L'architecture de Spark et ses composants clés

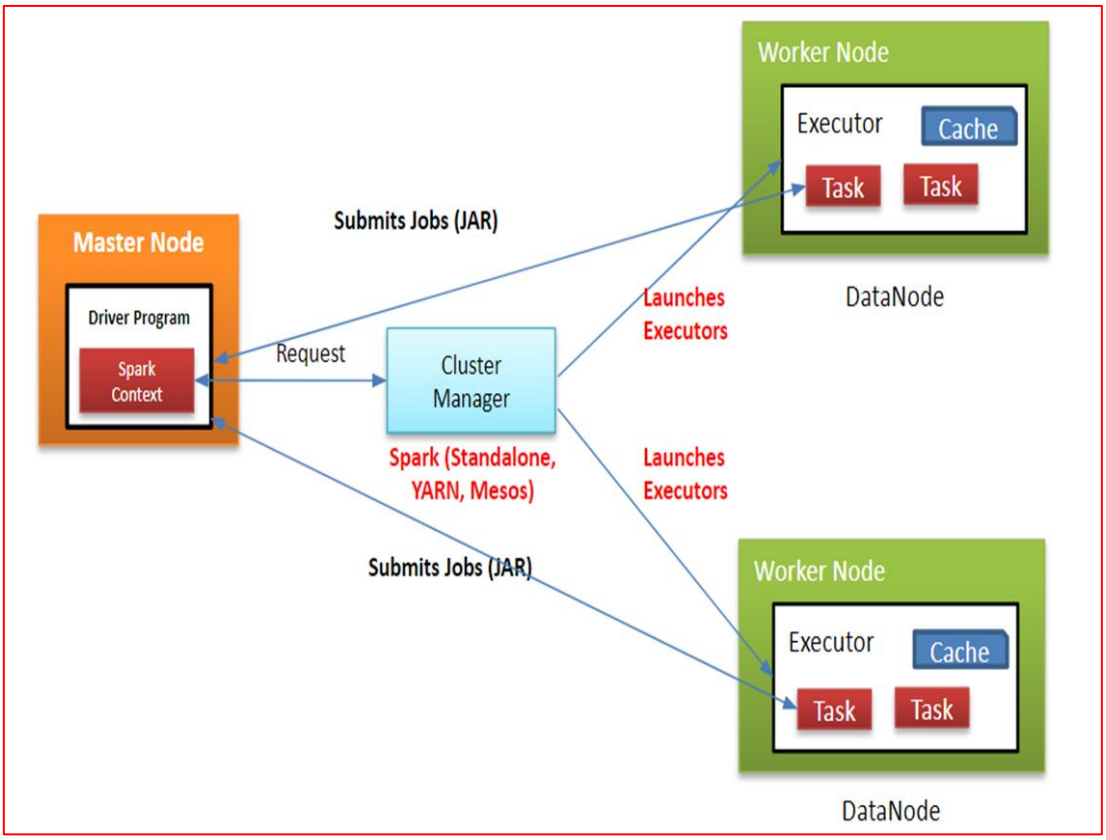

Figure 1. L'architecture de Spark et ses composants clés [1]

Architecture de Spark : Composants et Interactions

1. Master Node (Nœud Maître)

- **Driver Program** :
  - Cœur de l'application Spark.
  - Contient la logique métier (ex : transformations, actions).
  - Convertit le code utilisateur en **DAG (Directed Acyclic Graph)** pour l'optimisation.
- **Spark Context** :
  - Point d'entrée principal pour interagir avec le cluster.
  - Coordonne les ressources et communique avec le **Cluster Manager**.

2. Cluster Manager

- **Rôle** : Gère les ressources du cluster (mémoire, CPU) pour exécuter les applications Spark.
- **Types supportés** :
  - **Standalone** (mode natif de Spark),
  - **YARN** (pour l'intégration Hadoop),
  - **Mesos** (gestionnaire de cluster générique).
- **Fonctionnement** :
  - Reçoit les demandes de ressources du **Driver Program**.
  - Alloue des **Worker Nodes** pour exécuter les tâches.

3. Worker Node (Nœud Travailleur)

- **Executor** :
  - Processus lancé sur chaque Worker Node.
  - Exécute les **Tasks** (unités de travail) et conserve les données en **Cache** (mémoire/disk).
  - Communique les résultats au Driver.
- **Task** :
  - Unité minimale de traitement.
  - Une partition de données est traitée par une Task.
  - Exemple : Appliquer une fonction map() sur un RDD.

4. DataNode (Hadoop HDFS)

- **Rôle** : Stocke les données distribuées (blocs HDFS).
- **Intégration avec Spark** :
  - Spark **lit/écrit** des données depuis/vers HDFS.
  - Les **Worker Nodes** accèdent aux **DataNodes** pour charger les données à traiter.
- **Cache** :
  - Les données fréquemment utilisées sont mises en cache dans les Executors pour éviter des lectures répétées depuis HDFS.

Flux d'Exécution d'un Job Spark

1. Le **Driver Program** soumet l'application (fichier JAR) au **Cluster Manager**.
2. Le Cluster Manager alloue des **Worker Nodes** et lance les **Executors**.
3. Le **Spark Context** divise le travail en **Stages** et **Tasks**.
4. Les **Tasks** sont distribuées aux Executors via le Cluster Manager.

5. Les Executors lisent les données depuis les **DataNodes**, exécutent les Tasks, et stockent les résultats intermédiaires en **Cache**.
6. Les résultats finaux sont retournés au Driver ou écrits dans HDFS.

Schéma Réel vs. Composants Hadoop

- **DataNode** :
  - N'est pas un composant natif de Spark, mais fait partie de **Hadoop HDFS**.
  - Spark peut s'interfacer avec HDFS pour le stockage distribué.
- **Pourquoi cette mention ?**
  - L'image (**Figure *1***) montre une intégration classique entre Spark (traitement) et HDFS (stockage).

Exemple Concret

Scénario : Calcul du nombre de mots dans un fichier stocké sur HDFS.

1. **Driver Program** : Définit la logique (**textFile()**, **flatMap()**, **reduceByKey()**).
2. **Cluster Manager** (YARN) : Alloue 3 Worker Nodes.
3. **Executors** :
   - Chargent les blocs du fichier depuis les **DataNodes**.
   - Exécutent les Tasks de comptage en parallèle.
   - Stockent les résultats partiels en **Cache**.
4. **Driver** : Agrège les résultats et affiche le total.

Points Clés à Retenir

- **Scalabilité** : Ajout facile de Worker Nodes pour traiter de gros volumes de données.
- **Tolérance aux pannes** :
  - Si un Executor échoue, ses Tasks sont relancées sur un autre Worker.
  - Les données perdues sont régénérées grâce au **lineage** des RDDs.
- **Performance** :
  - Le **Cache** réduit les accès coûteux à HDFS.
  - Le traitement en mémoire est 100x plus rapide qu'avec MapReduce.

I. Installer Spark sur Windows

1. Vérifier la compatibilité des versions

Avant de commencer, consultez ce tableau de compatibilité :

Composant	Version recommandée	Notes
Java JDK	JDK 8 ou JDK 11	Spark 3.x+ ne supporte pas JDK 17+ en local (risque d'erreurs).
Spark	3.5.5 (dernière stable)	Choisir un package "pre-built" pour Hadoop.
Hadoop	3.3.6	Version incluse dans Spark 3.5.0.

2. Installer Java JDK

Étapes :

1. **Télécharger JDK 11** :
   - **JDK 11 ou JDK 8:** https://www.openlogic.com/openjdk-downloads

2. **Installer JDK** en suivant les instructions par défaut.
3. **Configurer les variables d'environnement** :
   - JAVA_HOME : Chemin d'installation du JDK (ex: **C:\Program Files\Java\jdk-11.0.20**).
   - Ajouter %JAVA_HOME%**bin** à la variable Path.

3. Installer Apache Spark

Étapes :

1. **Télécharger Spark** (version précompilée pour Hadoop) :
   - **Spark 3.5.0 (Hadoop 3.3):** https://spark.apache.org/downloads.html
   - **Spark 3.5.5 (Hadoop 3.2.0):** https://spark.apache.org/downloads.html

2. **Extraire le fichier** dans un dossier sans espaces (ex: C:**Spark**\spark-3.5.5).
3. **Configurer les variables d'environnement** :
   - SPARK_HOME : Chemin du dossier Spark (ex: C:**Spark**\spark-3.5.5).
   - Ajouter %SPARK_HOME%**bin** à la variable **Path**.

4. Installer WinUtils (Obligatoire pour Windows)

Spark nécessite winutils.exe pour fonctionner sur Windows.

Étapes :

1. **Télécharger WinUtils** pour Hadoop 3.3.6 :
   - Lien : https://github.com/cdarlint/winutils → Choisir **hadoop-3.2.0**.

2. **Créer un dossier** C:**hadoop**\hadoop-3.2.0.
3. **Configurer les variables d'environnement** :

- HADOOP_HOME : C:**hadoop**\ hadoop-3.2.0.
- Ajouter %HADOOP_HOME%**bin** à la variable **Path**.

5. Configurer Python (Optionnel pour PySpark)

Si vous utilisez PySpark :

1. **Installer Python 3.8+**.
2. **Ajouter Python au Path** lors de l'installation.
3. **Installer pyspark** via pip :

```
pip install pyspark
```

6. Tester l'installation

Ouvrez **Invite de commandes** ou **PowerShell** et exécutez :

```
spark-shell  # Pour Scala/Java
```

ou

```
pyspark  # Pour Python
```

Résultat attendu : Un shell interactif **Spark** avec le **logo ASCII**.

Dépannage courant

- **Erreur "JAVA_HOME not set"** : Vérifiez les variables d'environnement.
- **Erreur "Could not locate executable null\bin\winutils.exe"** : Vérifiez HADOOP_HOME.
- **Problèmes de permissions** : Exécutez PowerShell en tant qu'administrateur.

La **relation** entre l'architecture **Spark** illustre dans l'image (**Figure 1**) et des exemples pratiques (**CSV, PDF/images**):

1. Master Node (Driver Program et Spark Context)

Rôle dans les exemples :

- C'est le point de départ de vos scripts Python (ex : **csv_to_hdfs.py**, **binary_files.py**).
- Le **Driver Program** contient votre code (lecture de CSV, traitement d'images, etc.).
- Le **Spark Context** :
  - Convertit votre code en tâches distribuées.
  - Gère la communication avec le **Cluster Manager** (ex : YARN) pour demander des ressources.

Exemple concret :

spark = SparkSession.builder.getOrCreate() # Initialise le Spark Context

df = spark.read.csv("**hdfs:///data.csv**") # Le Driver planifie la lecture depuis HDFS

2. Cluster Manager (YARN/Standalone/Mesos)

Rôle dans les exemples :

- Alloue les ressources (CPU, mémoire) pour exécuter vos jobs.
- Détermine quels **Worker Nodes** exécuteront les **Executors**.

Exemple concret :

- Quand vous écrivez **df.groupBy("age").agg(...)**, le Cluster Manager décide combien d'Executors sont nécessaires pour traiter les données.

3. Worker Node (Executors et Tasks)

Rôle dans les exemples :

- **Executors** :
  - Exécutent les **Tasks** (ex : lire un bloc de CSV, découper une image, extraire des métadonnées PDF).
  - Stockent les données en **Cache** (ex : après un df.persist()).
- **Tasks** :
  - Unités de travail parallélisées (ex : traiter 100 lignes de CSV, analyser une image).

Exemple concret :

Chaque partition du CSV est traitée par une Task sur un Executor

df = spark.read.csv("hdfs:///data.csv").repartition(8) **# → 8 Tasks**

4. DataNode (HDFS)

Rôle dans les exemples :

- Stocke les fichiers bruts (CSV, PDF, images) sous forme de blocs distribués.
- Les **Worker Nodes** accèdent directement aux blocs via le réseau.

Exemple concret :

Lecture depuis HDFS (blocs répartis sur les DataNodes)

images_rdd = spark.sparkContext.binaryFiles("**hdfs:///images/*.jpg**")

Relation Complète : Workflow d'un Exemple CSV

1. **Driver** (Master Node) :
   - Lit votre script **spark.read.csv(...)**.
   - Contacte le **Cluster Manager** pour demander 4 Executors.
2. **Cluster Manager** :
   - Alloue 4 **Worker Nodes** avec des Executors.
3. **Worker Nodes** :
   - Chaque Executor lit un bloc de 128 Mo du CSV depuis les **DataNodes**.
   - Exécute les Tasks (ex : **parsing, filtrage grade > 12**).
4. **Cache** :
   - Si vous faites df.persist(), les données intermédiaires restent en mémoire sur les Executors.
5. **Résultats** :
   - Les Executors envoient les résultats au **Driver**, qui écrit le résultat final dans HDFS.

Cas des Fichiers Binaires (PDF/Images)

- **Différence claire** :
  - Les fichiers ne sont pas structurés en lignes comme un CSV.
  - Utilisation de **spark.sparkContext.binaryFiles** pour lire les octets bruts.
- **Relation avec l'architecture** :
  - Les **DataNodes** stockent les fichiers images/PDF en blocs.
  - Les **Executors** traitent chaque fichier comme une Task indépendante.
  - Le **Cache** peut stocker des métadonnées extraites (ex : taille d'une image).

Exemple :

```
# Chaque image est une Task traitée sur un Executor

images_rdd = spark.sparkContext.binaryFiles("hdfs:///photos/*.jpg")

images_rdd.map(process_image).collect()  # "process_image" s'exécute sur les Workers
```

Points Clés à Retenir

1. **Abstraction** : Spark cache la complexité distribuée. Votre code reste le même, que les données soient sur HDFS ou en local.

2. **Parallélisme** : Plus il y a de Worker Nodes, plus les Tasks s'exécutent en parallèle (ex : 1000 images → 1000 Tasks).
3. **Optimisation** :

   - Le **Cache** évite de relire les données depuis HDFS.
   - Le **partitionnement** (ex : repartition()) impacte directement le nombre de Tasks.

Arrêter et Visualiser le Comportement d'une Session Spark

1. Arrêter une Session Spark

Lorsque vous avez terminé votre traitement, il est recommandé d'arrêter la session Spark pour libérer les ressources.

```
spark.stop()
```

- **Pourquoi ?** Cela permet d'éviter la consommation excessive de mémoire et de ressources CPU, surtout si vous exécutez plusieurs sessions Spark sur la même machine.

2. Vérifier l'État d'une Session Spark

Vous pouvez vérifier si la session Spark est active avec : `print(spark)`

Si la session est toujours active, vous verrez un objet du type :

```
<pyspark.sql.session.SparkSession object at 0x...>
```

Si elle est arrêtée, vous obtiendrez une erreur lorsque vous tenterez d'exécuter des commandes Spark.

3. Visualiser le Comportement de Spark (UI Web)

Spark propose une interface Web pour surveiller les tâches en cours d'exécution.

Accéder à l'interface Spark UI

1. **Démarrez votre application Spark**, puis ouvrez un navigateur et accédez à :
2. `http://localhost:4040`
3. **Ce que vous pouvez voir :**

   - **Jobs** : Liste des tâches en cours et terminées.
   - **Stages** : Détails des étapes de traitement.
   - **Storage** : Informations sur l'utilisation de la mémoire.

- **Executors** : Utilisation des ressources par chaque nœud.
- **SQL** : Explications des requêtes SQL exécutées.

Si Spark tourne sur un cluster, remplacez `localhost` par l'adresse de votre **master node**.

4. Afficher les Jobs en Cours via Python

Vous pouvez aussi voir l'état des tâches Spark en Python :

```
spark.sparkContext.statusTracker.getJobIdsForGroup()
```

Cela retourne une liste des ID des jobs en cours.

5. Vérifier l'Utilisation des Ressources

Pour voir comment Spark utilise la mémoire et le CPU sur votre machine :

- Sous **Linux/macOS**, utilisez :
- `top`
- `htop`
- Sous **Windows**, utilisez :
- `taskmgr  # (Ouvrir le Gestionnaire des tâches)`

6. Logger l'Activité de la Session

Vous pouvez activer les logs pour voir ce que fait Spark :

```
spark.sparkContext.setLogLevel("INFO")
```

Les niveaux possibles :

- `"ALL"` : Affiche tous les logs.
- `"DEBUG"` : Détails avancés.
- `"INFO"` : Infos générales (recommandé).
- `"WARN"` : Avertissements uniquement.
- `"ERROR"` : Erreurs seulement.
- `"OFF"` : Désactiver les logs.

Action	Commande
Arrêter Spark	`spark.stop()`
Vérifier si Spark est actif	`print(spark)`
Accéder à l'interface UI	`http://localhost:4040`
Voir les tâches en cours	`spark.sparkContext.statusTracker.getJobIdsForGroup()`
Activer les logs	`spark.sparkContext.setLogLevel("INFO")`

II. TP PySpark – Introduction au Traitement de Données Massives

Introduction

Apache Spark est un moteur de traitement distribué, conçu pour traiter rapidement de grandes quantités de données. Dans ce TP, nous allons manipuler un petit fichier CSV pour apprendre les bases du traitement de données avec **PySpark**.

Objectifs pédagogiques

À la fin de ce TP, vous serez capables de :

- Initialiser une session Spark.
- Lire et afficher un fichier CSV.
- Comprendre le schéma d'un DataFrame.
- Filtrer, grouper et agréger des données.
- Créer et appliquer une fonction personnalisée (UDF).
- Comprendre le partitionnement des données.

Étape 0 : Préparation des Données

Objectif : Comprendre la structure des données
Fichier : **students.csv**

```
id,name,age,grade
1,Alice,22,16
2,Bob,24,12
3,Charlie,23,14
```

Remarques :

- Format CSV = comme un tableau Excel, mais en texte.
- Chaque ligne représente un étudiant avec 4 champs : ID, nom, âge, note.

Pourquoi ? :
Spark a besoin de données structurées pour travailler efficacement. Le CSV est un format simple pour démarrer.

Étape 1 : Démarrage Spark et Lecture des Données

```
from pyspark.sql import SparkSession

# 1. Initialiser Spark
spark = SparkSession.builder \
    .appName("TP Spark Basics") \
    .config("spark.driver.memory", "2g") \
```

```
    .getOrCreate()
```

Explication :

- **SparkSession** : C'est le point d'entrée principal de **Spark** (comme le moteur d'une voiture)
- **.appName()** : Donne un nom à l'application (utile pour le débogage)
- **.config()** : Configure la mémoire allouée (2 Go ici)
- **getOrCreate()** : Lance la session Spark

Pourquoi ? :
Spark a besoin d'être "**démarré**" avant de travailler, comme un ordinateur qu'on allume.

```
# 2. Lire le CSV local
df = spark.read \
    .option("header", True) \
    .option("inferSchema", True) \
     .option("sep",";")  \
    .csv("students.csv")
```

Explication :

- **spark.read** : Méthode pour lire des données
- **.option("header", True)** : La première ligne contient les noms des colonnes
- **.option("inferSchema", True)** : Devine automatiquement les types de données (ex: age=nombre entier)
- **csv(...)** : Chemin du fichier (préfixe file:// pour les fichiers locaux)

Pourquoi ? :
Spark doit savoir comment interpréter le fichier. Sans **header=True**, il traiterait la première ligne comme des données.

```
# 3. Afficher le schéma
df.printSchema()
df.show()
```

Résultat :

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- grade: integer (nullable = true)
```

Explication :

- Affiche la "**carte d'identité**" des données
- **nullable = true** signifie que les valeurs peuvent être manquantes

Pourquoi ? :
Comprendre la structure des données est essentiel avant toute analyse.

Étape 2 : Manipulations de Données

Filtrer les étudiants ayant une note > 12

```
# 4. Filtrage des notes > 12
filtered_df = df.filter(col("grade") > 12)
```

Explication :

- **filter()** : Comme un tamis qui garde seulement certaines lignes
- **col("grade")** : Sélectionne la colonne "**grade**"
- **> 12** : Condition de filtrage

```
# 5. Calcul de la moyenne par âge
stats_df = df.groupBy("age") \
    .agg(avg("grade").alias("moyenne"))
```

Explication :

- **groupBy("age")** : Regroupe les étudiants par âge
- **agg(avg("grade"))** : Calcule la moyenne des notes pour chaque groupe

Pourquoi ? :
Montre comment agréger des données, une opération clé en analyse.

Étape 3 : Optimisation Spark

Repartitionner les données

```
# 6. Partitionnement des données
repartitioned_df = df.repartition(4)
```

Explication :

- Spark découpe les données en "**partitions**" (morceaux)
- **repartition(4)** : Divise les données en 4 morceaux
- Analogie : Comme diviser **un gros colis** en **petits paquets** pour le transporter plus vite

Pourquoi ? :
Plus de partitions = traitement parallèle plus efficace, mais trop de partitions = surcharge.

Exercices Guidés

1. Filtrage Complexe : Afficher les étudiants de plus de 20 ans et ayant une note > 13 :

```
df.filter((col("age") > 20) & (col("grade") > 13))
```

- **&** : Combine deux conditions (ET logique)

- Parenthèses indispensables !

2. Jointure : Jointure avec un autre DataFrame :

```
courses_df = spark.createDataFrame([(1, "Math"), (2, "Bio")], ["id", "cours"])
df.join(courses_df, "id", "inner")
```

- **inner** : Ne garde que les **ID** présents dans les deux tables

3. UDF : Catégorisation des notes (UDF)

```
from pyspark.sql.functions import udf

def classer_note(grade):
    return "A" if grade > 15 else "B" if grade > 12 else "C"

udf_classer = udf(classer_note, StringType())
df.withColumn("catégorie", udf_classer(col("grade")))
```

Conseils Pédagogiques

1. Utilisez l'interface web Spark (**http://localhost:4040**) pour montrer visuellement les jobs et les étapes
2. Comparez avec Excel :
   - **DataFrame** = feuille Excel
   - **filter()** = filtre automatique
   - **groupBy()** = tableau croisé dynamique
3. Faites modifier le CSV pour ajouter des erreurs (ex: texte dans la colonne "grade") et montrez l'importance du schéma

Pourquoi apprendre Spark ?

- **Big Data** : Spark gère des données trop grosses pour la mémoire d'un seul ordinateur.
- **Vitesse** : Le traitement parallèle est **100x** plus rapide qu'avec Excel/pandas pour de gros datasets.
- **Emploi** : 85% des entreprises utilisent **Spark** pour l'analyse de données.
- Très demandé en **Big Data**, **IA**, et **Data Engineering.**

III. Analyse de texte avec PySpark — Étude de cas NLP

Étape 1 : Création du Fichier Texte

Fichier : **discours.txt**

```
L'innovation technologique transforme notre quotidien. Les smartphones connectent le monde,
l'intelligence artificielle révolutionne la médecine, et les véhicules autonomes redéfinissent
```

les transports. Cependant, ces progrès soulèvent des questions éthiques. La protection des données personnelles devient cruciale. Les algorithmes doivent éviter les biais discriminatoires. L'éducation numérique est la clé pour préparer les générations futures. Ensemble, nous devons trouver un équilibre entre progrès technologique et valeurs humaines. La cybersécurité, l'énergie verte et l'accès équitable aux technologies sont les défis majeurs de cette décennie.

Ce qu'il faut apprendre

- Manipuler des fichiers texte avec **PySpark**.
- Nettoyer du texte (**tokenization**, **suppression de mots vides**).
- Utiliser **groupBy()** et **count()** pour compter les occurrences.
- Trier les données avec **orderBy()**.

Les Étapes

- On charge le texte.
- On vérifie s'il est vide.
- On l'affiche.
- On découpe le texte en mots (**Tokenizer**).
- On sépare chaque mot en une ligne (**explode**).
- On supprime les mots inutiles (**filter**).
- On compte la fréquence des mots (**groupBy.count()**).
- On trie par ordre décroissant (**orderBy(desc("count"))**).
- On affiche les **5 mots les plus fréquents**.
- On sauvegarde les résultats dans un fichier **CSV**.

Utiliser **PySpark** pour traiter **de grandes quantités de texte**.

1. Importation des bibliothèques

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import Tokenizer
from pyspark.sql.functions import explode, desc, col
```

- **SparkSession** : Permet de créer une session **Spark** pour traiter des données.
- **Tokenizer** : Un outil de **PySpark ML** pour découper du texte en mots.
- **explode** : Transforme une colonne contenant des listes en plusieurs lignes (chaque élément devient une ligne).
- **desc** : Trie les résultats en ordre décroissant (utile pour classer les mots les plus fréquents).
- **col** : Représente une colonne d'un DataFrame PySpark, utile pour les transformations.

2. Initialisation de Spark

```
spark = SparkSession.builder \
        .appName("Analyse de Texte") \
        .master("local[*]") \
        .getOrCreate()
```

- **SparkSession.builder** : Crée une session Spark.

- **.appName("Analyse de Texte")** : Donne un nom à l'application Spark (utile pour l'affichage dans l'interface Spark UI).
- **.master("local[*]")** : Exécute Spark en local en utilisant **tous les cœurs disponibles** (* signifie "**tous les cœurs**").
- **.getOrCreate()** : Crée la session si elle n'existe pas, sinon la récupère.

3. Lecture du fichier texte

```
text_df = spark.read.text("discours.txt")
```

- **spark.read.text("discours.txt")** : Charge un fichier texte sous forme d'un DataFrame PySpark.
- Chaque ligne du fichier devient une ligne dans le **DataFrame** avec une seule colonne appelée "**value**".

4. Vérification si le fichier est vide

```
if text_df.isEmpty():
    print("Le fichier discours.txt est vide ou introuvable.")
```

- **text_df.isEmpty()** : Vérifie si le **DataFrame** est vide.
- **print("Le fichier discours.txt est vide ou introuvable.")** : Affiche un message si le fichier est vide ou inexistant.

Remarque : isEmpty() n'existe pas en PySpark, il faut utiliser :

```
if text_df.count() == 0:
```

5. Affichage du texte brut

```
print("=== Texte original ===")
text_df.show(truncate=False)
```

- **.show(truncate=False)** : Affiche le contenu du **DataFrame** sans tronquer les lignes longues.

6. Tokenisation (découpage en mots)

```
tokenizer = Tokenizer(inputCol="value", outputCol="mots")
words_df = tokenizer.transform(text_df)
```

- **Tokenizer(inputCol="value", outputCol="mots")** :
  - inputCol="value" → Prend la colonne "value" contenant le texte.
  - outputCol="mots" → Crée une nouvelle colonne contenant une **liste de mots**.
- **.transform(text_df)** : Applique le découpage et retourne un DataFrame.

7. Séparation des mots (Explode)

```
words_sep = words_df.withColumn("mot", explode(col("mots")))
```

- **withColumn("mot", explode(col("mots"))) :**
  - explode(col("mots")) → Transforme la colonne "mots" (qui contient des **listes**) en plusieurs lignes **(chaque mot devient une ligne)**.
  - withColumn("mot", ...) → Crée une nouvelle colonne "mot".

Exemple :
Si "**mots**" contient ["**bonjour**", "**le**", "**monde**"], alors on obtient :

```
+--------+
| mot    |
+--------+
|bonjour |
|le      |
|monde   |
+--------+
```

8. Suppression des mots vides

```
mots_vides = ["les", "des", "et", "la", "de", "le"]
words_clean = words_sep.filter(~col("mot").isin(mots_vides))
```

- **Liste mots_vides** : Contient les mots inutiles à supprimer.
- **col("mot").isin(mots_vides)** : Vérifie si un mot est dans la liste.
- **~ (tilde)** : Inverse la condition (NOT en SQL), donc **on garde les mots qui ne sont pas dans mots_vides**.
- **.filter(...)** : Applique le filtre.

Exemple :
Si **words_sep** contient ["**bonjour**", "**le**", "**monde**", "**des**"], après filtrage il reste :

```
+--------+
| mot    |
+--------+
|bonjour |
|monde   |
+--------+
```

Les mots "**le**" et "**des**" ont été supprimés.

9. Comptage des occurrences des mots

```
word_count = words_clean.groupBy("mot").count().orderBy(desc("count"))
```

- **.groupBy("mot")** : Regroupe les mots identiques.
- **.count()** : Compte combien de fois chaque mot apparaît.
- **.orderBy(desc("count"))** : Trie par ordre décroissant (du mot le plus fréquent au moins fréquent).

Exemple :
Si **words_clean** contient :

```
+--------+
| mot    |
+--------+
| chat   |
| chien  |
| chat   |
| souris |
| chat   |
| chien  |
+--------+
```

Après **groupBy("mot").count()** :

```
+--------+-----+
| mot    |count|
+--------+-----+
| chat   |  3  |
| chien  |  2  |
| souris |  1  |
+--------+-----+
```

Puis après **.orderBy(desc("count"))**, le mot le plus fréquent est en haut.

10. Affichage des 5 mots les plus fréquents

```
print("=== Top 5 des mots clés ===")
word_count.show(5)
```

- **.show(5)** : Affiche les **5 mots les plus fréquents**.

11. Sauvegarde des résultats

```
word_count.write.mode("overwrite").csv("resultats_analyse")
```

- **.write.csv("resultats_analyse")** : Enregistre les résultats au format CSV.
- **.mode("overwrite")** : Remplace le fichier s'il existe déjà.

Résultats Pédagogiques

1. Compréhension des étapes clés du NLP avec Spark :
   - **Chargement → Nettoyage → Transformation → Analyse**
2. Visualisation des opérations :
   - Avant/après la tokenization
   - Impact du filtrage des mots vides
3. Applications réelles :
   - Analyse de sentiments
   - Détection de thèmes dominants

- Comparaison de textes

Exercice Pratique

1. Modifier le code pour trouver les mots les **moins** fréquents
2. Ajouter une colonne "longueur" avec le nombre de lettres par mot
3. Filtrer les mots de moins de 5 caractères

Solution partielle :

```
#Afficher les mots les moins fréquents
word_count.orderBy("count").show(5)
# Ajouter la longueur
from pyspark.sql.functions import length
words_clean = words_clean.withColumn("longueur", length("mot"))
# Filtrer
words_filtered = words_clean.filter(col("longueur") >= 5)
```

III. PySpark pour traiter des fichiers binaires (images)

Ce qu'il faut apprendre

- Utiliser PySpark pour traiter des fichiers binaires (**images**).
- Extraire et manipuler des métadonnées de fichiers.
- Filtrer et organiser des données efficacement avec **PySpark**.
- Générer des statistiques utiles avec **groupBy()** et **count()**.

Objectifs

1. On charge **les images en format binaire** avec PySpark.
2. On extrait **les métadonnées** (chemin, taille en Ko, date de modification).
3. On filtre **les images trop volumineuses** (> 500 Ko).
4. On affiche **les informations des images filtrées**.
5. On sauvegarde **les chemins des images analysées** en CSV.
6. On génère **des statistiques** sur la répartition des tailles d'images.

Étape 1 : Préparation des Données

Dossier : **images/** contenant 10 images JPG

```
chemin_absolu/
├── images/
│   ├── chat1.jpg
│   ├── chien2.jpg
│   └── ...
```

1. Importation des bibliothèques

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import input_file_name, length, col
```

- **SparkSession** : Permet de créer une session Spark.
- **input_file_name** : Récupère le **nom et chemin** des fichiers analysés.
- **length** : Permet de calculer la taille des fichiers.
- **col** : Représente une colonne d'un DataFrame, utile pour les transformations.

2. Initialisation de Spark

```
spark = SparkSession.builder \
    .appName("Analyse d'Images") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()
```

- **.appName("Analyse d'Images")** : Donne un nom à l'application Spark.
- **.config("spark.driver.memory", "4g")** : Alloue **4 Go** de mémoire au driver Spark.
- **.getOrCreate()** : Crée la session si elle n'existe pas déjà.

3. Lecture des images en format binaire

```
images_df = spark.read.format("binaryFile") \
    .option("pathGlobFilter", "*.jpg") \
    .load("images/")
```

- **format("binaryFile")** : Permet de lire les fichiers binaires (images, vidéos, etc.).
- **.option("pathGlobFilter", "*.jpg")** : Filtre uniquement les **fichiers .jpg**.
- **.load("images/")** : Charge tous les fichiers images du répertoire.

Structure du DataFrame après chargement

```
root
 |-- path: string (nullable = true)            # Chemin du fichier
 |-- modificationTime: timestamp (nullable = true)  # Date de modification
 |-- length: long (nullable = true)           # Taille du fichier en octets
 |-- content: binary (nullable = true)      # Contenu de l'image en binaire
```

4. Extraction des métadonnées

```
images_metadata = images_df.select(
    input_file_name().alias("chemin"),
    (col("length") / 1024).alias("taille_ko"),  # Conversion en Ko
    "modificationTime"
)
```

- **input_file_name().alias("chemin")** : Récupère le **chemin du fichier**.

- **col("length") / 1024** : Convertit la taille du fichier de **bytes** en **Ko**.
- **alias("taille_ko")** : Renomme la colonne.
- **"modificationTime"** : Récupère la **date de modification** de l'image.

5. Filtrage des images trop lourdes

```
images_filtrees = images_metadata.filter(col("taille_ko") < 500)
```

- **filter(col("taille_ko") < 500)** : Ne garde que les images **de moins de 500 Ko**.

6. Affichage des résultats

```
print("=== Métadonnées des images filtrées ===")
images_filtrees.show(5, truncate=False)
```

- **.show(5, truncate=False)** : Affiche **5 images**, sans tronquer les chemins trop longs.

7. Sauvegarde des chemins des images filtrées

```
images_filtrees.select("chemin") \
   .write.mode("overwrite") \
   .csv("images_analysées")
```

- **.select("chemin")** : Sélectionne uniquement la colonne contenant les **chemins des fichiers**.
- **.write.mode("overwrite")** : Remplace le fichier s'il existe déjà.
- **.csv("images_analysées")** : Sauvegarde les résultats au format **CSV**.

8. Statistiques sur la répartition des tailles d'images

```
stats_images = images_metadata.groupBy("taille_ko") \
   .count() \
   .orderBy(col("count"), ascending=False)
```

- **.groupBy("taille_ko")** : Regroupe les images **par taille**.
- **.count()** : Compte le **nombre d'images** pour chaque taille.
- **.orderBy(col("count"), ascending=False)** : Trie du **plus fréquent au moins fréquent**.

Affichage des statistiques

```
stats_images.show()
```

- Affiche la **répartition des images en fonction de leur taille**.

III. PySpark pour traiter des fichiers binaires (Vidéos)

Pourquoi utiliser Spark pour des Vidéos ?

- **Batch processing** : Traiter des milliers de vidéos en parallèle
- **Intégration ML** : Ajouter de la détection d'objets avec **MLlib**
- **Scalabilité** : Passer d'une vidéo à 10 000 vidéos sans changer le code

Conseils Pédagogiques

- ✓ Installer **OpenCV** avec pip install opencv-python-headless
- ✓ Montrer la différence entre :
  - **Traitement brut** (données binaires)
  - **Traitement de métadonnées**
  - **Traitement de contenu** (requiert des librairies spécialisées)
- ✓ Utiliser une vidéo de 10 secondes pour des tests rapides.

Étape 1 : Préparation des Données

Fichier : **video.mp4**

Étape 2 :

1. Importation des bibliothèques

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import input_file_name, udf
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, FloatType
import cv2 #pip install openCv-python
import numpy as np
```

- **`SparkSession`** : Permet de créer une session Spark pour exécuter des traitements distribués.
- **`input_file_name`** : Récupère le chemin du fichier.
- **`udf`** : Définit une **User Defined Function** (UDF) pour transformer les données.
- **`StructType`** et **`StructField`** : Permet de définir un schéma pour les données.
- **`cv2`** **(OpenCV)** : Bibliothèque utilisée pour analyser les vidéos.
- **`numpy`** : Utilisé pour convertir les données binaires de la vidéo en tableau.

2. Initialisation de Spark

```
spark = SparkSession.builder \
    .appName("Analyse Vidéo") \
    .config("spark.driver.memory", "8g") \
    .getOrCreate()
```

- **Création d'une session Spark** nommée **`"Analyse Vidéo"`**.
- **Allocation de 8 Go de mémoire** pour améliorer les performances.

3. Lecture du fichier vidéo en binaire

```
video_df = spark.read.format("binaryFile") \
    .load("video.mp4")
```

- **Lecture d'un fichier vidéo au format binaire**.
- `binaryFile` permet de charger des fichiers non textuels.

4. Définition de l'UDF pour extraire les métadonnées

```
# Fonction pour extraire les métadonnées vidéo
def get_video_metadata(path):
        cap = cv2.VideoCapture(path)

        if not cap.isOpened():
           return ("Erreur", 0, 0, 0, 0.0)

        fps = cap.get(cv2.CAP_PROP_FPS)
        total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
        width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        duration = total_frames / fps if fps > 0 else 0.0

        cap.release()

        return ("OK", total_frames, width, height, round(duration, 2))

# Lire les métadonnées localement
status, total_frames, width, height, duration = get_video_metadata("video.mp4")

# Créer un DataFrame Spark avec les résultats
data = [("video.mp4", status, total_frames, width, height, duration)]
```

```
def get_video_metadata(path):
```

- **Définition de fonction** :

On crée une fonction appelée `get_video_metadata` qui prend un **chemin de vidéo** (ex: `"video.mp4"`) comme paramètre.

```
    cap = cv2.VideoCapture(path)
```

- **Chargement de la vidéo :**

On utilise **OpenCV** (`cv2`) pour ouvrir le fichier vidéo à l'aide de la classe `VideoCapture`.

- `cap` devient un objet qui permet d'accéder aux images (frames) de la vidéo.

```
    if not cap.isOpened():
```

```
        return ("Erreur", 0, 0, 0, 0.0)
```

- **Vérification d'erreur** :

Si la vidéo ne peut pas être ouverte (fichier non trouvé ou corrompu), on retourne un **statut d'erreur** et des valeurs nulles.

```
fps = cap.get(cv2.CAP_PROP_FPS)
```

- **FPS (Images par seconde)** :

On récupère le **nombre d'images par seconde** (frames per second) de la vidéo.

```
total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
```

- **Nombre total d'images** :

On récupère le **nombre total d'images** (ou frames) dans la vidéo.

```
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
```

- **Taille de la vidéo** :

On récupère les dimensions de la vidéo en pixels :

- `width` : la largeur
- `height` : la hauteur

```
duration = total_frames / fps if fps > 0 else 0.0
```

- **Durée de la vidéo (en secondes)** :

On calcule la **durée** de la vidéo avec la formule :
`nombre total d'images / FPS`
(On évite la division par zéro en vérifiant que `fps > 0`.)

```
cap.release()
```

- **Libération des ressources :**

On ferme le fichier vidéo pour libérer la mémoire.

```
return ("OK", total_frames, width, height, round(duration, 2))
```

- **Retour des métadonnées :**

On retourne un **tuple** contenant :

- `"OK"` pour dire que tout s'est bien passé

- `total_frames`
- `width, height`
- `duration` (arrondie à 2 décimales)

Partie suivante : Appel et usage

```
status, total_frames, width, height, duration =
get_video_metadata("video.mp4")
```

Appel de la fonction :

On appelle la fonction avec le fichier `"video.mp4"`
Les informations retournées sont stockées dans des variables.

```
data = [("video.mp4", status, total_frames, width, height, duration)]
```

5. Définition du schéma pour structuré pour stocker les métadonnées

```
schema = StructType([
    StructField("fps", StringType(), True),
    StructField("total_frames", IntegerType(), True),
    StructField("width", IntegerType(), True),
    StructField("height", IntegerType(), True),
    StructField("duration_sec", FloatType(), True)
])
```

StringType(), IntegerType(), FloatType() : Types de données.
True signifie que les valeurs peuvent être nulles.

6. Application de l'UDF à la colonne `content`

```
metadata_udf = udf(get_video_metadata, schema)
video_analysis_df = video_df.withColumn("metadata", metadata_udf("content"))
```

On **convertit la fonction Python en une UDF (User-Defined Function)** utilisable avec PySpark.
`.withColumn("metadata", metadata_udf("content"))` :

- ✓ Applique la fonction à chaque ligne du DataFrame.
- ✓ Stocke le résultat dans une nouvelle colonne `metadata`.

7. Séparation des colonnes

```
results_df = video_analysis_df.select(
    input_file_name().alias("chemin"),"metadata.*"
)
```

Récupération du chemin du fichier.
Extraction des colonnes de `metadata`.
input_file_name() : Ajoute une colonne avec le chemin du fichier vidéo.
"**metadata.***" : Explose la colonne **metadata** pour récupérer **fps**, **width**, etc.

8. Affichage des résultats

```
print("=== Métadonnées Vidéo ===")
results_df.show(truncate=False)
```

- **Affichage des métadonnées vidéo** sans troncature.

```
=== Métadonnées Vidéo ===
+-------------------------+-----+-------------+-----+------+------------+
|chemin                   |fps  |total_frames |width|height|duration_sec|
+-------------------------+-----+-------------+-----+------+------------+
|video.mp4                |30.0 |4500         |1280 |720   |150.0       |
+-------------------------+-----+-------------+-----+------+------------+
```

9. Sauvegarde des résultats

```
results_df.write.mode("overwrite").csv("resultats_video")
```

- **Sauvegarde des résultats dans un fichier CSV**.
- **Mode `overwrite`** : Remplace les fichiers existants.

Erreurs Courantes et Solutions

Problème	Solution
cv2.error	Vérifier l'installation d'OpenCV
Mémoire insuffisante	Augmenter spark.driver.memory
Format non supporté	Convertir en MP4 avec FFmpeg

IV – Traitement massif de fichiers PDF avec Apache Spark

Objectif final **:** Extraire du texte propre des **PDFs** → **nettoyer** → **préparer un dataset pour entraîner une IA** de modification/synthèse de documents

Dans ce TP, vous êtes des **ingénieurs Big Data** chargés de **préparer** une grande collection de fichiers PDF pour l'apprentissage d'un modèle d'intelligence artificielle.
Le but est d'**extraire**, **nettoyer**, **structurer** les contenus texte des PDFs en utilisant **Apache Spark**.

Compétences cibles :

- Traitement distribué avec Apache Spark
- Extraction et nettoyage de texte non structuré
- Optimisation de stockage pour le Machine Learning

Matériel nécessaire

- Librairies :
  - `pyspark`
  - `pdfplumber` ou `PyMuPDF` (`fitz`) pour lire les PDFs
  - `nltk` (nettoyage linguistique)
  - `spacy` pour tokenization avancée

```
pip install  pdfplumber
pip install  nltk
```

Partie 1 – Chargement massif de PDFs avec Spark

1. **Structure des données** :
   Les fichiers PDF sont stockés dans un dossier `/data/pdfs/`.
2. **Objectif** :
   Lire **tous les fichiers PDF** en **parallèle** dans un **DataFrame** Spark.

Partie 2 – Extraction du texte

1. **Lire chaque PDF** :
   - Utiliser **`pdfplumber`** ou **`fitz`** pour extraire tout le texte brut.
2. **Nettoyage de base** :
   - Supprimer les caractères spéciaux, multiples espaces, sauts de lignes inutiles.
   - Ne garder que du texte exploitable.

Partie 3 – Structuration

1. **Structurer le contenu** :
   - **Colonnes** : `filename`, `raw_text`, `clean_text`, `length`, `nb_pages`
   - Calculer automatiquement le nombre de pages et la taille du texte.
2. **Sauvegarde** :
   - Sauvegarder le DataFrame en **Parquet** ou en **Delta Lake** pour traitement ultérieur.

Partie 4

1. **Segmentation en paragraphes** : Split le texte en paragraphes ou sections.
2. **Filtrage** : Supprimer les documents trop courts (< 500 caractères).
3. **Indexation** : Créer une table d'index pour retrouver rapidement les documents.

Partie 1 – Chargement des PDFs

Objectif : Lire efficacement des milliers de fichiers PDF en parallèle.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, IntegerType
from pyspark.sql.functions import regexp_extract, lit
import pdfplumber
import os
import re
# 1. Initialiser une session Spark
spark = SparkSession.builder \
    .appName("PDF Processing for AI Training") \
    .getOrCreate()
```

- Import des librairies.
  - **pyspark** : Core Spark pour le traitement distribué
  - **pdfplumber** : Extraction texte depuis PDFs
  - **re** : Expressions régulières pour le nettoyage

- Création d'une session Spark.
  - **.appName()** : Identifie le job dans l'UI Spark
  - **.getOrCreate()** : Réutilise une session existante ou en crée une nouvelle.

```
# 2. Définir le dossier source
pdf_folder = "data/pdfs/"

# 3. Lister tous les fichiers PDF
pdf_files = [os.path.join(pdf_folder, f) for f in os.listdir(pdf_folder) if f.endswith('.pdf')]

# 4. Créer un DataFrame Spark avec les chemins
pdf_df = spark.createDataFrame(pdf_files, StringType()).toDF("filepath")
```

 # 3 : Cette ligne utilise une **compréhension de liste** pour :

- Parcourir tous les fichiers du dossier `pdf_folder` avec `os.listdir(pdf_folder)`.
- Garder uniquement ceux qui se terminent par `.pdf` grâce à `f.endswith('.pdf')`.
- Construire le **chemin complet** du fichier PDF avec `os.path.join(pdf_folder, f)`.

Résultat : une liste `pdf_files` contenant les **chemins complets** de tous les fichiers PDF dans le dossier.

`# 4 :` Cette ligne crée un **DataFrame Spark** à partir de la liste `pdf_files` :

- `spark.createDataFrame(pdf_files, StringType())` crée un DataFrame avec **une seule colonne** contenant des chaînes (`StringType`).
- `.toDF("filepath")` renomme cette colonne en `"filepath"`.

Résultat : un DataFrame Spark `pdf_df` contenant un champ `filepath` pour chaque fichier PDF trouvé dans le dossier.

Partie 2 – Extraction et Nettoyage

Objectif : Extraire le texte brut et appliquer des transformations de base.

```
# 5. Définir une UDF (User Defined Function)
def extract_clean_text(path):
    try:
        with pdfplumber.open(path) as pdf:
            text = ''
            for page in pdf.pages:
                text += page.extract_text() + '\n'  # Extraction par page
        text = re.sub(r'\s+', ' ', text)  # Remplacer les espaces multiples
        return text.strip()  # Supprimer les espaces en début/fin
    except:
        return ""  # Gestion d'erreurs

extract_udf = udf(extract_clean_text, StringType())
```

[1]Déclaration d'une fonction Python nommée extract_clean_text qui prend en paramètre path, c'est-à-dire le **chemin d'un fichier PDF**.

[2]Démarrage d'un bloc try/except pour capturer les erreurs potentielles lors de la lecture du fichier (ex. fichier manquant ou mal formé).

[3]Utilise la bibliothèque pdfplumber pour **ouvrir le fichier PDF** spécifié par le chemin path.

[4]Cela crée un objet pdf contenant toutes les pages du fichier.

[5]Initialise une variable text vide (en réalité avec un espace) qui servira à stocker **tout le texte extrait** du PDF.

[6]Pour chaque page du PDF :

- `page.extract_text()` extrait le **texte brut** de cette page.
- Ce texte est ajouté à la variable `text`, suivi d'un saut de ligne.

[7]Utilise la bibliothèque `re` (expressions régulières) pour :

- Remplacer tous les **espaces multiples** (y compris tabulations, retours à la ligne) par **un seul espace**.

[8]Supprime les espaces en début et fin de texte et retourne le résultat.

[9]Si une **erreur se produit** à n'importe quelle étape (ex. PDF vide, corrompu, etc.), la fonction retourne une **chaîne vide**.

[10]Cette ligne convertit la fonction extract_clean_text en une **UDF utilisable dans Spark SQL**, avec un **type de retour StringType (chaîne)**.

```
# 6. Appliquer l'UDF pour créer la colonne 'clean_text'
pdf_df = pdf_df.withColumn("clean_text", extract_udf(col("filepath")))
```

Partie 3 – Structuration

Objectif : Ajouter des métadonnées et optimiser le stockage.

```
# 7. Ajouter des colonnes métadonnées
from pyspark.sql.functions import length
[1]pdf_df = pdf_df.withColumn("filename", regexp_extract(col("filepath"), "[^/]+$", 0)) \
        [2].withColumn("length", length(col("clean_text"))) \
        [3].withColumn("nb_pages", lit(0))  # À compléter avec une UDF si nécessaire
# 8. Afficher un échantillon
[4]pdf_df.show(truncate=False)
```

[1]Cette ligne ajoute une nouvelle colonne `"filename"` contenant uniquement le **nom du fichier PDF** (sans le chemin complet).

- `regexp_extract(col("filepath"), "[^/]+$", 0)` : extrait tout ce qui vient **après le dernier `/`** dans le chemin (`filepath`), soit le nom du fichier (`[^/]+$` = "dernière séquence sans slash").

[2]Ajoute une colonne `"length"` qui contient la **longueur du texte** extrait (`clean_text`) pour chaque fichier PDF.

- Cela donne une idée du **volume de texte** contenu dans chaque document.

[3]Ajoute une colonne `"nb_pages"` initialisée à **0** pour tous les fichiers.

- `lit(0)` crée une valeur constante 0.
- *Remarque* : cette colonne est là **en prévision d'un traitement** futur par exemple, si tu veux utiliser une **UDF pour compter les pages du PDF**.

[4]Affiche le contenu du DataFrame `pdf_df` **sans tronquer les chaînes** :

- `truncate=False` permet de voir **tout le contenu des colonnes**, même si elles sont longues (utile pour vérifier le texte extrait).

```
# 9. Sauvegarder en Parquet
pdf_df.write.mode("overwrite").parquet("output/pdf_cleaned.parquet")
```

- **Parquet** : Format columnaire optimisé pour les requêtes Spark/ML.

Partie 4 – Bonus

Le filtrage :

```
from pyspark.sql.functions import length
# Filtrer les documents trop courts
pdf_df_filtered = pdf_df.filter(length(col("clean_text")) >= 500)
```

Résultats

filename	raw_text	clean_text	length	nb_pages
doc1.pdf	"..."	"Lorem ipsum"	1500	10

- **Parquet** : Un dossier contenant des fichiers `.parquet` partitionnés.
- **Recommandations** :
  - Tester avec un sous-ensemble de PDFs avant le full dataset.
  - Monitorer l'UI Spark pour détecter les skews de données.

V – Apprentissage Word2Vec sur les textes nettoyés

Objectif

À partir des **PDF nettoyés** dans la **Partie 1**, entraîner un **modèle Word2Vec** distribué en **Spark MLlib** pour obtenir des **embeddings** de mots.

Ces vecteurs de mots pourront ensuite être utilisés dans un modèle IA de **génération** ou **modification** de documents.

→ Objectif Global

Entraîner un modèle d'embeddings de mots (**Word2Vec**) distribué sur les textes nettoyés, pour capturer les relations sémantiques entre les mots.

→ Compétences cibles :

- Manipulation de modèles de NLP distribué avec Spark MLlib
- Transformation de texte en représentations vectorielles
- Optimisation des hyperparamètres pour l'apprentissage non supervisé

→ Diagramme du Pipeline

[PDF nettoyés] --> [Tokenization] --> [Word2Vec] --> [Embeddings] --> [IA de génération]

→ Objectifs Pédagogiques

1. **Spark MLlib** : Comprendre l'entraînement distribué de modèles NLP.
2. **Embeddings** : Saisir l'utilité des vecteurs de mots pour l'IA générative.
3. **Optimisation** : Expérimenter avec les hyperparamètres (vectorSize, minCount).
4. **Pipeline IA** : Préparer des données pour des tâches avancées (ex: fine-tuning de GPT).

Partie 1 – Chargement des Données Nettoyées

Objectif : Récupérer les textes prétraités depuis le stockage Parquet.

```
# 1. Lire le fichier Parquet généré précédemment
pdf_df = spark.read.parquet("output/pdf_cleaned.parquet")
```

- Utilise **spark.read.parquet()** pour charger les données nettoyées en parallèle.
- **Pourquoi Parquet** ? Format compressé et optimisé pour les requêtes Spark.

Partie 2 – Tokenization des Textes

Objectif : Convertir le texte en listes de mots (**tokens**) exploitables par **Word2Vec**.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, StringType
import re
# 2. Définir une UDF de tokenization basique
def simple_tokenizer(text):
    tokens = re.findall(r'\b\w+\b', text.lower())
    return tokens
tokenizer_udf = udf(simple_tokenizer, ArrayType(StringType()))
```

- [1] : re.findall(r'\b\w+\b', ...) détecte les mots (ex: "data-science" → ["**data**", "**science**"]).
- [1] : text.lower() Conversion en minuscules pour homogénéiser les tokens (éviter "**Data**" vs "**data**").

```
# 3. Appliquer la tokenization
pdf_df = pdf_df.withColumn("tokens", tokenizer_udf(pdf_df.clean_text))
```

- **Résultat** : Une colonne tokens de type Array<String> est ajoutée au DataFrame.
- **Exemple** : "Hello AI world!" → ["hello", "ai", "world"].

Partie 3 – Entraînement du Modèle Word2Vec

Objectif : Apprendre des e mbeddings de mots en contexte distribué.

```
from pyspark.ml.feature import Word2Vec
# 4. Configurer le modèle
word2Vec = Word2Vec(
    vectorSize=100,
    minCount=5,
    windowSize=5,
    inputCol="tokens",
    outputCol="result"
)
# 5. Entraîner le modèle sur tous les documents
model = word2Vec.fit(pdf_df)
```

- **Paramètres clés** :
  - vectorSize=100 : Chaque mot sera représenté par un vecteur de **100 dimensions**. Plus la taille est grande, plus la représentation peut être riche, mais cela coûte plus de ressources.
  - minCount=5 : Seuls les mots qui apparaissent **au moins 5 fois** dans le corpus seront pris en compte. Cela permet d'éliminer les mots rares ou les fautes de frappe qui n'apportent pas de valeur.
  - windowSize=5 : Lors de l'apprentissage, le modèle regarde un **contexte de 5 mots avant et après** chaque mot pour apprendre les relations de voisinage sémantique.
  - **inputCol**="tokens" : Indique la **colonne d'entrée** contenant les textes tokenisés (souvent une liste de mots). Dans ce cas, elle s'appelle "tokens".

- **outputCol**="result" : Indique le nom de la **colonne de sortie**, où seront stockés les vecteurs représentant les mots ou documents. Ici, "result" contiendra les représentations vectorielles calculées.

```
# 6. Explorer les embeddings
result = model.getVectors()  # DataFrame des mots et leurs vecteurs
result.show(5, truncate=False)
```

- **Sortie** :

```
+--------------------+------------------------------------------------------------+
|word                |                vector                                      |
+--------------------+------------------------------------------------------------+
|intelligence        |       [0.12, -0.45, 0.76, ..., 0.98]                       |
|data                |       [-0.32, 0.21, 0.09, ..., -0.17]                      |
+--------------------+------------------------------------------------------------+
```

Partie 4 – Sauvegarde et Utilisation des Résultats

Objectif : Persister le modèle et exploiter les embeddings.

```
# 7. Sauvegarder le modèle entier
model.save("save/word2vec_model")
# 8. Sauvegarder les embeddings en Parquet
result.write.mode("overwrite").parquet("embeddings.parquet")
```

- **Pourquoi 2 formats** ?
  - ✓ `model.save()` : Permet de réutiliser l'objet Word2Vec (ex: `findSynonyms`).
  - ✓ `Parquet` : Optimisé pour l'intégration avec d'autres outils (ex: TensorFlow, PyTorch).

```
# 9. Trouver des mots similaires
synonyms = model.findSynonyms("intelligence", 5)  # Top 5 mots proches
synonyms.show()
```

- **Exemple de sortie** :

```
+------------+------------------+
|word        | similarity       |
+------------+------------------+
|ai          |    0.92          |
|algorithm   |   0.85           |
|learning    |   0.83           |
+------------+------------------+
```

1. **Améliorer la tokenization** avec NLTK/Spacy :

```
[1]from nltk.corpus import stopwords
[2]stop_words = set(stopwords.words("english"))

[3]def advanced_tokenizer(text):
    [4]tokens = re.findall(r'\b\w+\b', text.lower())
    [5]tokens = [t for t in tokens if t not in stop_words and len(t) > 2]  # Filtrage
    [6]return tokens
```

- [1]Importe le module **stopwords** depuis le corpus **NLTK**. Il contient des listes de mots fréquents et peu informatifs (comme "**the**", "**is**", "**and**") dans différentes langues.
- [2]Charge la liste des **stopwords** en anglais depuis **NLTK**, puis la transforme en **ensemble (set)** pour un accès plus rapide (recherche plus rapide que dans une liste).
- [3]Déclaration d'une **fonction personnalisée** nommée advanced_tokenizer qui prend une chaîne de caractères text en entrée.
- [4]Utilise une expression régulière pour extraire tous les mots du texte, après l'avoir converti en minuscules.
  - `\b\w+\b` signifie : tous les mots composés de caractères alphanumériques délimités par des bordures de mots.
  - `text.lower()` permet d'uniformiser en minuscules pour faciliter le filtrage.
- [5]Filtre les tokens pour :
  - Supprimer les mots qui sont dans `stop_words` (les mots inutiles comme "the", "in", "and", etc.).
  - Supprimer les mots ayant une longueur inférieure ou égale à 2 (ex. : "**to**", "**at**", etc.).
- [6]Renvoie la liste de tokens **nettoyés** et **filtrés**, prêts pour une analyse ou une vectorisation (par ex. Word2Vec, TF-IDF, etc.).

Exemple d'utilisation :

text = "This is an example of a simple text preprocessing step in NLP."

print(advanced_tokenizer(text))

résultat : ['example', 'simple', 'text', 'preprocessing', 'step', 'nlp']

IV – Préparer un dataset pour Fine-tuning LLM

Objectif Global

Transformer les textes nettoyés en un dataset structuré au format **JSONL** pour le fine-tuning de modèles de langage (ex: LLaMA, Mistral).

Compétences cibles :

- ✓ Compréhension du format JSONL pour l'entraînement NLP
- ✓ Création de paires "prompt/completion" pour l'apprentissage supervisé
- ✓ Manipulation avancée de DataFrames Spark

Objectifs Pédagogiques

1. **Format JSONL** : Comprendre son rôle dans l'entraînement des LLMs.
2. **Prompt Engineering** : Apprendre à structurer des instructions pour guider un modèle.
3. **Intégration Spark/IA** : Préparer des données **à grande échelle** pour des frameworks comme **Hugging Face**.
4. **Optimisation** : Gérer les limites de taille de contexte des LLMs (ex: 4096 tokens pour GPT-4).

Partie 1 – Chargement des Données Nettoyées

Objectif : Récupérer les textes prêts pour la génération de prompts.

```
# 1. Charger les données depuis Parquet
pdf_df = spark.read.parquet("output/pdf_cleaned.parquet")
```

- **Astuce** : Vérifier la présence de la colonne `clean_text` avec `.printSchema()`.

Partie 2 – Création des Paires Prompt/Completion

Objectif : Structurer les textes en entrées/sorties pour l'entraînement.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
# 2. Définir une UDF pour générer des prompts
def create_prompt(text):
    return f"Document context:\n{text}\nInstruction: Résumez ce document en 5 lignes."
```

```
prompt_udf = udf(create_prompt, StringType())

# 3. Ajouter les colonnes prompt/completion

pdf_df = pdf_df.withColumn("prompt", prompt_udf(col("clean_text"))) \
               .withColumn("completion", lit(""))  # Champ vide à remplir ultérieurement
```

[1]Définit une fonction Python appelée `create_prompt` :

- Elle prend un texte `text` en entrée.
- Elle retourne une chaîne formatée qui suit une structure type **prompt** pour un modèle de langage (ex. GPT) :

[2]Étapes :

1. `withColumn("prompt", ...)` : ajoute une nouvelle colonne `prompt` au DataFrame `pdf_df`, contenant le résultat de l'UDF appliquée à la colonne `"clean_text"`.
2. `withColumn("completion", lit(""))` : ajoute une colonne `"completion"` contenant pour l'instant des chaînes vides (`""`). Cette colonne est souvent utilisée lors de l'entraînement de modèles (prompt → completion), où la réponse n'est pas encore fournie.

Exemple concret :

Si clean_text =

"Artificial Intelligence is transforming the world in many ways, including healthcare and education."

Alors prompt sera :

Document context:
Artificial Intelligence is transforming the world in many ways, including healthcare and education.
Instruction: Résumez ce document en 5 lignes.

- completion : Initialisé vide pour labellisation manuelle/automatique future.
- **Exemple de résultat** :

```
{"prompt": "Document context:\nLe NLP...\nInstruction: Résumez...", "completion": ""}
```

Partie 3 – Sauvegarde en Format JSONL

Objectif : Exporter le dataset dans un format compatible avec les librairies de fine-tuning.

```
# 4. Sélectionner les colonnes pertinentes
final_df = pdf_df.select("prompt", "completion")
# 5. Sauvegarder en JSON (un fichier par partition Spark)
final_df.write.mode("overwrite").json("output/fine_tuning_dataset_json")
```

- **Problème** : Spark génère des fichiers multiples (part-*.json).
- **Solution** : Fusionner les fichiers en un seul JSONL via terminal :

```
cat output/fine_tuning_dataset_json/*.json > output/final_dataset.jsonl
```

Résultat Final – Exemple de JSONL

```
{"prompt": "Document context:\nL'IA transforme...\nInstruction: Résumez...", "completion": ""}
{"prompt": "Document context:\nLe deep learning...\nInstruction: Résumez...", "completion": ""}
```

Bonus – Améliorations

1. **Auto-génération des réponses** (avec GPT-4 ou Mistral) :

   Un **exemple** de génération automatique de réponses (completion) pour chaque ligne d'un DataFrame contenant des prompts.

```
[1]from openai import OpenAI
[2]client = OpenAI()
[3]def generate_completion(prompt):
    [4]response = client.chat.completions.create(
        [5]model="gpt-4",
        [6]messages=[{"role": "user", "content": prompt}]
    )
    [7]return response.choices[0].message.content
# UDF Spark (à utiliser avec prudence pour éviter les coûts)
completion_udf = udf(generate_completion, StringType())
pdf_df = pdf_df.withColumn("completion", completion_udf(col("prompt")))
```

   [1]On importe la bibliothèque openai (ici avec une syntaxe adaptée à une version récente du SDK openai).

   - [2]client = OpenAI() crée un **client de connexion** à l'API d'OpenAI (il utilise ta clé API en interne, qu'il faut configurer).

   [3]On crée une fonction Python generate_completion(prompt) qui :

   1. Envoie le prompt à l'API d'OpenAI (ici gpt-4) en mode chat.
   2. Le message est envoyé comme si un utilisateur disait : *"Voici le prompt"*.
   3. L'API retourne une réponse dans response.choices[0].message.content.
   4. Cette réponse (le résumé, par exemple) est **renvoyée** par la fonction.

   On transforme generate_completion en **UDF (fonction utilisable sur un DataFrame Spark)** :

- Le type de retour est StringType() (une chaîne de caractères).
  Attention : chaque appel de cette fonction contactera l'API OpenAI (donc coûte du temps + de l'argent !). Si tu as 1000 lignes → 1000 requêtes API.

Pour chaque ligne du DataFrame pdf_df :

- On applique completion_udf sur la colonne "prompt" ;
- Le résultat est stocké dans la colonne "completion" (résumé ou réponse générée par l'IA).

Conseils importants :

- Ce code **fonctionne** mais il est **très lent et coûteux** pour de grands volumes.
- Il vaut mieux **générer les completions par lots en Python, en dehors de Spark**, puis les réimporter dans Spark.

2. **Découpage des textes longs** :

Ce code vise à **découper les textes longs** (comme ceux extraits de PDF) en **morceaux de 500 mots maximum**, pour faciliter le traitement par des modèles comme GPT, qui ont une limite de tokens.

```
from pyspark.sql.functions import split, explode
# Découper le texte en chunks de 500 mots
[1]pdf_df = pdf_df.withColumn("chunks", split(col("clean_text"), " ", 500)) \
            .withColumn("chunk", explode(col("chunks")))
```

[1]**Tentative** de création d'une colonne "chunks" contenant des morceaux de 500 mots, en utilisant :

- split(col("clean_text"), " ", 500) :
  - Cela utilise la fonction split() de Spark.
  - Elle découpe le texte en utilisant les **espaces** comme séparateur.
  - Le 3e argument 500 **ne découpe pas tous les 500 mots**, **il limite** le **nombre de morceaux** à 500 (ce n'est pas ce qu'on veut).

Problème : split(..., 500) ne découpe pas *tous les 500 mots*, mais **limite le nombre total de morceaux à 500** → donc **cette ligne ne fonctionne pas pour des vrais chunks de 500 mots**.

On ajoute une nouvelle colonne `"chunk"` :

- Elle est créée avec `explode(col("chunks"))` :
  - `explode()` transforme chaque élément de la liste `"chunks"` en une **ligne séparée** dans le DataFrame.
  - Cela permet de traiter chaque **morceau de texte séparément** dans les prochaines étapes (comme la génération de résumé).
- **Ce que tu veux probablement faire**

Si tu veux **vraiment découper le texte en blocs de 500 mots**, il faut d'abord écrire une fonction personnalisée comme ceci :

```
from pyspark.sql.functions import udf

from pyspark.sql.types import ArrayType, StringType

def split_into_chunks(text, chunk_size=500):

    words = text.split()

    return [' '.join(words[i:i+chunk_size]) for i in range(0, len(words), chunk_size)]

split_udf = udf(split_into_chunks, ArrayType(StringType()))

pdf_df = pdf_df.withColumn("chunks", split_udf(col("clean_text")))

pdf_df = pdf_df.withColumn("chunk", explode(col("chunks")))
```

- `split_into_chunks` : une fonction qui divise un texte en morceaux de 500 mots.
- `split_udf` : une UDF Spark pour appliquer cette logique à chaque ligne du DataFrame.
- `withColumn("chunks", ...)` : crée une colonne avec la liste des chunks.
- `explode` : transforme chaque chunk en une ligne séparée.

3. **Création de plusieurs types de prompts** :

```
# Variante pour la paraphrase
def paraphrase_prompt(text):
    return f"Document context:\n{text}\nInstruction: Paraphraser ce texte en gardant le sens."
# Variante pour Q/R
def qa_prompt(text):
    return f"Document context:\n{text}\nInstruction: Posez 3 questions pertinentes sur ce docu
ment."
```

Workflow Complet

[PDFs] → [Nettoyage Spark] → [Création de Prompts] → [JSONL] → [Fine-Tuning] → [Modèle IA]

Prochaines Étapes

- ✓ **Labellisation** : Remplir manuellement/automatiquement la colonne `completion`.
- ✓ **Fine-Tuning** : Utiliser des librairies comme `transformers` ou `trl` pour adapter un modèle open-source.
- ✓ **Évaluation** : Tester le modèle sur des prompts de validation.

Références :

[1] https://www.google.com/url?sa=i&url=https%3A%2F%2Fnpntraining.medium.com%2Fapache-spark-architecture-9f3fdeffed9c&psig=AOvVaw3uQTDKOGIFAj4MFt8Dh8Le&ust=1745009869408000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCJDXy73634wDFQAAAAAdAAAAABAE