# Spark

## Introduction

• **Spark :** framework qui développé en réponse aux problèmes de Big Data

• **Problèmes adressés par Spark :**

— manipulation massive de données

— analyse en temps réel de données

— adaptabilité et scalabilité

• **Apache Spark :** framework Open Source de traitement de données

• **Histoire d’Apache Spark :**

— développé par l’université de Berkeley en 2009

— développé pour surmonter les problèmes des solutions existantes (Hadoop, MapReduce, etc.).

— repris par la fondation Apache en 2013

— vise à fournir une solution de traitement de données beaucoup plus rapide et plus facile à utiliser que celles existantes

• Spark est utilisé pour résoudre divers problèmes liés Big Data :

— analyse de données en temps réel

— analyse des objets connectés

— analyse bio-informatique et génomique

— analyse astronomique

— apprentissage automatique

— etc.

## Caractéristiques

### Distribué et scalable

• Spark est un système **distribué** qui traite les données sur plusieurs machines.

• **Cluster :** ensemble de ces machines qui communiquent entre elles

• **Nœud :** machine d’un cluster

• Il est également facilement **scalable** et **flexible** : il permet de :

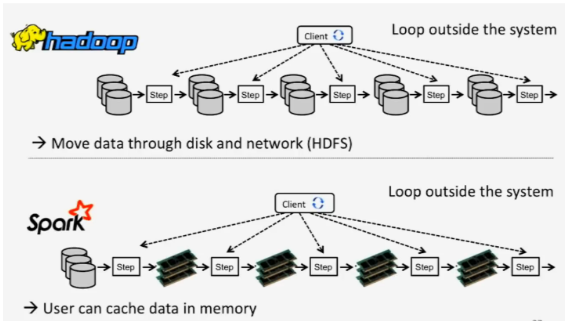
— ajouter des nœuds pour augmenter la capacité de traitement

— supprimer des nœuds inutilisés pour réduire les couts

### Rapide et performant

• Spark effectue le traitement des données **en mémoire**: les données sont stockées et manipulées directement dans la mémoire vive de l’ordinateur.

• Cela permet un traitement très rapide des données sans avoir besoin de les écrire sur un support entre chaque étape.



• Spark utilise le moteur d’exécution haute performance DAG (Directed Acyclic Graph) pour accélérer le traitement des données parallélisées.

### Facile d’utilisation

• Spark est conçu pour être facile à utiliser.

• Il propose des API simples et intuitives.

• Il est supporté dans plusieurs langages de programmation, dont Python, Scala, Java, R, SQL, etc.

• Il est possible d’utiliser Spark avec des notebooks comme Jupyter ou dans un terminal interactif via l’outil spark-shell.

### Limites de Spark

• **Configuration** **complexe** des clusters qui demande une connaissance en administration système avancé

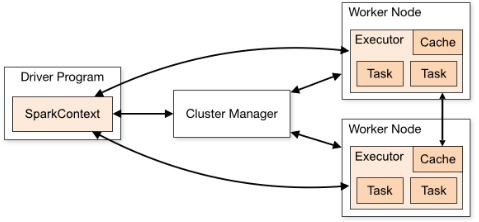
• **Consommation de mémoire de travail importante :** le stockage et traitements en mémoire vive par Spark peut être assez gourmands en mémoire

• **Latence :** le transfert de données d’un nœud à l’autre peut être long et ralentir les traitements itératifs

## Architecture

• Spark est un système de traitement de traitement distribué.

• Il nécessite de nombreux composants pour fonctionner. Ils sont réunis dans un cluster



• Le travail de Spark se base sur une architecture de type **maitre-esclave :**

— le maitre est le point d’entrée de l’application Spark

— les esclaves sont les machines de traitement

• Dans cette architecture, on trouve : le driver, les tâches à accomplir, les workers, le cluster manager, le contexte, etc.

### Le Driver

• C’est le point d’entrée de l’application Spark.

• Le nœud driver est le coordinateur de travail à effectuer.

• C’est lui qui créé et détient le contexte Spark responsable de la soumission des tâches aux workers.

• Note : sorte de chef qui planifie et suit l’exécution des tâches.

### Les Workers

• Ce sont les nœuds machines qui effectuent le travail.

• Un worker possède une mémoire dédiée au stockage des données et des partitions de travail.

• C’est lui qui effectue le travail d’une tâche à accomplir.

### Le Cluster Manager

• C’est le gestionnaire de ressources du cluster.

• Le cluster manager est responsable de gérer l’état des workers, la répartition et la planification des tâches.

• Spark peut utiliser différents cluster manager :

— Spark Standalone (manageur local)

— Apache Mesos (manageur distribué)

— Hadoop YARN (manageur distribué)

— Kubernetes (manageur distribué)

### Le contexte

• C’est l’interface entre le driver et les workers.

• Il permet de **paramétrer le cluster** et de soumettre les tâches aux workers avec l’aide du cluster manager.

• Le contexte est le point d’entrée qui récupère les données à traiter et le travail à effectuer.

### Une tâche

• C’est une **unité de traitement** de données.

• Une tâche est créée par le driver à partir des données sources et du programme à exécuter.

• L’exécution d’un programme est donc une suite de tâches à effectuer sur une suite de données.

### A

• Le **driver** reçoit les données et le traitement.

• Le **contexte** définit le cluster et les tâches à effectuer.

• les tâches sont réparties sur les workers sous la supervision du **cluster manager**.

• Les **workers** effectuent les tâches et retournent les résultats au driver.

ABCDE

## Fonctionnement

• Le principe de travail de Spark est de faire du traitement de données massivement et rapidement.

• Pour réaliser cet objectif, Spark utilise la **parallélisation**.

### Parallélisation

• La parallélisation permet de **diviser** un problème en plusieurs sous-problèmes.

• Puis de les résoudre en même temps (parallèle) sur différents processeurs.

• Le plus souvent réparti sur plusieurs machines : le cluster.

• Cette **parallélisation** est réalisée par :

— la division des données en Resilient Distributed Dataset (RDD)

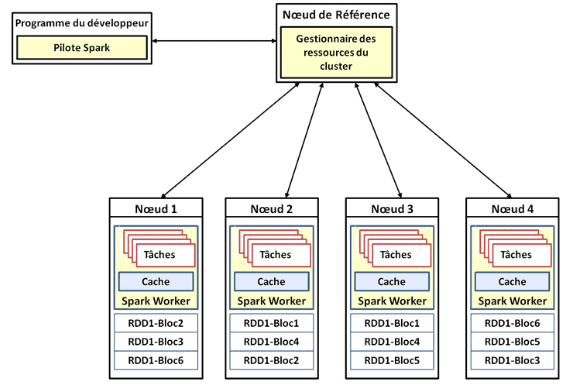
— la division du programme par le Direct Acyclic Graph

### Resilient Distributed Dataset (RDD)

• Pour améliorer le traitement des données, Spark manipule les informations sous un format divisé en blocs d’informations réparties sur les différents nœuds.

• C’est le Resilient Distributed Dataset (RDD).

• Les blocs de données sont appelés **partitions** et sont stockées en plusieurs exemplaires sur les différents nœuds du cluster.

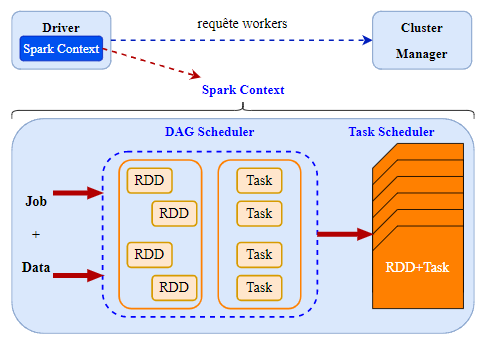


### Direct Acyclic Graph (DAG)

• Lorsque Spark reçoit un programme, celui-ci va être également transformé.

• Le DAG va permettre de **diviser le travail** à accomplir en action à effectuer les unes à la suite des autres.

• Une tâche est ensuite planifiée pour une action sur un bloc de données



### Résumé

• Spark va diviser les données en partitions et les répartir sur les différents nœuds du cluster.

• Puis va diviser le programme en tâches à effectuer sur les partitions.

• Ensuite, il va répartir les tâches sur les différents nœuds du cluster.

• Le travail peut alors être effectué en même temps.

### Spark : flux de données

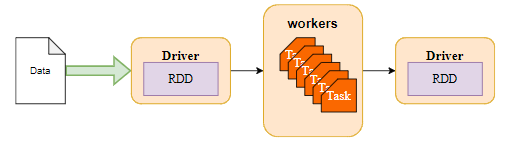
• Le travail de traitement des données par Spark peut se faire selon différentes configurations :

— traitement par lots (batch)

— traitement en micro-batch

— traitement en temps réel (streaming)

### Traitement par lots (batch)



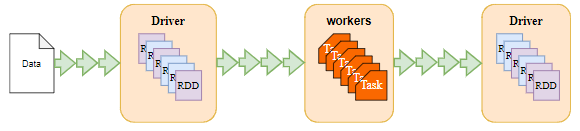
• Consiste à charger en mémoire l’ensemble des données à traiter.

• Une seule RDD sur les différents nœuds du cluster (stocké et traité de manière distribuée).

• Le traitement s’effectue une fois le chargement complet des données effectuées.

• L’ensemble est ensuite regroupé sur un seul RDD.

### Le traitement en micro-batch



• Le traitement en micro-batch consiste à charger les données par petits lots appelés **micro-batch** au lieu de tout charger d’un coup.

• Intérêt : traitement au fur et à mesure par petits groupe sans attendre d’avoir tout récupéré.

• Le traitement en micro-batch est particulièrement adapté aux données très volumineuses ou aux données incrémentales.

### Le traitement en temps réel (streaming)

• Le traitement en temps réel consiste à traiter les données au fur et à mesure de leur arrivée.

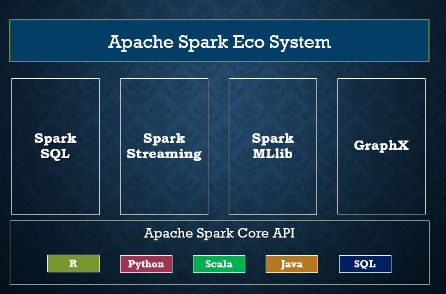
• C’est le traitement le plus rapide car les données sont traitées au fur et à mesure de leur arrivée.

• En pratique, le traitement en temps réel est une agrégation des données traitées en micro-batch en continu.

ABCDE

## Modules Spark

• Spark est subdivisé en différents modules répondant chacun à un besoin spécifique :



### Spark Core

• Module qui fournit les fonctionnalités de base de Spark.

• Fournit l’ensemble des fonctionnalités permettant de :

— définir les inputs / outputs du job Spark

— manipuler et transformer les RDD

— lancer des jobs Spark

— établir le contexte

• Il contient toutes les API permettant de piloter ces fonctionnalités dans les différents langages de programmation supportés par Spark : R, Python, Scala, Java, SQL

### Spark SQL

• **Spark SQL :** module qui permet de manipuler des données structurées.

• Il permet de lire et de manipuler des données avec des fonctionnalités de requêtage identique à celles retrouvées dans les requêtes SQL.

• Les données seront structurées sous une forme de structure multidimensionnelle rappelant les tables SQL appelée **DataFrame**.

• Les manipulations, appelées **Transformation**, seront effectuées de manière distribuée sur les nœuds du cluster.

• Les fonctions de transformation sont utilisables dans les différents langages de programmation supportés par Spark.

### Spark streaming

• Spark streaming : module qui permet le traitement des flux de données.

• Il permet de prendre en charge les données issues de source en temps réel (kafka, Flume, Kinesis, sockets, réseaux, etc.) et de les traiter en temps réel.

• Couplé avec les fonctionnalités de Spark SQL, il permet de traiter les données en temps réel et de les manipuler avec les mêmes fonctionnalités que celles de Spark SQL.

• Les flux de données définis sous une forme de séquences de RDD appelé **DStreams** (Discretized Streams).

• Les données sont ensuite injectées et traitées par petits lots appelés **micro-batches**.

### Spark MLlib

• Spark MLlib : module qui permet de fournir des fonctionnalités de machine learning distribuées.

• Librairie adaptée pour le traitement distribué de données volumineuses nécessaires aux pipelines d’apprentissage du machine learning.

• Il inclut des fonctionnalités de : traitement de données, classification, clustering, régression, normalisation.

### GraphX

• Spark GraphX : module qui permet de manipuler des données sous forme de graphes.

• Il contient des fonctionnalités permettant de gérer, manipuler des données graphes en les distribuant.

• Son rôle est de fournir une infrastructure de gestion des données graphes de grandes tailles, souvent couplées à des algorithmes de machine learning pour les analyser.

## Session et contexte

• Pour démarrer une application Spark, il faut créer une session Spark et lui fournir un contexte Spark.

• La session représente l’objet d’interface avec le cluster.

• Il contient le contexte contenant les paramètres et configurations du cluster.

### Session

• **SparkSession :** c’est une instance de travail d’une application Spark permettant d’interagir avec le cluster.

• Le rôle de la session Spark est de fournir un point d’entrée à l’application.

• C’est l’interface entre l’utilisateur et le cluster.

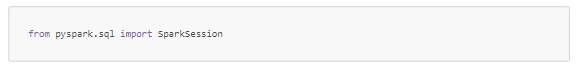
• Il en existe deux types :

— Spark Session SQL : pour les traitements batch

— Spark Session Streaming : pour les traitements en temps réel

• C’est à partir de la session que sont chargées et manipulées les données.

• Il existe sous forme d’un objet de type **SparkSession** importé du module **spark.sql**.

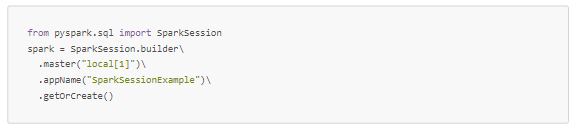


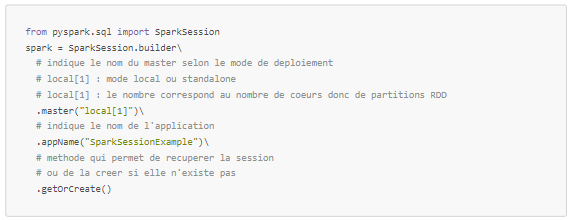
• La création d’une session peut se faire :

— automatiquement (par défaut, environnement de développement, etc.)

— manuellement (environnement de production, etc.)

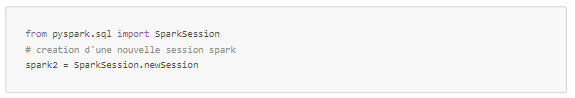
• Créée à partir de la méthode **builder()** de la classe SparkSession.





• Il est possible de créer plusieurs sessions.

• La méthode **newSession()** permet de créer une nouvelle sessions possédant les mêmes configurations (nom, master, context) que la session courante.



• Par défaut, la SparkSession va créer :

— un objet **SparkContext**

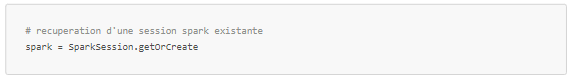
— un objet **SparkConfig**

— tous les autres contextes, selon la configuration (HiveContext, StreamingContext, etc.).

• Remarque :

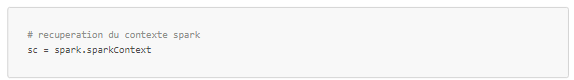
— Dans de nombreux environnements (pyspark-shell, databricks, etc.), la session est créée automatiquement avec son contexte.

— Elle peut alors être utilisées directement à partir de l’objet **spark**.



— Sans indication contraire, le contexte est créé automatiquement à partir de la création de la session.

— Il est accessible à partir de l’objet **sc** ou la propriété **spark.sparkContext**.



• Au besoin, des configurations peuvent être ajoutées à la session.



• Les configurations peuvent être ajoutées ou récupérées à partir de la propriété **spark.conf**.

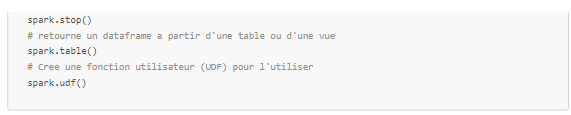


• Dans le cas d’utilisation de Hive (moteur de gestion de données distribuées), il est possible d’ajouter le contexte.



• Méthodes les plus communes :

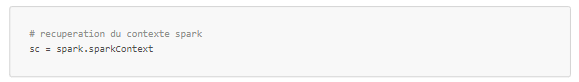




### Contexte

• **SparkContext :** c’est un objet de la SparkSession qui contient les configurations et informations de l’application Spark.

• Le contexte est accessible à partir de la propriété **spark.sparkContext** lorsqu’il n’est pas créé autoamtiquement dans l’objet **sc**.



• Le SparkContext contient toutes les configurations de l’application.

• C’est lui qui soumet les tâches au cluster.

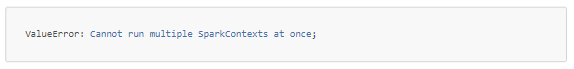
• C’est également lui qui récupère les informations sur l’état du cluster.

• Le SparkContext est lié à l’exécution de Spark par la JVM (Java Virtual Machine).

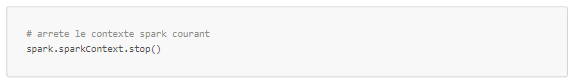
• Il ne peut y avoir qu’**une seule instance** en même temps.

• S’il y a plusieurs sessions en cours, elles partagent le même contexte.

• Si plusieurs contextes sont créés, un message d’erreur est affiché :



• Il est possible d’arrêter le contexte avec la méthode **stop()**.



• Attention : l’exécution va déclencher un message dans les logs Spark qui n’est plus utilisable tant qu’un context n’a pas été recréé.



• Variables et méthodes les plus communes :



• Note :

— une variable broadcast est une variable distribuée à tous les nœuds du cluster.

— wholeTextFiles() garde la liste des shards de fichiers et leur contenu, à la différence de textFile() qui regroupe en un seul RDD.

## RDD et Dataframe

## Utilisation RDD

## Utilisation de PySpark

### Session Spark

• Remarque : utiliser « from pyspark.sql import SparkSession »

• **Initialiser une session Spark :**

session = SparkSession.builder \

.appName("nom\_application")

.getOrCreate()

• **Fermer la session Spark :** session.stop()

### CSV

• **Charger un fichier CSV avec un délimitateur de base (la virgule) :**

data = session.read.csv("data/covid\_de.csv", header=True, inferSchema=True)

• **Charger un fichier CSV avec un point-virgule comme délimitateur :**

data = session.read.option("delimiter", ";").csv("data/car.csv", header=True, inferSchema=True)

• **Enregistrer les données dans un fichier CSV :** dataframe.write.csv("chemin/vers/fichier\_de\_sortie.csv", header=True)

### DataFrames

data.printSchema() # afficher le schéma

data.createOrReplaceTempView("covid") # créer une vue temporaire en la nommant

récupérer une colonne

df.select("colonne").show()

df.select(df.columns[index]).show()

df. select(df.City).show() (avec City étant le nom de la colonne)

df.select(F.col("colonne")).show()

col = F.col("colonne")

df.select(col).show()

withColumn

concaténation

actualisation de la colonne en minuscule

renommage de la colonne A en colonne B (ou withColumnRenamed)

changer le type d’une valeur

df["colonne"].cast("string")

.withColumn(….).withColumn(…).WithColumn(…)

.withColumn("colonne", F.when(df["colonne"] == "valeur", 1).when(……., 2).when(…, 3).otherwise(0) # when équivalent en python d’un « si », otherwise équivalent du « else »

when(condition, valeur\_si\_vraie)

otherwise(valeur\_si\_faux)

spark.createDataFrame(data = data, schema = columns)

// grouper par fruit

// transposer notre colonne « Provenance » en plusieurs

// faire la somme de « Quantité » par « Provenance » pour chaque « Fruit »

df.groupBy("Fruit").pivot("Provenance").sum("Quantite").show()

### Requêtes

### A

A

•

—