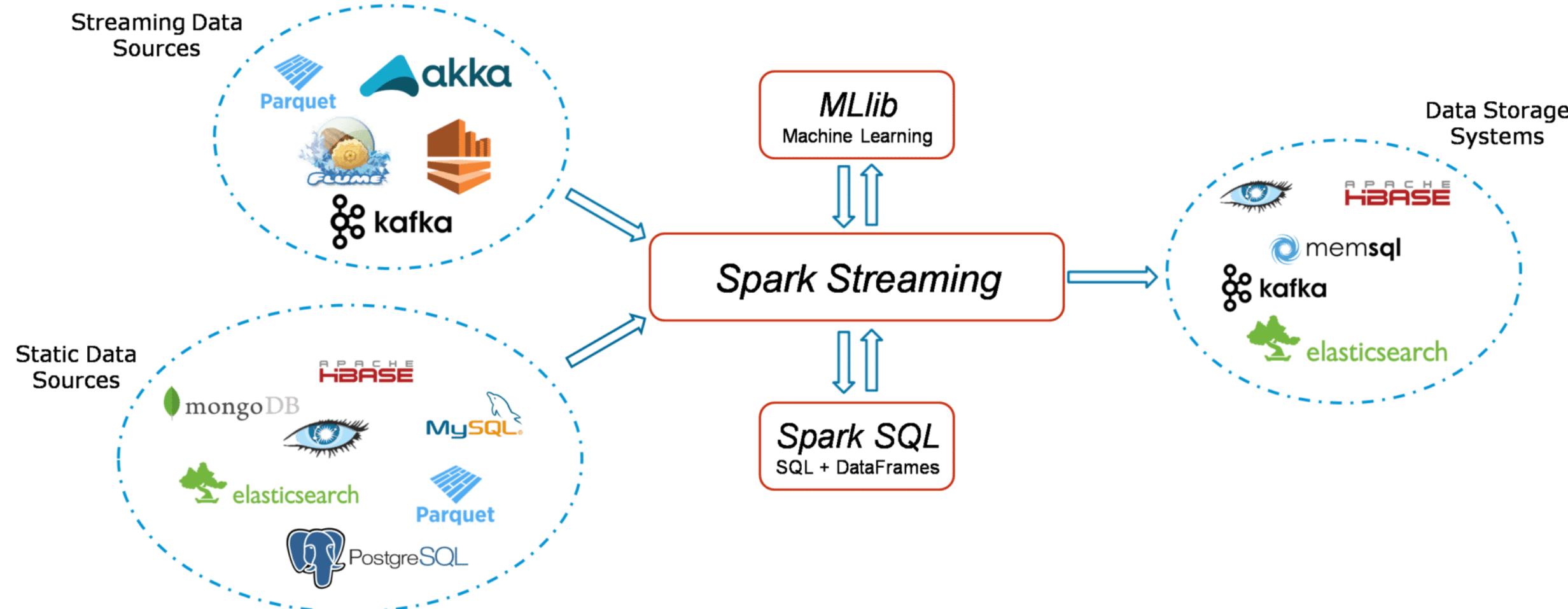
```
// convert to RDDs and take the first 10 rows  
val eventsRDD = deviceEventsDS.rdd.take(10)
```

# Spark SQL: Traitement de Données Structurées





# Streaming avec Apache Spark



Traitement en Temps  
Réel



Micro-Batchs



DStreams (Discretized  
Stream)



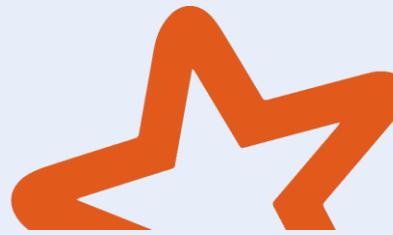
Streaming Structuré  
Structuré



# Exemple Spark SQL – Spark Streaming

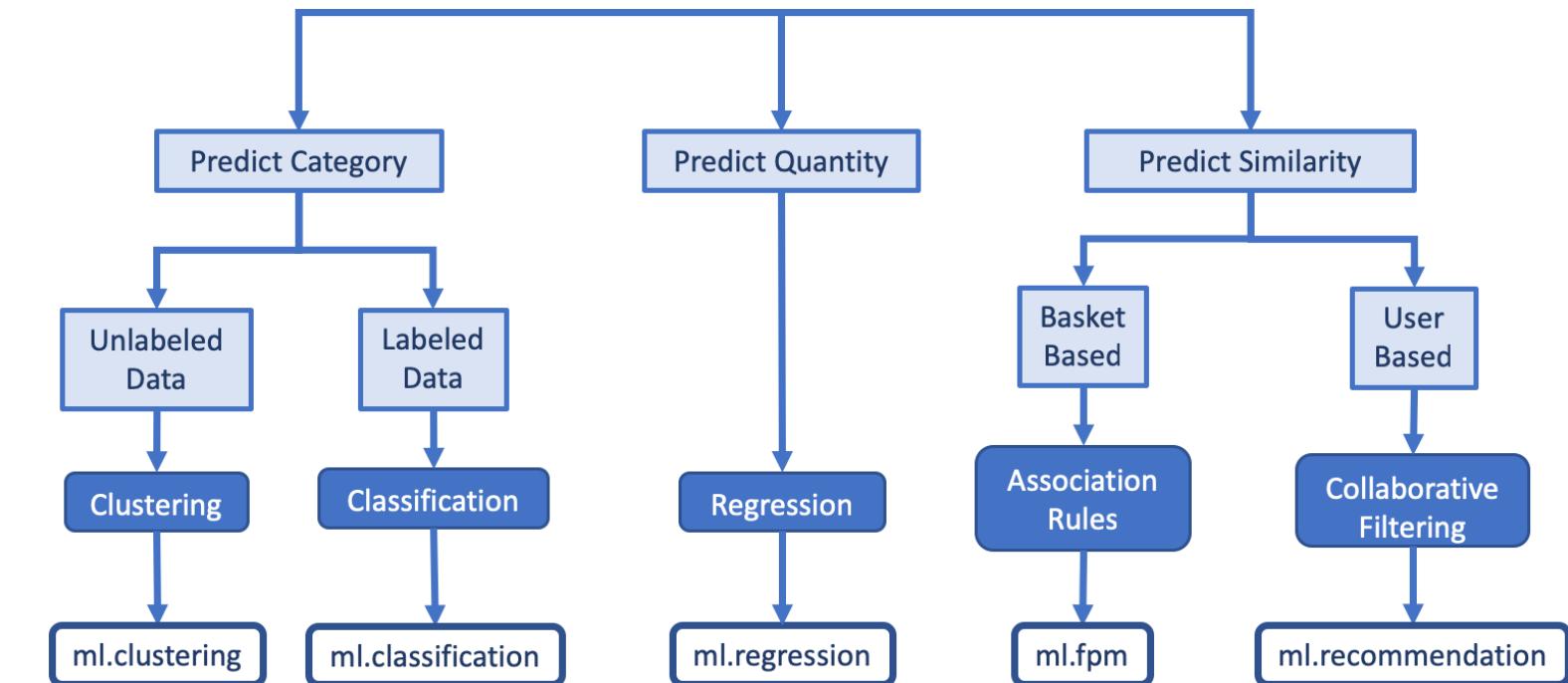
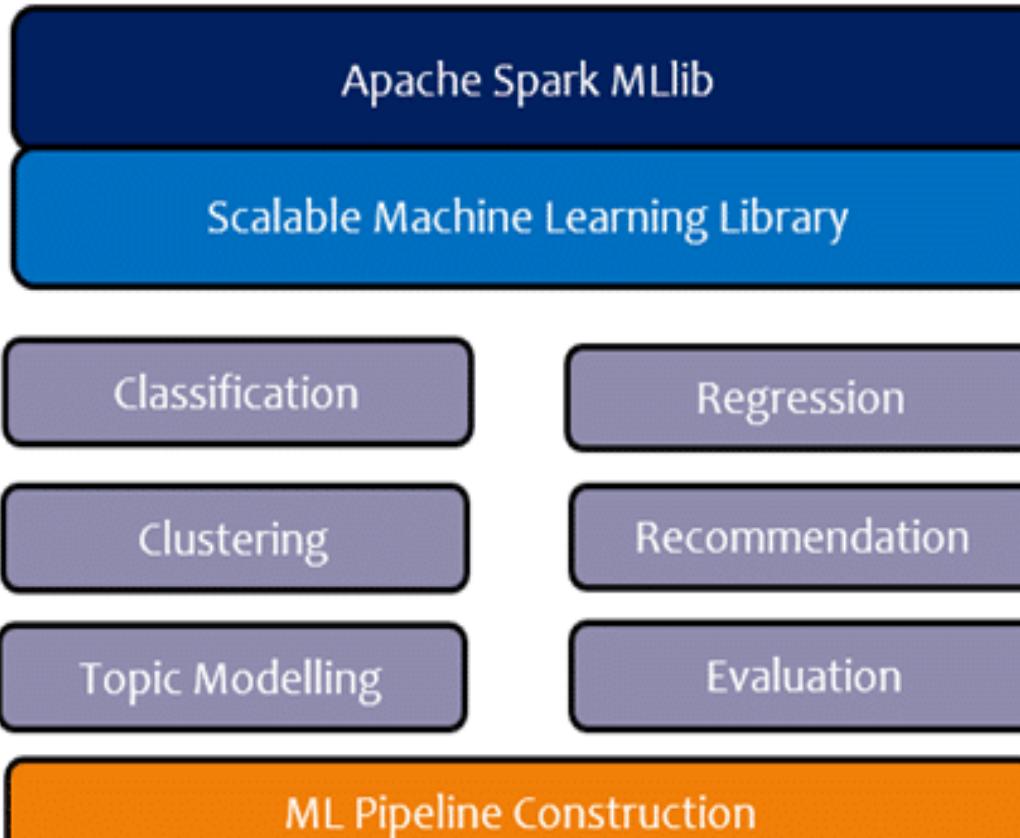
```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import *
3
4 # Initialiser une SparkSession
5 spark = SparkSession.builder \
6     .appName('Spark SQL and Streaming Example') \
7     .enableHiveSupport() \
8     .getOrCreate()
9
10 # Lire les données de la base de données PostgreSQL
11 jdbcDF = spark.read \
12     .format("jdbc") \
13     .option("url", "jdbc:postgresql://localhost:5432/database") \
14     .option("dbtable", "table") \
15     .option("user", "username") \
16     .option("password", "password") \
17     .load()
18
19 # Enregistrer le DataFrame en tant que vue SQL temporaire
20 jdbcDF.createOrReplaceTempView("table")
21 |
```

```
1 Effectuer une requête SQL
2 sqlDF = spark.sql("SELECT * FROM table WHERE some_condition")
3
4 # Créer un DataFrame à partir du flux de données
5 streamingDF = spark.readStream \
6     .format("socket") \
7     .option("host", "localhost") \
8     .option("port", 9999) \
9     .load()
10
11 # Appliquer une transformation sur le flux de données
12 transformedDF = streamingDF.selectExpr("CAST(value AS STRING)")
13
14 # Écrire le DataFrame transformé dans la base de données Hive
15 transformedDF.write \
16     .format("hive") \
17     .saveAsTable("hive_table")
18 |
```



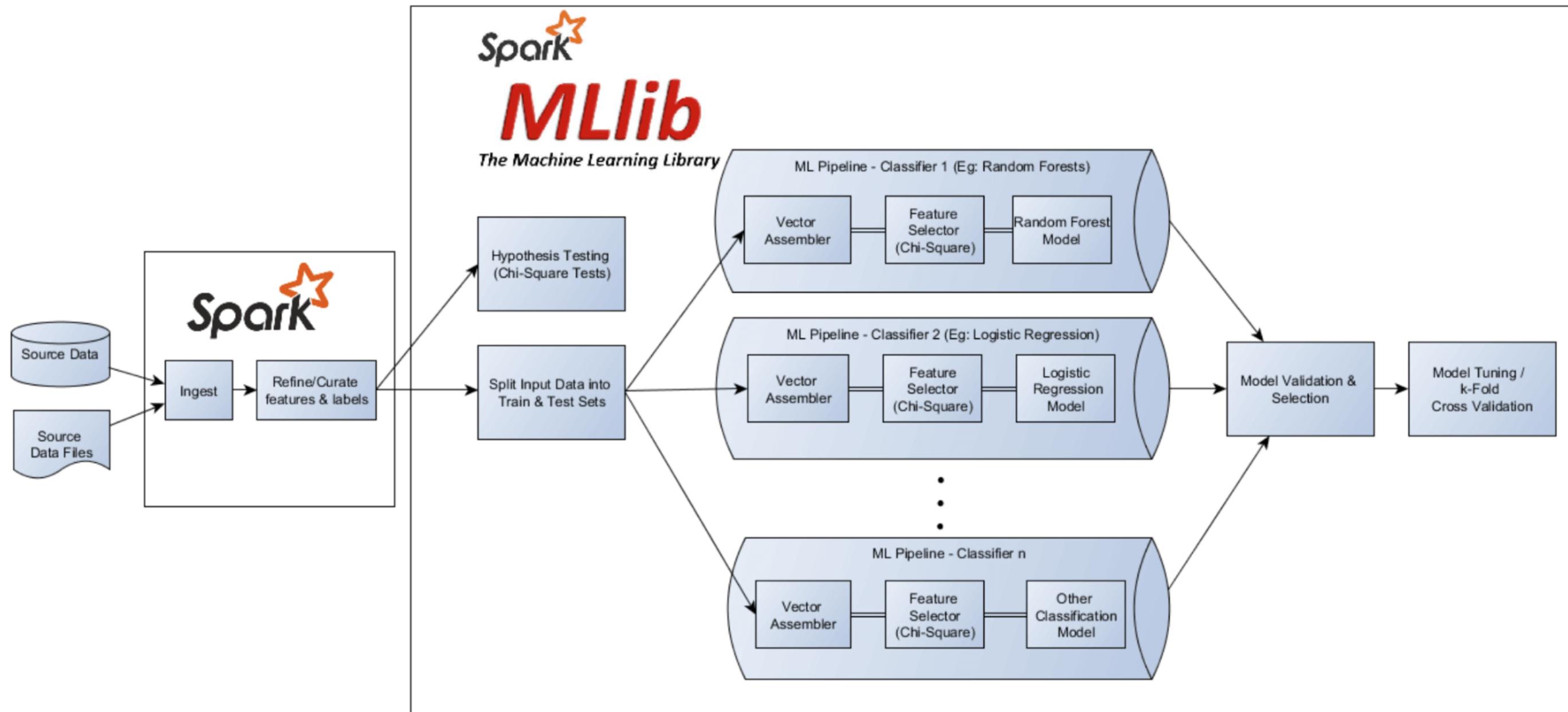
# Machine Learning avec Spark (MLlib)

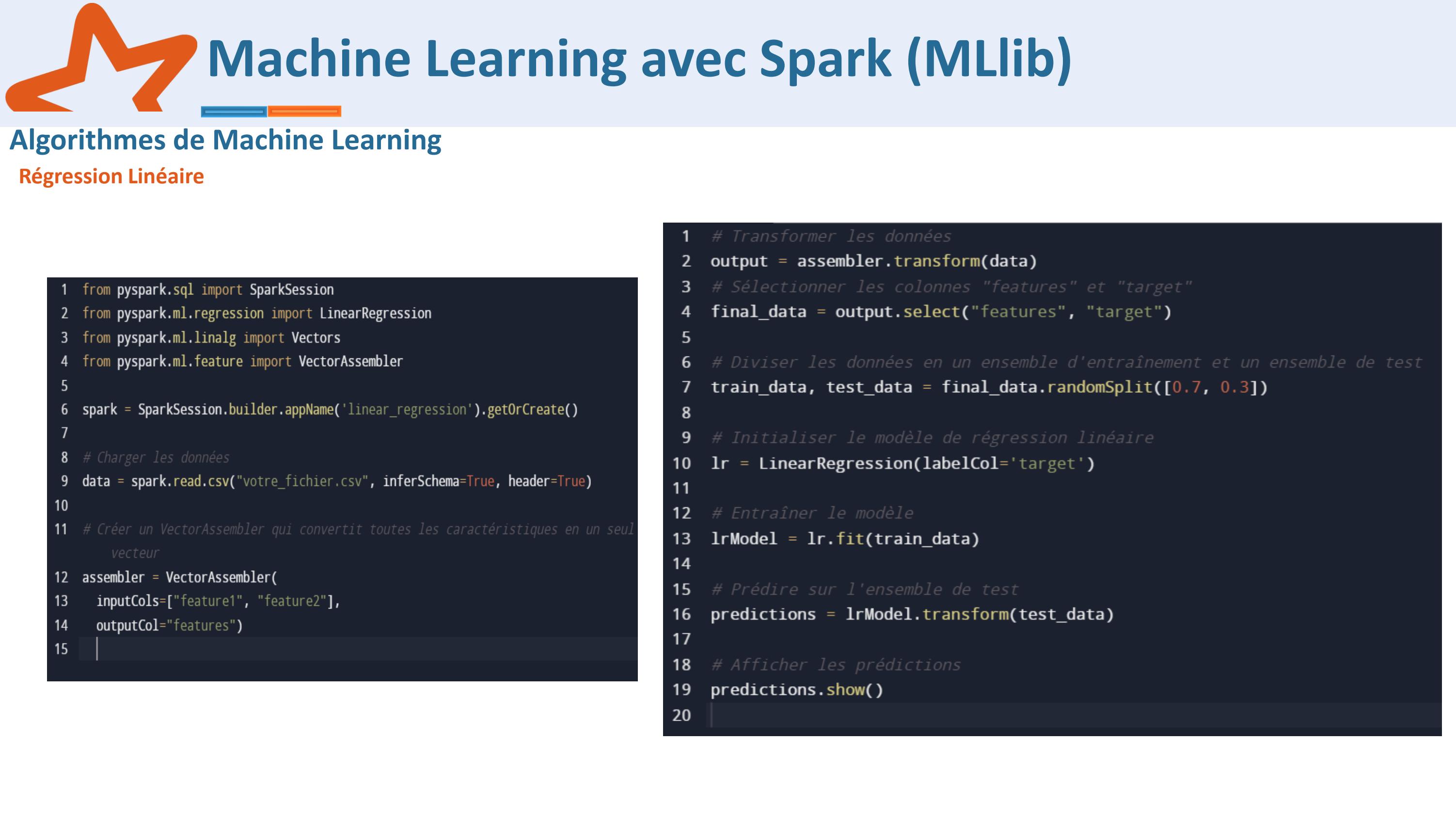
## Algorithmes de Machine Learning



# Machine Learning avec Spark (MLlib)

## API de Pipeline



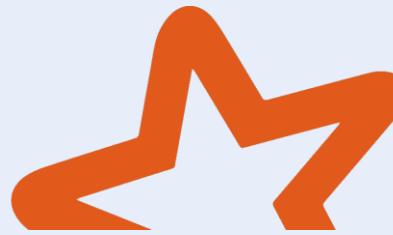


## Algorithmes de Machine Learning

### Régression Linéaire

```
1 from pyspark.sql import SparkSession
2 from pyspark.ml.regression import LinearRegression
3 from pyspark.ml.linalg import Vectors
4 from pyspark.ml.feature import VectorAssembler
5
6 spark = SparkSession.builder.appName('linear_regression').getOrCreate()
7
8 # Charger les données
9 data = spark.read.csv("votre_fichier.csv", inferSchema=True, header=True)
10
11 # Créer un VectorAssembler qui convertit toutes les caractéristiques en un seul
12 # vecteur
12 assembler = VectorAssembler(
13     inputCols=["feature1", "feature2"],
14     outputCol="features")
15 |
```

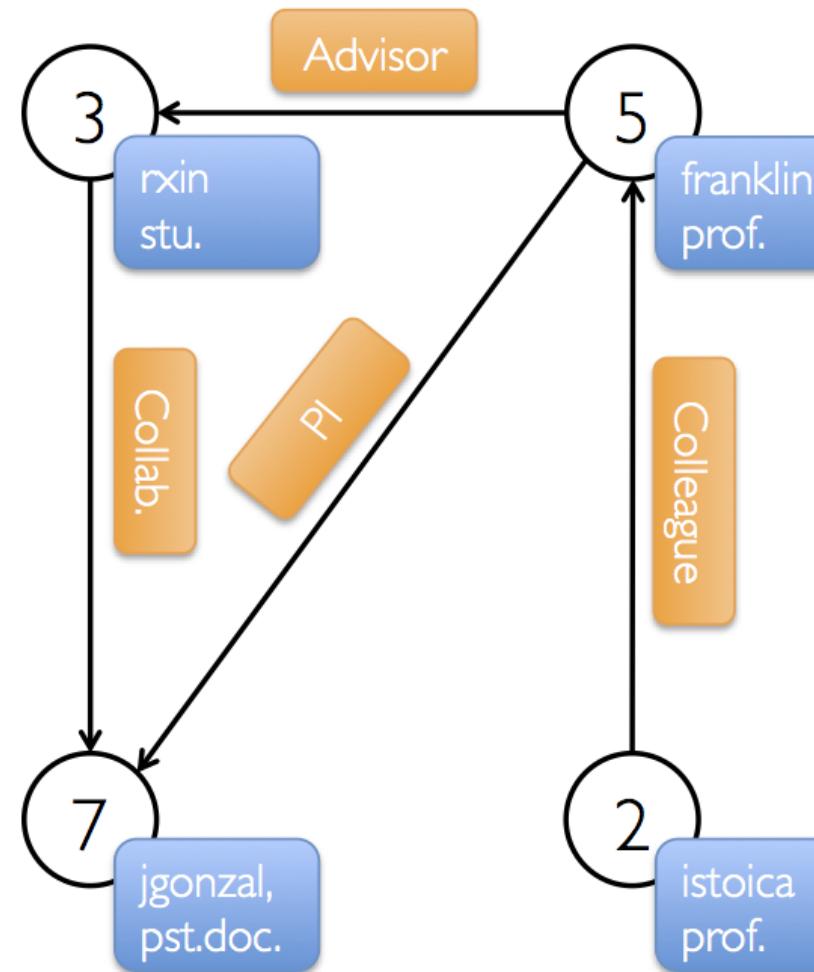
```
1 # Transformer les données
2 output = assembler.transform(data)
3 # Sélectionner les colonnes "features" et "target"
4 final_data = output.select("features", "target")
5
6 # Diviser les données en un ensemble d'entraînement et un ensemble de test
7 train_data, test_data = final_data.randomSplit([0.7, 0.3])
8
9 # Initialiser le modèle de régression linéaire
10 lr = LinearRegression(labelCol='target')
11
12 # Entraîner le modèle
13 lrModel = lr.fit(train_data)
14
15 # Prédire sur l'ensemble de test
16 predictions = lrModel.transform(test_data)
17
18 # Afficher les prédictions
19 predictions.show()
20 |
```



# Traitement de Graphes avec Spark (GraphX)

## Représentation des Graphes

### Property Graph



### Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

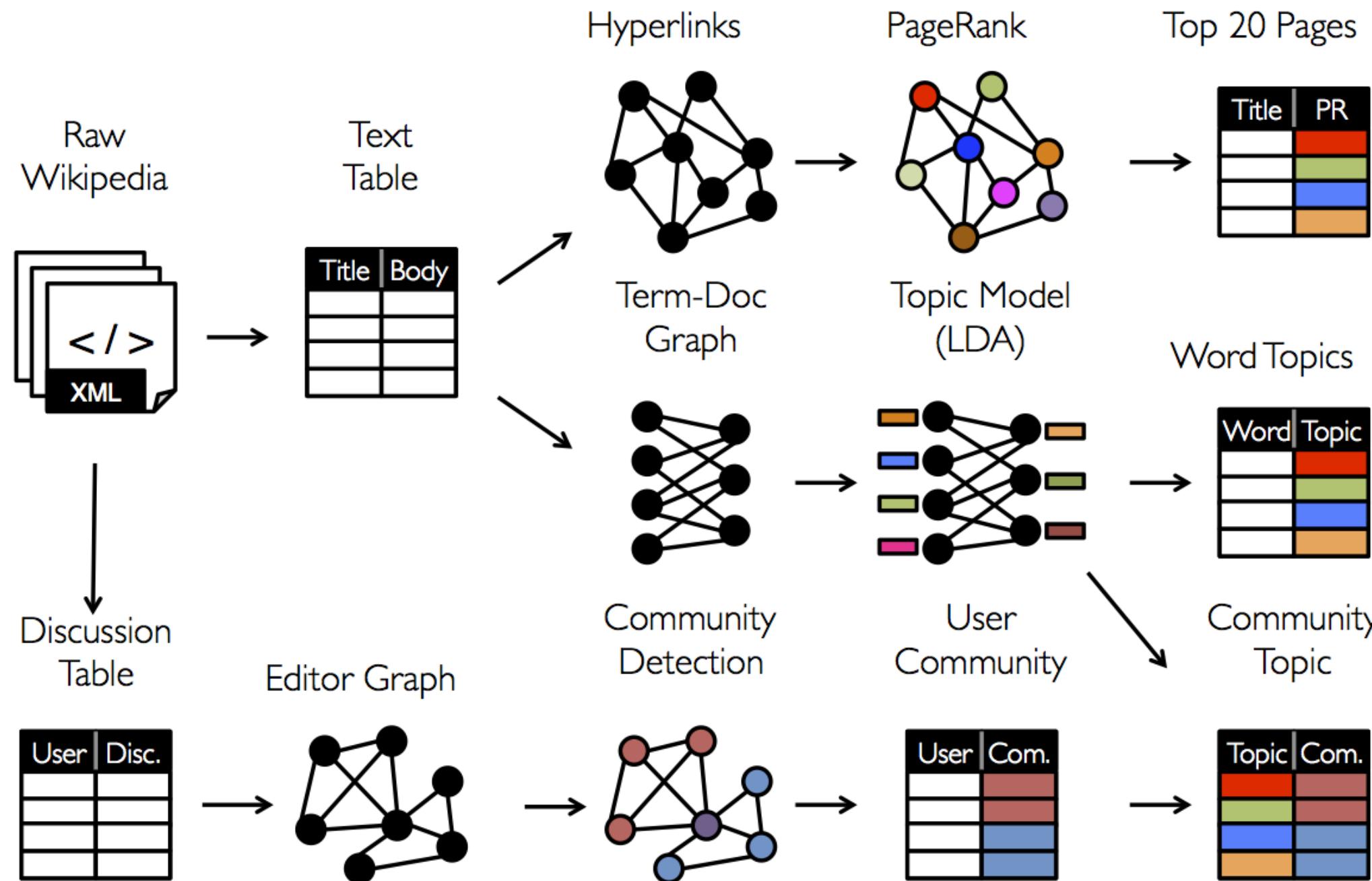
### Edge Table

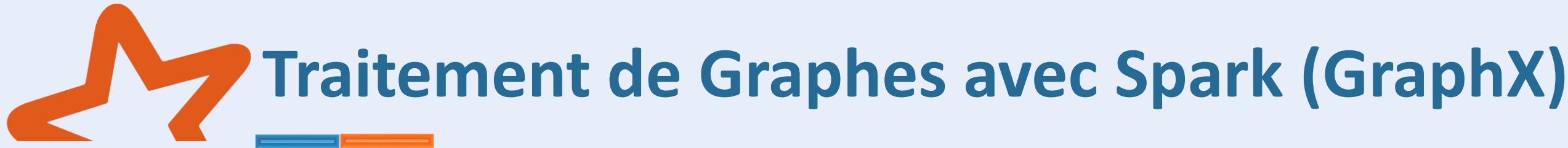
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI



# Traitements de Graphes avec Spark (GraphX)

## Algorithmes de Graphes





## Algorithmes de Graphes

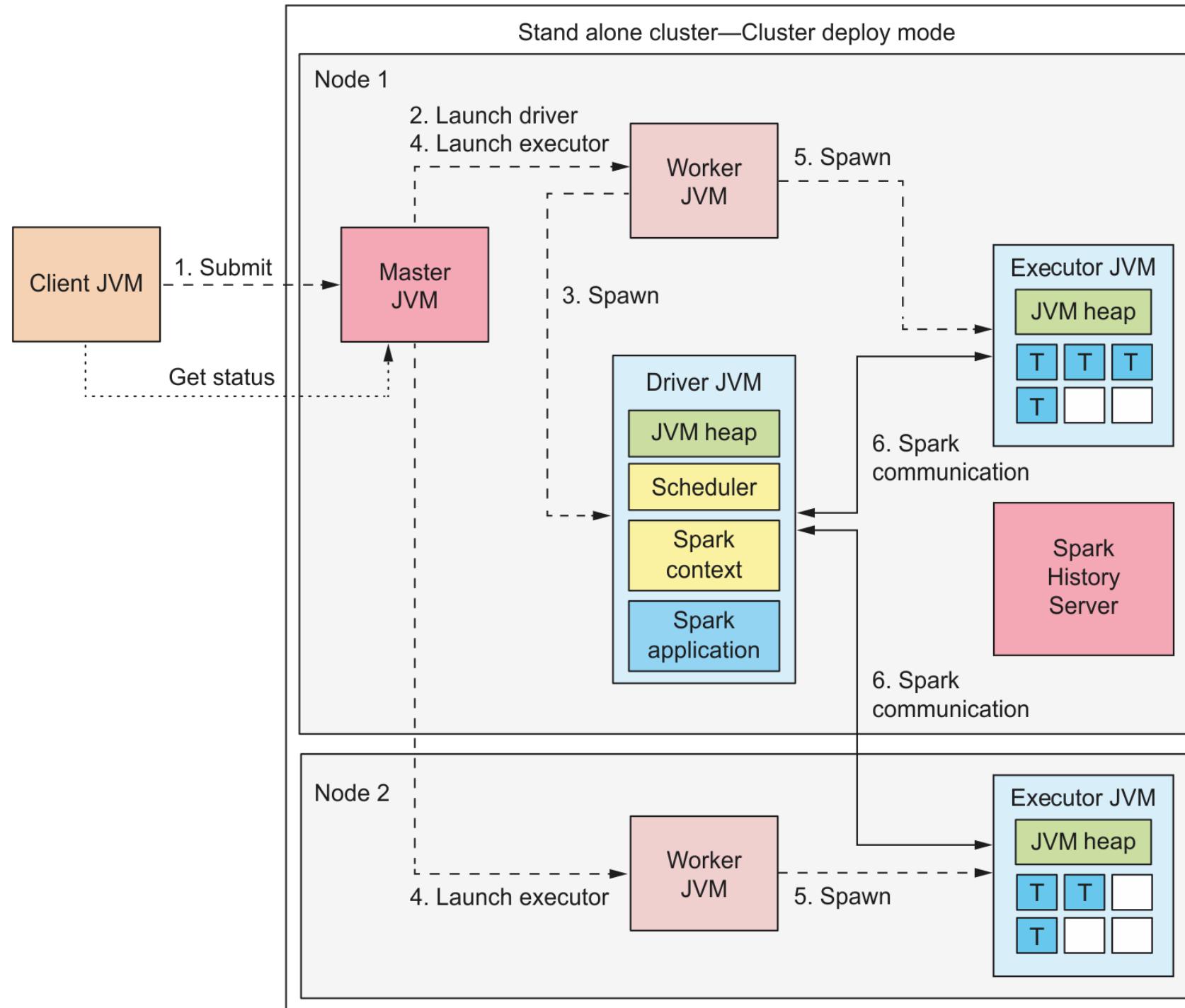
### PageRank

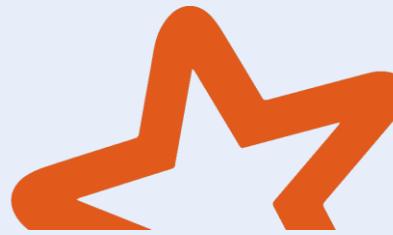
```
1 import org.apache.spark.graphx.GraphLoader
2
3 // Load the edges as a graph
4 val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
5 // Run PageRank
6 val ranks = graph.pageRank(0.0001).vertices
7 // Join the ranks with the usernames
8 val users = sc.textFile("data/graphx/users.txt").map { line =>
9   val fields = line.split(",")
10  (fields(0).toLong, fields(1))
11 }
12 val ranksByUsername = users.join(ranks).map {
13   case (id, (username, rank)) => (username, rank)
14 }
15 // Print the result
16 println(ranksByUsername.collect().mkString("\n"))
17 |
```



# Options de Déploiement de Spark

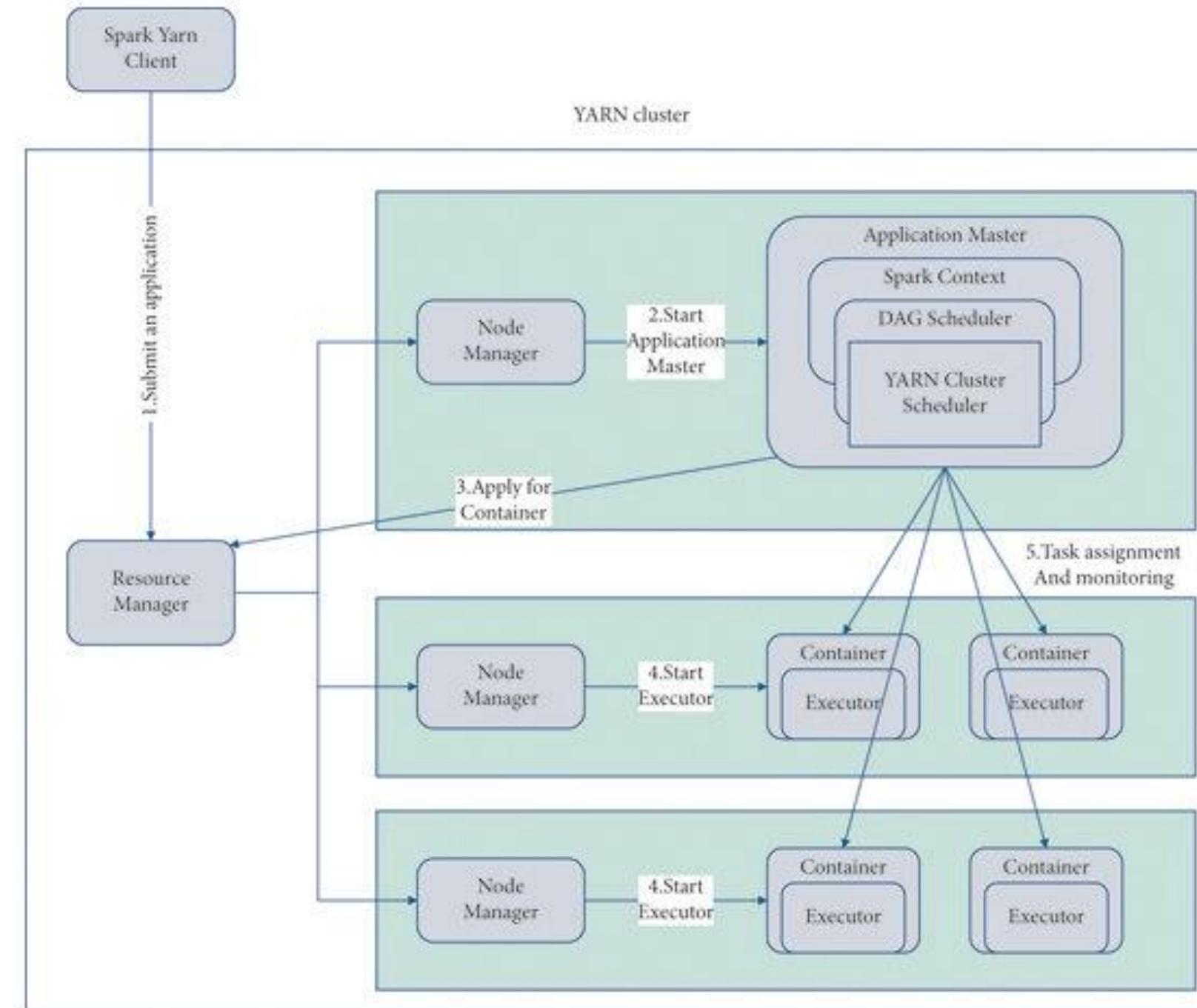
## Mode Local

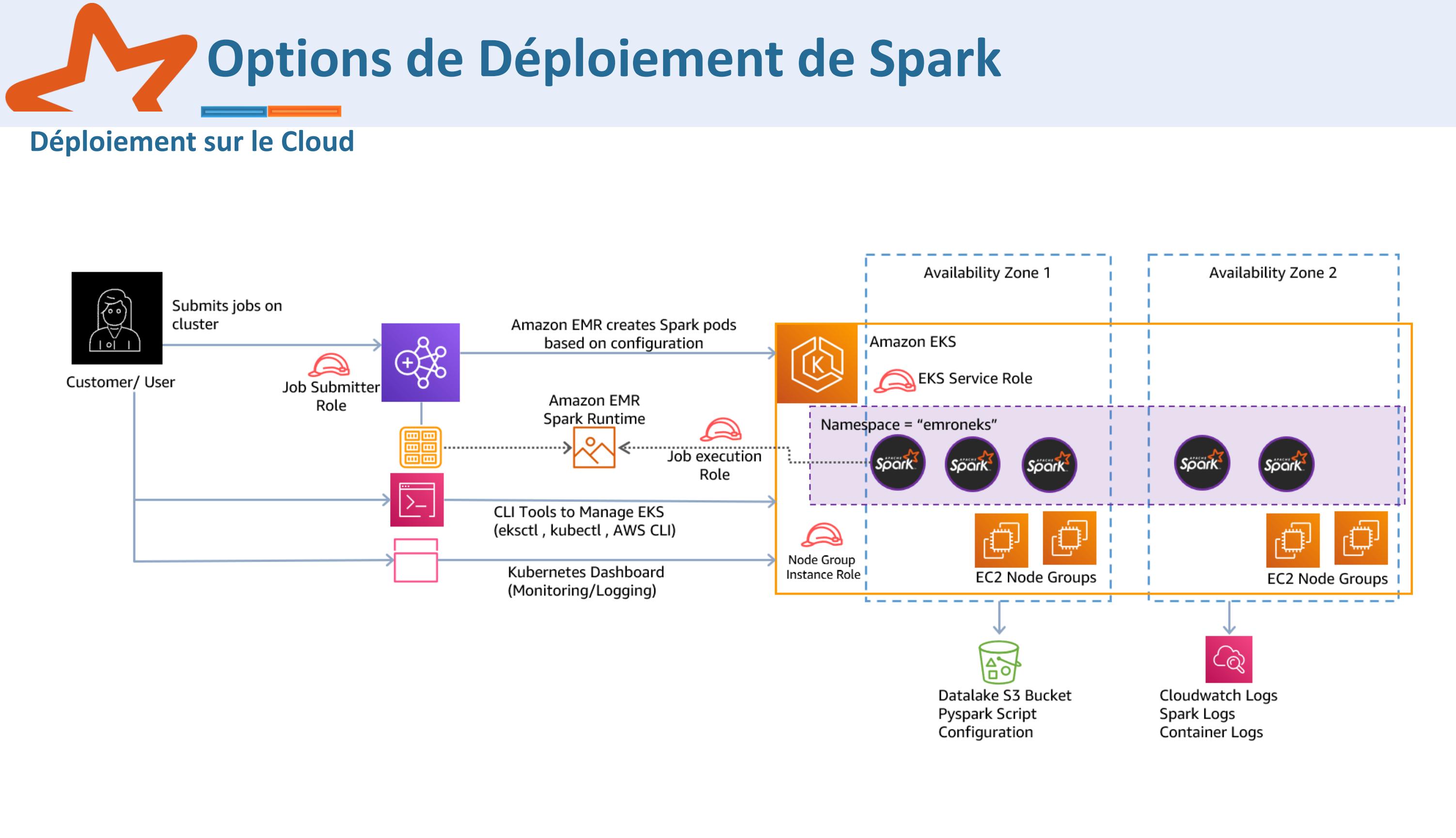


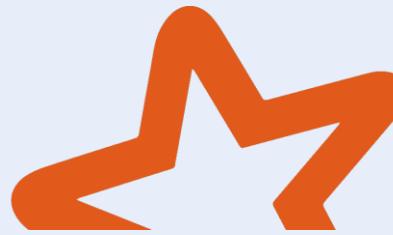


# Options de Déploiement de Spark

## Mode YARN Cluster

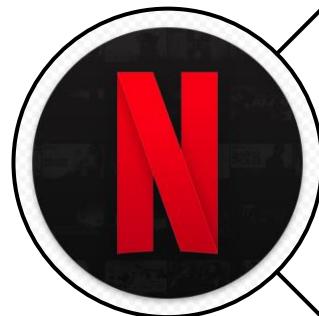




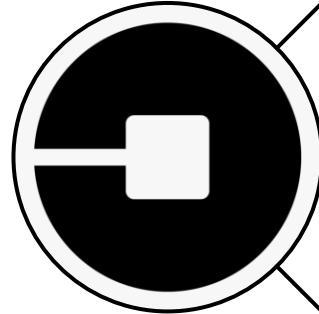


# Cas d'Utilisation et Applications

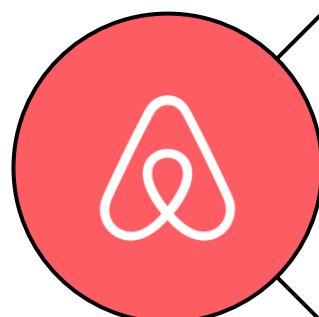
## Cas d'utilisations réels



Recommandation de contenu en temps réel.



Analyse des données de trajets pour l'optimisation de l'algorithme de tarification.



Analyse des données des utilisateurs pour améliorer l'expérience client.

## Applications industrielles



Finance : Détection de fraudes en temps réel.



Santé : Analyse de données génomiques.

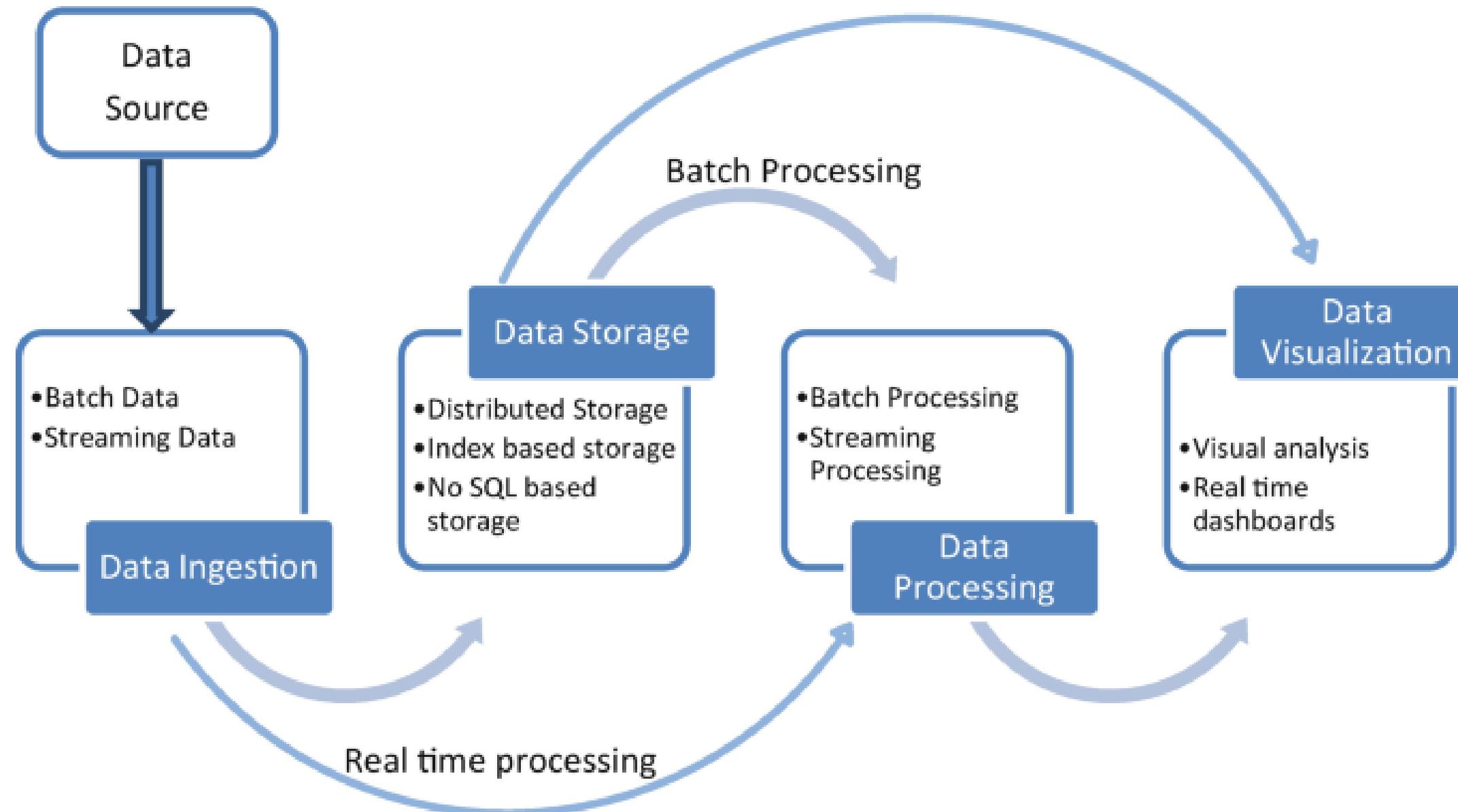


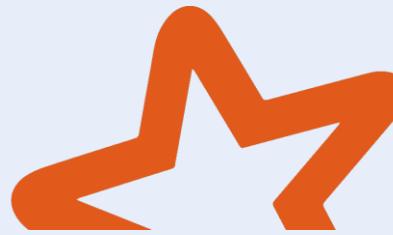
Télécommunications : Analyse de données de réseau pour la détection de pannes et l'optimisation de la performance.



# Cas d'Utilisation et Applications

## Cas d'utilisations réels en temps réel

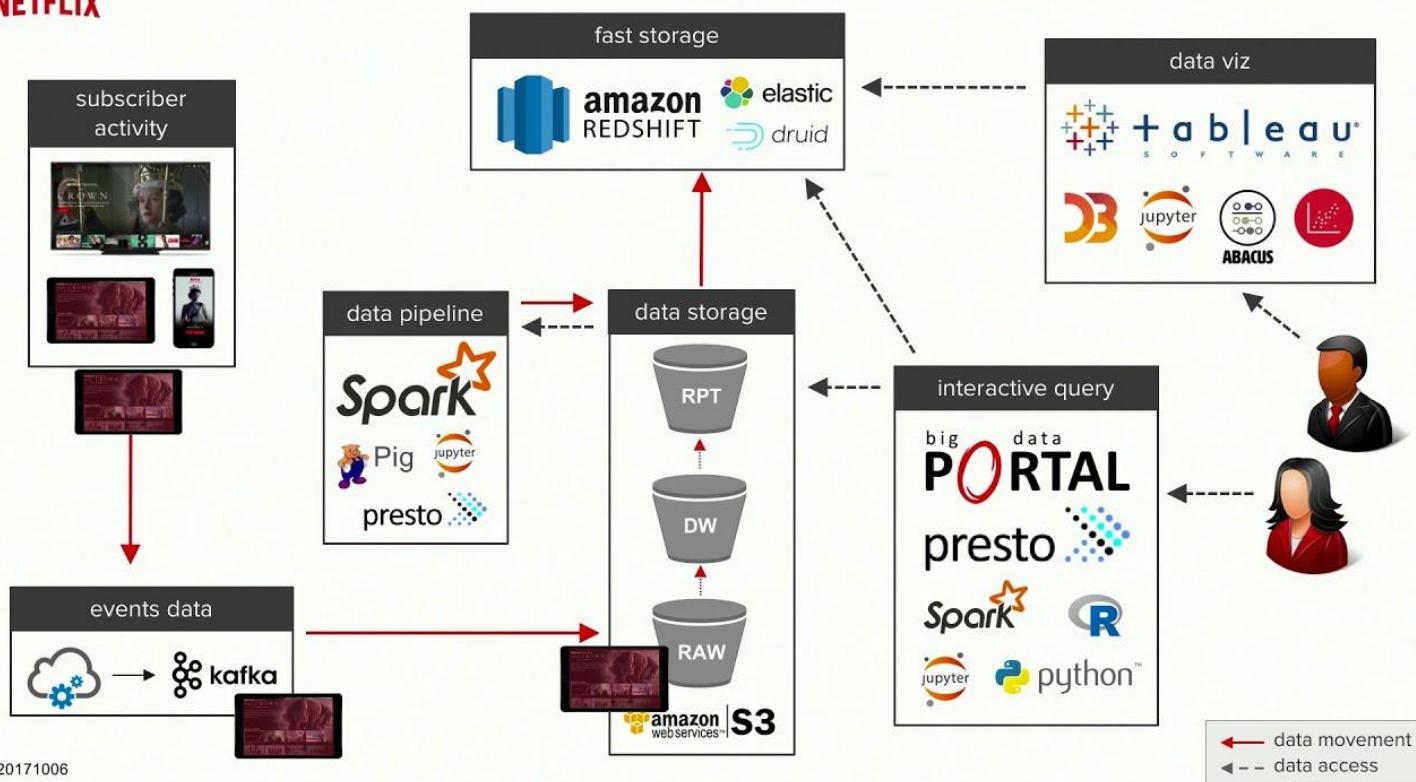




# Cas d'Utilisation et Applications

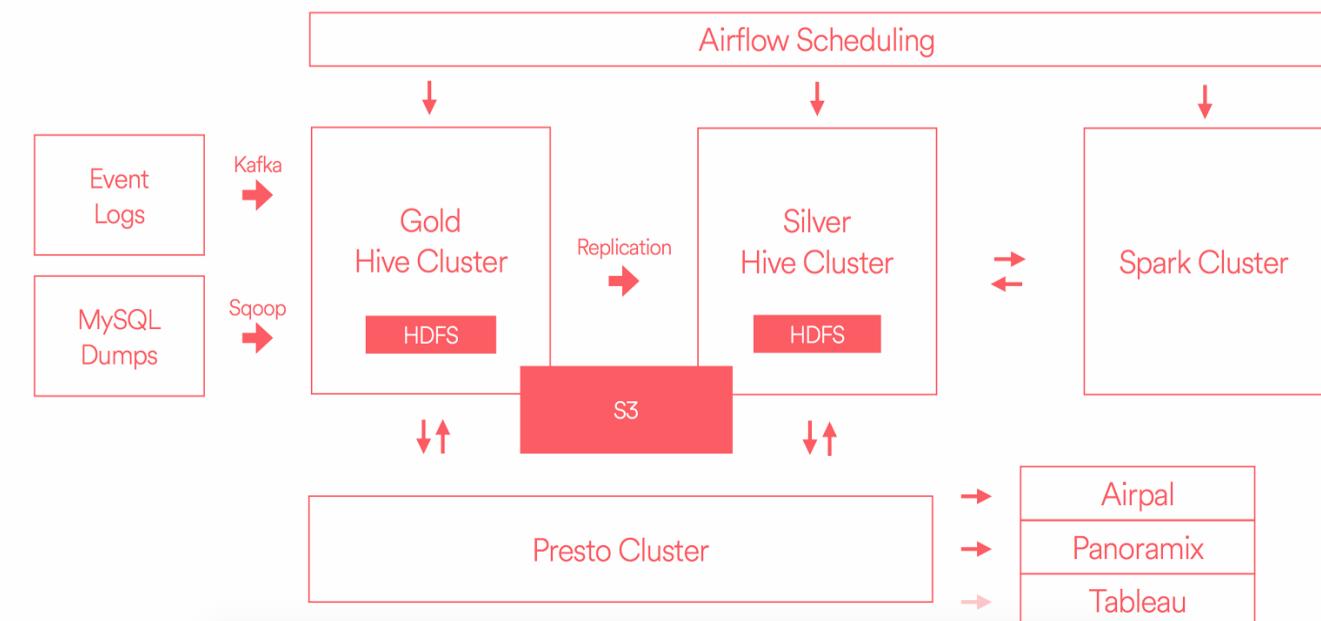
## Exemples de Cas d'utilisations réels

NETFLIX



20171006

AIRBNB DATA INFRA





## Avantages

Performances élevées

Flexibilité et extensibilité

Support de langages multiples

Large communauté et écosystème

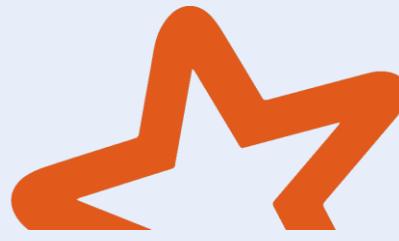
## Inconvénients

Complexité de la configuration

Ressources requises

Courbe d'apprentissage

Dépendance à Java ou Scala



# Alternatives à Spark



## Hadoop MapReduce

Hadoop MapReduce est un framework de traitement de données en lots qui a été largement utilisé avant l'arrivée de Spark. Il est fiable mais peut être lent pour certains types de tâches de traitement.

## Flink

Apache Flink est une plateforme de calcul distribué open source conçue pour le traitement de flux de données en temps réel. Il est connu pour ses performances élevées et son support pour la programmation de fenêtres de temps.

## Storm

Apache Storm est un framework de traitement de flux de données en temps réel distribué, conçu pour le traitement de flux de données à haut débit. Il est souvent utilisé pour des applications de streaming telles que le suivi des événements, l'analyse en temps réel et les systèmes de recommandation.



## Gestion de Données avec Spark et Hive : De la Connexion au Cluster à l'Exploration des Données

1. **Configuration de l'environnement** : Cluster Big Data Hadoop, Spark, Hive (Postgres) et Jupyter
2. **Chargement des données dans Hadoop** : Charger le fichier 'demo.csv' dans Hadoop en utilisant la commande `hadoop fs -put`
3. **Transformation des données avec Spark** : Transformer les données en les nettoyant et en convertissant la colonne `age` en `int`
4. **Stockage des données transformées dans Hive** : Stocker les données dans une table '`adulte`' en utilisant la methode `write.mode()` et l'option `saveAsTable()`
5. **Interrogation des données avec SQL** : Utiliser la methode `spark.sql()` pour afficher les adultes
6. **Visualisation des données** : Utiliser la bibliothèque `matplotlib` pour créer un graphique à partir de vos données



## Configuration de l'environnement

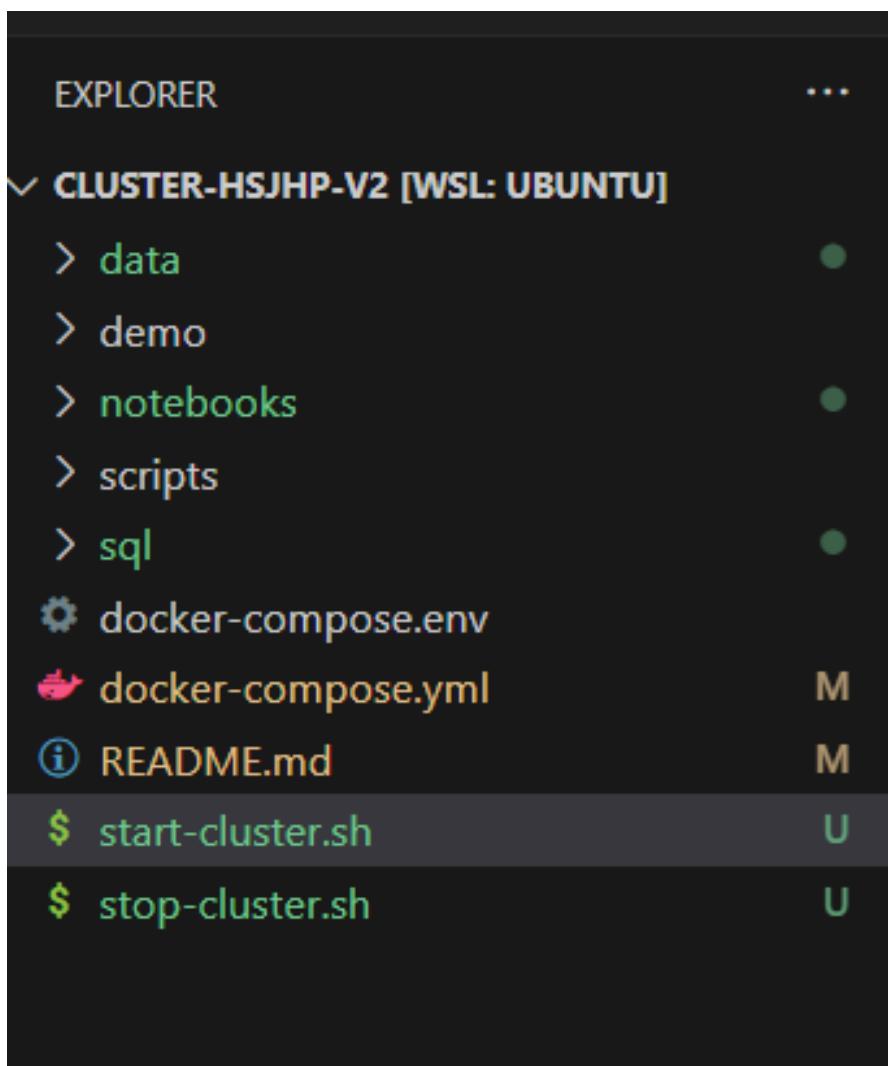
```
barryma@DESKTOP-LBFN3RA ~ $ conda deactivate
barryma@DESKTOP-LBFN3RA ~ $ docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
NAMES           STATUS          PORTS
hive-server     Up 13 minutes  0.0.0.0:10000->10000/tcp, 10002/tcp
spark-worker-2  Up 13 minutes  0.0.0.0:8083->8081/tcp
spark-worker-1  Up 13 minutes  0.0.0.0:8081->8081/tcp
cluster-hsjhp-v2-postgres-1 Up 13 minutes  5432/tcp
namenode        Up 13 minutes (healthy) 0.0.0.0:9870->9870/tcp, 0.0.0.0:9010->9000/tcp
resourcemanager Up 12 minutes (healthy) 0.0.0.0:8088->8088/tcp
spark-master    Up 13 minutes  0.0.0.0:7077->7077/tcp, 6066/tcp, 0.0.0.0:8080->8080/tcp
hive-metastore-postgresql Up 13 minutes  5432/tcp
nodemanager     Up 13 minutes (healthy) 0.0.0.0:8042->8042/tcp
jupyter         Up 13 minutes (healthy) 4040/tcp, 0.0.0.0:8888->8888/tcp
hive-metastore  Up 13 minutes  10000/tcp, 0.0.0.0:9083->9083/tcp, 10002/tcp
historyserver   Up 13 minutes (healthy) 0.0.0.0:8188->8188/tcp
datanode        Up 13 minutes (healthy) 0.0.0.0:9864->9864/tcp
```

```
barryma@DESKTOP-LBFN3RA ~ $ docker stats --no-stream
CONTAINER ID  NAME          CPU %      MEM USAGE / LIMIT      MEM %      NET I/O            BLOCK I/O
/0           PIDS          0.87%      328.8MiB / 7.623GiB  4.21%      7.04kB / 7.34kB    0B / 0B
9b2836792a1a  hive-server   0.13%      275.3MiB / 7.623GiB  3.53%      20.5kB / 149kB    0B / 0B
ca3468fcacd1  spark-worker-2 0.15%      270.6MiB / 7.623GiB  3.47%      20.5kB / 149kB    0B / 0B
1301bf357cc6  spark-worker-1 0.00%      21.41MiB / 7.623GiB  0.27%      2.21kB / 0B       0B / 0B
e3398cb30145  cluster-hsjhp-v2-postgres-1 0.23%      297.9MiB / 7.623GiB  3.82%      338kB / 301kB     0B / 0B
462cbaa2ff95  namenode      0.78%      475.1MiB / 7.623GiB  6.09%      588kB / 188kB     0B / 0B
5f82f7ba5c43  resourcemanager 0.14%      283.3MiB / 7.623GiB  3.63%      300kB / 36.8kB    0B / 0B
f038123e647d  hive-metastore-postgresql 0.00%      45.49MiB / 7.623GiB  0.58%      183kB / 153kB     0B / 0B
95cbfe4de84e  nodemanager   0.00%      148.1MiB / 7.623GiB  1.90%      226kB / 458kB     0B / 0B
c3cc66dcd278  jupyter       0.89%      382MiB / 7.623GiB   4.89%      159kB / 183kB     0B / 0B
7f304b6df54c  hive-metastore 0.24%      266MiB / 7.623GiB   3.41%      4.26kB / 1.95kB   0B / 0B
bec5b19f5ca7  historyserver 0.95%      284.2MiB / 7.623GiB  3.64%      73.2kB / 304kB    0B / 0B
```

Container CPU usage ⓘ		Container memory usage ⓘ	
8.07% / 800% (8 CPUs available)		3.3GB / 7.44GB	
Name	Image ↑	Status	Port(s)
cluster-hsjhp-v2	bde2020/hadoop-namenode:2.0.0	Running (13/1)	
datanode	bde2020/hadoop-datanode:2.0.0	Running	9864:9864 ⓘ
historyserver	bde2020/hadoop-historyserver:2	Running	8188:8188 ⓘ
namenode	bde2020/hadoop-namenode:2.0.0	Running	9010:9000 ⓘ
<a href="#">Show all ports (2)</a>			
nodemanager	bde2020/hadoop-nodemanager:2	Running	8042:8042 ⓘ
resourcemanager	bde2020/hadoop-resourcemanager:2	Running	8088:8088 ⓘ
hive-metastore	bde2020/hive:2.3.2-postgresql-m	Running	9083:9083 ⓘ
hive-server	bde2020/hive:2.3.2-postgresql-m	Running	10000:10000 ⓘ
hive-metastore-postg	bde2020/hive-metastore-postgre:2	Running	
spark-master	bde2020/spark-master:3.3.0-had	Running	7077:7077 ⓘ
spark-worker-1	bde2020/spark-worker:3.3.0-had	Running	8081:8081 ⓘ
spark-worker-2	bde2020/spark-worker:3.3.0-had	Running	8083:8081 ⓘ
jupyter	jupyter/pyspark-notebook:python	Running	8888:8888 ⓘ
postgres-1	postgres:13	Running	



## Chargement des données dans Hadoop



The screenshot shows a terminal window with the following content:

```
cluster-hsjhp-v2 [WSL: Ubuntu]
← → ⌂ cluster-hsjhp-v2 [WSL: Ubuntu]
docke docker-compose.yml M $ start-cluster.sh U X README.md M

$ start-cluster.sh
1 #!/bin/bash
2
3 docker-compose up -d
4 echo "LANCEMENT DU CLUSTER....." && sleep 120
5
6 echo "STOCKAGE DES DONNEES....."
7 docker exec namenode hdfs dfs -mkdir /data
8 docker exec namenode hdfs dfs -put /opt/data/demo.csv /data/
9
10 echo "BIENVENUE"
11 sed -n 13,32p README.md
```

```
● (base) barryma@DESKTOP-LBFN3RA:~/Workspace/cluster/cluster-hsjhp-v2$ docker exec namenode hdfs dfs -mkdir /data
● (base) barryma@DESKTOP-LBFN3RA:~/Workspace/cluster/cluster-hsjhp-v2$ docker exec namenode hdfs dfs -put /opt/data/demo.csv /data/
2024-07-19 12:31:12,502 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhosttrusted = false, remotehosttrusted = false
```



# Transformation des données avec Spark

This screenshot shows the left panel of a Jupyter Notebook interface. The sidebar displays a file tree with a notebook named 'demo.ipynb' selected. The main area contains two code cells:

```
[1]: from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
import re

# Create a SparkSession
spark = SparkSession.builder \
    .appName("demo-app") \
    .enableHiveSupport() \
    .getOrCreate()

[5]: # Read the data from Hadoop
df = spark.read.format("csv").option("header", "true").load("hdfs://namenode:9000/data/demo.csv")

print(df)
DataFrame[name: string, age: string, city: string]
```

The notebook title is 'Jupyter Notebook' and the first section is 'Step 1: Configuration de l'environnement'. The second section is 'Step 2: Transformation des données avec Spark'.

This screenshot shows the right panel of a Jupyter Notebook interface. The sidebar displays a file tree with a notebook named 'demo.ipynb' selected. The main area contains several code cells:

```
[15]: # Define a UDF to clean the data
def clean_data(s):
    # Remove all non-alphabetic characters
    s = re.sub(r'[^a-zA-Z]', '', s)
    # Remove all extra spaces
    s = re.sub(r'\s+', ' ', s).strip()
    return s

clean_data_udf = udf(clean_data)

# Assuming df is your DataFrame and 'column_to_clean' is the column you want to clean
df_clean = df.withColumn('clean_data', clean_data_udf(df['name']))

df_clean = df_clean.withColumn("age",
                               df_clean["age"]
                               .cast('int'))

df_clean.printSchema()

root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- city: string (nullable = true)
 |-- clean_data: string (nullable = true)

[16]: # Perform a simple transformation
transformed_df = df_clean.filter(df.age > 18)

print(transformed_df)
DataFrame[name: string, age: int, city: string, clean_data: string]
```



## Stockage des données transformées dans Hive & Interrogation des données avec SQL

The screenshot shows the JupyterLab interface with a notebook titled "demo.ipynb". The left sidebar displays a file tree with the following structure:

- / notebooks /
- Name      Last Modified
- metastore\_db      an hour ago
- spark-wareh...      an hour ago
- demo.ipynb      an hour ago
- derby.log      an hour ago

The main content area contains two code cells:

**Step 3: Stockage des données transformées dans Hive**

```
[17]: # Write the transformed DataFrame to Hive  
transformed_df.write.mode("overwrite").saveAsTable("adulte")
```

**Step 4: Interrogation des données avec SQL**

```
[18]: # Query the data from Hive  
result = spark.sql("SELECT * FROM adulte LIMIT 10")  
  
# Show the result  
result.show()
```

The output of the second code cell is a table:

name	age	city	clean_data
John	23	New York	John
Anna	45	London	Anna
Peter	36	Paris	Peter
Barry	27	Dakar	Barry
Linda	78	Berlin	Linda
Tom	22	Sydney	Tom



## Visualisation des données

Screenshot of JupyterLab interface showing a data visualization step.

The interface includes a file browser on the left and a code editor/terminal on the right.

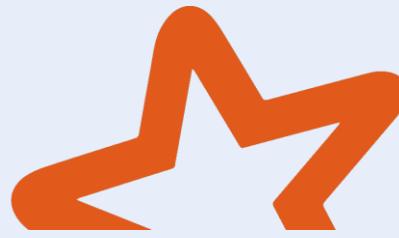
**File Browser:** Shows the directory structure under "notebooks".

**Code Editor:** Displays the following Python code in the "demo.ipynb" notebook:

```
[19]: import matplotlib.pyplot as plt  
  
# Convert the Spark DataFrame to a Pandas DataFrame  
result_pd = result.toPandas()  
  
# Plot the data  
result_pd.plot(kind='bar', x='name', y='age')  
plt.show()
```

**Output:** A bar chart titled "Step 5: Visualisation des données" showing age versus name. The x-axis is labeled "name" and lists "John", "Anna", "Peter", "Barry", "Linda", and "Tom". The y-axis is labeled "age" and ranges from 0 to 80. The chart shows Linda has the highest age at approximately 78, followed by Anna at approximately 45, Peter at approximately 37, Barry at approximately 28, John at approximately 23, and Tom at approximately 22.

name	age
John	23
Anna	45
Peter	37
Barry	28
Linda	78
Tom	22



# Merci de votre attention !

## 1 Résumé

Apache Spark est une plateforme puissante et flexible pour le traitement de données à grande échelle. Il offre des performances élevées, une large gamme de cas d'utilisation et une communauté active. Sa capacité à gérer des volumes importants de données et à exécuter des calculs de manière efficace en fait un choix populaire pour les entreprises et les organisations qui s'occupent de big data.

## 2 Tendances Futures

Le domaine du big data continue d'évoluer, avec des tendances émergentes telles que l'intelligence artificielle, l'apprentissage automatique, l'Internet des objets (IoT) et les données en temps réel. Spark est bien placé pour s'adapter à ces tendances et continuer à jouer un rôle majeur dans l'analyse et le traitement des données à grande échelle.

## 3 Sources

- <https://www.edureka.co/blog/spark-sql-tutorial/>
- <https://spark.apache.org/>
- <https://stackoverflow.com/>