Concepts Fondamentaux de Spark

SOMMAIRE DE LA SÉANCE

Comprendre et définir **le framework Apache Spark**

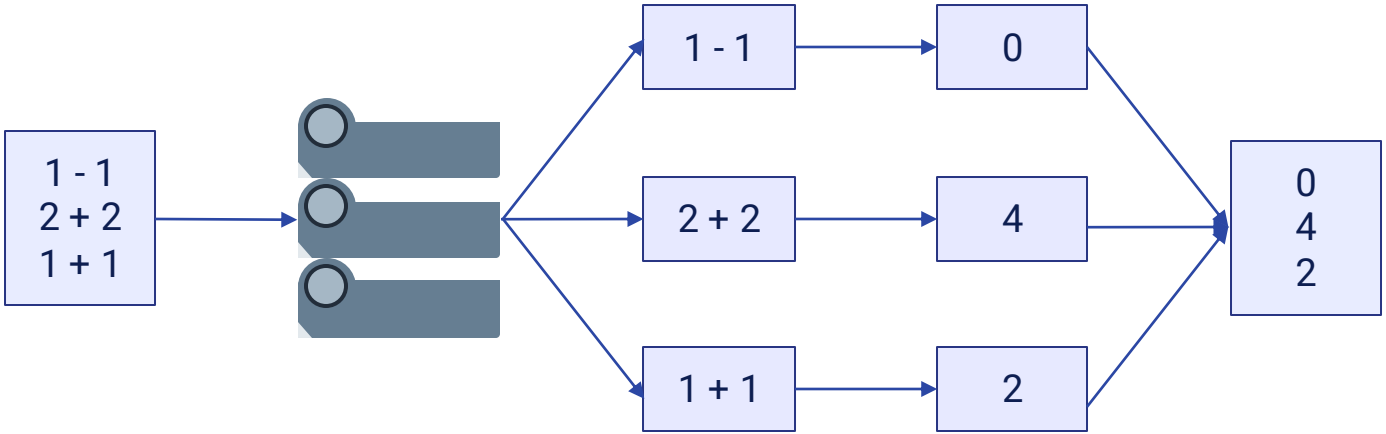
Comprendre les différentes structures de données dans **Apache Spark**

Quelques commandes en **Scala**/**Java/Python** pour la manipulation des **données** avec des **cas pratiques**

1-
Comprendre et définir
le framework Apache Spark

Apache Spark est un framework open source dédié aux calculs distribués.

- Il a été inventé pour gérer et traiter de gros volumes de données réparties sur plusieurs machines ou noeuds
- Développé à la base à l'université de Californie à Berkeley par AMPLab, Spark est aujourd'hui un projet de la fondation Apache

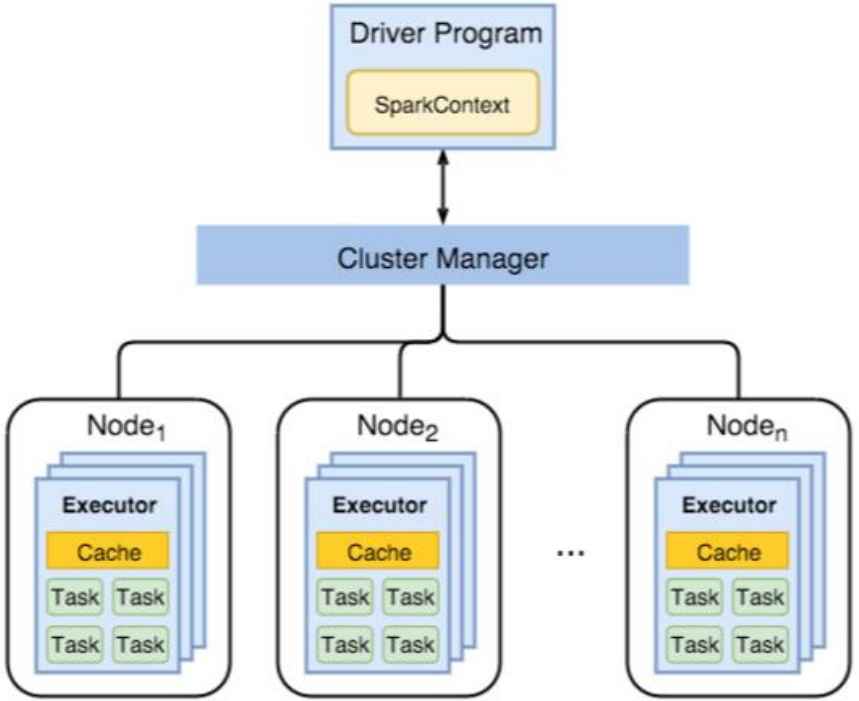


Architecture et fonctionnement de Spark

Spark : vue globale

L'architecture Spark tourne autour de :

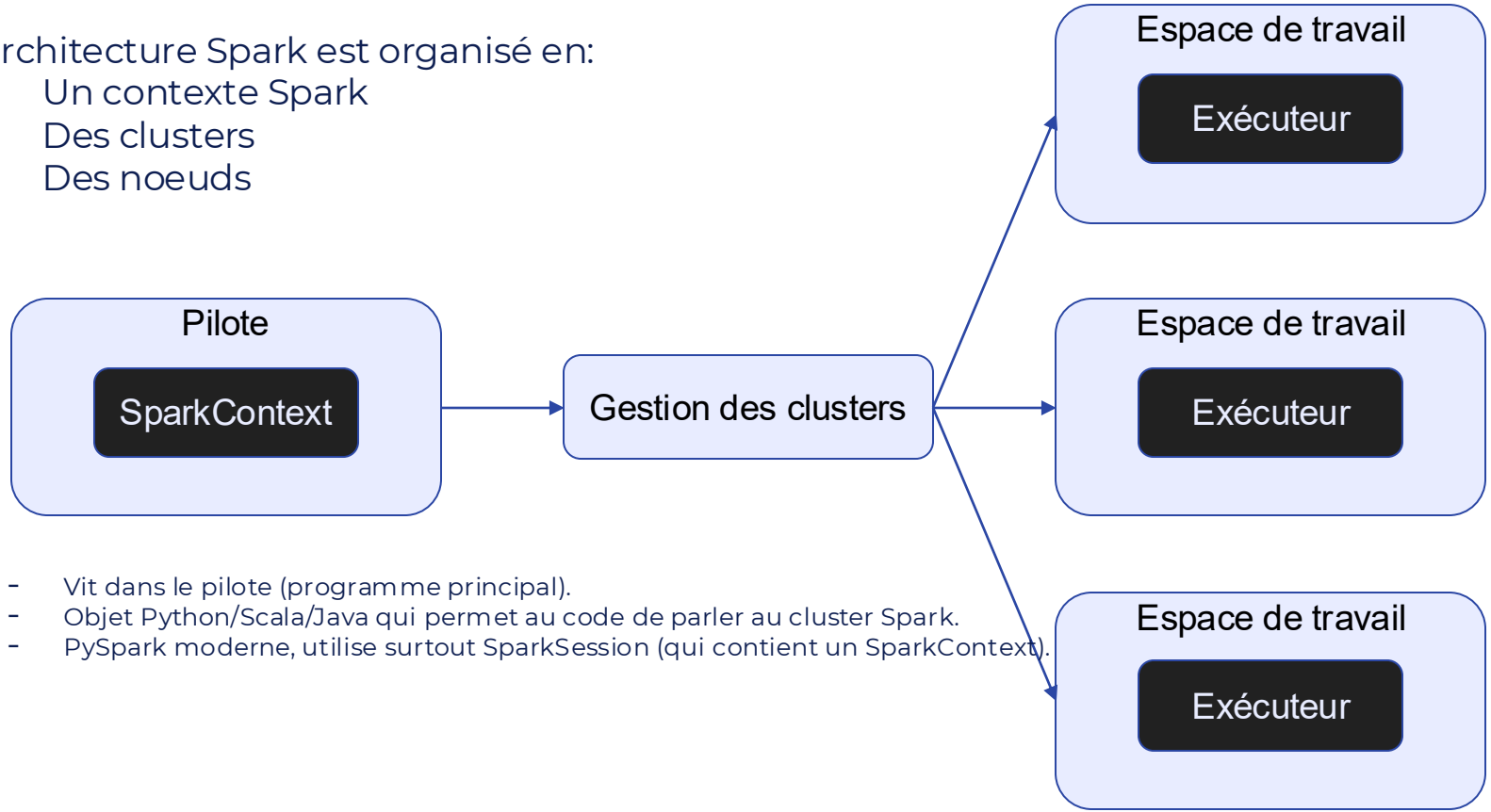
1. Application Spark
2. Driver (avec le SparkContext ou SparkSession)
3. Cluster (ensemble de machines)
4. Nœuds du cluster
    - nœud driver
    - nœuds workers
5. Executors (processus qui exécutent le code sur les workers)
6. Tasks / Jobs / Stages (unités de travail)



Architecture et fonctionnement de Spark

L'architecture Spark est organisé en:

- Un contexte Spark
- Des clusters
- Des noeuds



- Vit dans le pilote (programme principal).
- Objet Python/Scala/Java qui permet au code de parler au cluster Spark.
- PySpark moderne, utilise surtout SparkSession (qui contient un SparkContext).

Architecture et fonctionnement de Spark

- Apache Spark à une architecture master/slave avec deux demons et un cluster manager :
  - Master Daemon –(Master/Driver Process)
  - Worker Daemon –(Slave Process)

- Un cluster spark à un seul Master et un nombre indéterminé de slaves/Workers.

Architecture et fonctionnement de Spark - Driver

L’architecture Spark est organisé en:

- Le **driver** est responsable de l’exécution de la fonction main() de l’application et de créer le SparkContext.
- Spark Driver contient plusieurs composants (DAGScheduler, TaskScheduler, BackendScheduler, BlockManage) qui sont responsables de la traduction du code spark en job (ensemble de taches appelés tasks) qui sera exécuté dans le cluster.
- Le driver planifie l’exécution des jobs et négocie les ressources avec le cluster manager.
- Le Driver enregistre les métadatas des RDD et leurs partitions.

Architecture et fonctionnement de Spark – Cluster Manager

L’architecture Spark est organisé en:

- Un service externe responsable de l’allocation des ressources aux job lancés par le drivers.
- Trois types : Standalone, Mesos et YARN.
- Ces types sont différent en terme de scheduling, sécurité, monitoring

Architecture et fonctionnement de Spark – Executor

L’architecture Spark est organisé en:

- L’**executor** est un agent distribué responsable de l’exécution des taches. Chaque application spark a son propre process executor.
- L’Executors fonctionne tout au long de l’exécution de l’application. On parle de “Allocation statique des Executors”.
- On peut aussi opter pour une allocation dynamique des executors pour équilibrer les charges de traitement
- L’executor lit et écrit les données a partir des sources externes.
- L’Executor stocke les résultats des calculs en mémoire, cache ou sur le disque.

Architecture et fonctionnement de Spark – Worker

L’architecture Spark est organisé en:

- Le **worker**, est un nœud qui permet d’exécuter un programme dans le cluster. Si un process est lance pour une application, l’application requiert des executors dans les nœuds workers
- Dès que le SparkContext se connecte au cluster manager, il acquière un executors dans les worker nodes
- L’executors travaille indépendamment dans chaque tache et peuvent interagirent ensemble.

Architecture et fonctionnement de Spark – SparkContext

L’architecture Spark est organisé en:

- **SparkContext** permet d’établir une connexion avec le cluster manager. Peut être utilisé pour créer les RDD et les accumulateurs, et diffuse les variables dans le cluster.

- Il est recommandé d’avoir un seul SparkContext active par JVM, donc on doit appeler la méthode stop() du SparkContext active avant de créer un autre.

Architecture et fonctionnement de Spark

Exécution d'un job spark :

- Quand un client valide un code spark, le driver traduit le code qui contient des actions et des transformations en un graphe acyclique direct logique (DAG).
- A cette étape, le driver optimise l'ordre de déroulement des transformations et convertit le DAG logique en plan d'exécution physique avec un ensemble d'étapes
- Le driver crée des petites unités physiques d'exécution (les tasks) pour chaque étape.
- Le driver négocie les ressources avec le cluster manager
- Le cluster manager lance les executors dans les nœuds workers au nom du driver.
- A ce moment, le driver envoi les taches au cluster manager en se basant sur l'emplacement des données.

Architecture et fonctionnement de Spark

Exécution d'un job spark :

- Avant l'exécution des taches, ils s'enregistrent avec le driver pour qu'il puissant les surveiller.
- Le Driver planifie aussi les future taches en se basant sur leurs emplacements en mémoire.
- Quand la méthode main du programme driver se termine ou si on lance la méthode stop () du SparkContext, le driver terminera tous les executors et libère les ressources du cluster manager.

Les avantages de Spark

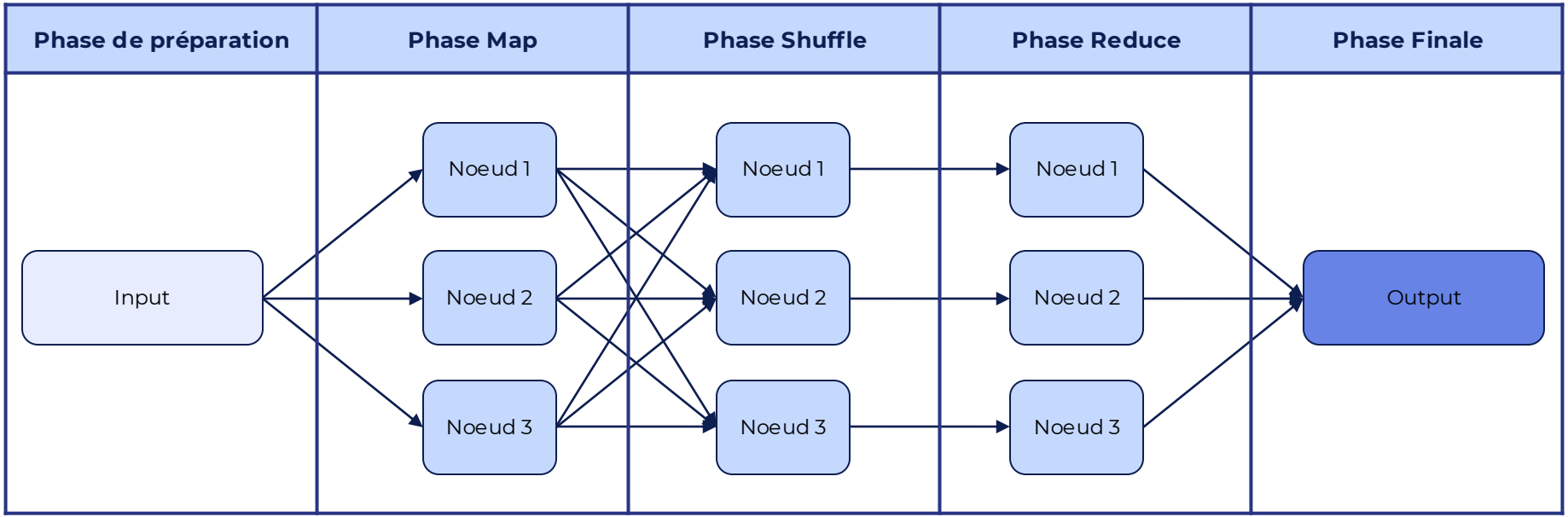
- Beaucoup plus rapide (shuffle en mémoire)
  - **10 fois plus rapide** que Hadoop sur disque et **100 fois plus rapide** en mémoire
- Écrit en **Scala** mais possibilité d'utiliser **Python**, **Java** et **R**
- Librairies spécialisées (**Spark Streaming**, **SparkSQL**, **GraphX**, **SparkML**)
- Exécution en **mémoire**
- Extension du **patron d'architecture MapReduce**
- Facile à **développer**

Limitations de Spark

- Pas de support pour le traitement en temps réel
- Problèmes avec les fichiers de petite taille
- Pas de système de gestion des fichiers: Spark est principalement un système de traitement, et ne fournit pas de solution pour le stockage des données. Il doit donc se baser sur d'autres systèmes de stockage tel que Hadoop HDFS ou Amazon S3.
- Coûteux

Opérations selon le patron d'architecture MapReduce

- MapReduce est à la base une technique d'**Hadoop.**



2-

Comprendre les différentes **structures de données** dans **Apache Spark**

Les structures de données dans Spark

Il existe trois types de structures de données dans Spark:

- Les RDD (Resilient Distributed Dataset) ayant divers qualités
  - Structure de données la plus simple de Spark
- Les Datasets
  - Tous les avantages des RDD et de Spark SQL
  - Depuis Spark 2.0 cette structure de données n'existe plus pour Python ou R (nous verrons ensuite pourquoi)
- Les Dataframes
  - Tous les avantages des RDD et des Dataframes en plus d'une API plus détaillée
  - Depuis Spark 2.0 cette nouvelle structure est une occurrence particulière des Datasets

Les RDD (Resilient Distributed Dataset)

- **Resilient** : supporte la tolérance aux pannes grâce à une modélisation du processus d'exécution par un DAG (Directed Acyclic Graph), et au recalcul des partitions manquantes.
- **Distributed** : données distribuées sur les nœuds du cluster.
- **Dataset :** collection de données partitionnée.
- Les RDD ont plusieurs caractéristiques intéressantes:
  - Le RDD est découpé en plusieurs blocs
  - Une RDD est associée à une liste de dépendances avec les RDD parents
  - Une RDD dispose d'une fonction pour calculer une partition
  - Optionnellement, un objet **Partitioner** pour les RDD de type clé/valeur
  - Optionnellement, une liste indiquant l'emplacement pour chaque partition

Les RDD (Resilient Distributed Dataset)

Chaque RDD possède des opérations et une méthode d'évaluation paresseuse :

- Un RDD supporte deux type d'opérations, les opération de **Transformation** et les opérations d'**Action**:
  - Une transformation consiste à appliquer une fonction sur *1 à n RDD* et à retourner un nouveau **RDD**
  - Une action consiste à appliquer une fonction et à retourner une valeur
- Les transformations sont paresseuses, évitant le calcul inutile. Ceci favorise l'optimisation du traitement
- Un RDD transformé est calculé lorsqu'une action est appliquée sur ce dernier

Les RDD (Resilient Distributed Dataset)

Certaines fonctions sont disponibles seulement sur certains types de RDD :

- mean, variance, stdev pour les RDD numériques
- Join pour les RDD clé/valeur
- L'enregistrement de fichier utilisant des RDD basées sur des fichiers de format séquentiel.

En général, on va notamment retrouver le plus souvent les types suivants:

- PairRDDFunctions
- DoubleRDDFunctions
- SequenceFileRDDFunctions

Les Datasets

- Les Datasets reprennent toute la structure déjà disponible dans les RDD et l'enrichit en ajoutant d'autres méthodes et fonctions.
- Un Dataset est une collection **fortement typée** d'objets (spécifiques à un domaine) qui peut être transformé en parallèle à l'aide d'opérations fonctionnelles ou relationnelles.
- Les opérations disponibles sur les Datasets sont divisées en **Transformations** et **Actions** (comme pour les RDD). Les transformations sont celles qui produisent de nouveaux Datasets, et les actions sont celles qui déclenchent des calculs et renvoient des résultats.
- Les exemples de transformations incluent les méthodes :
    - **map**, **filter**, **select** et **aggregate** (**groupBy**)
- Des exemples d'actions sont:
    - le **comptage**, l'**affichage** ou l'**écriture** de données sur des systèmes de fichiers.

Les Datasets

- Les Datasets sont eux aussi "paresseux". En interne, un Dataset représente un plan logique qui décrit le calcul nécessaire pour produire les données. Lorsqu'une action est invoquée, l'optimiseur de requêtes de Spark optimise le plan logique et génère un plan physique pour une exécution efficace de manière parallèle et distribuée
    - Pour explorer le plan logique ainsi que le plan physique optimisé, on peut utiliser la fonction **explain**
- Il y a un encodeur qui fait correspondre le type **T** (spécifique au domaine) au système de type interne de Spark. Un encodeur est utilisé pour dire à Spark de générer du code au moment de l'exécution pour sérialiser **T** dans une structure binaire. Cette structure binaire a souvent une empreinte mémoire beaucoup plus faible et est optimisée pour l'efficacité du traitement des données (par exemple, dans un format en colonnes).
    - Pour comprendre la représentation binaire interne des données, on peut utiliser la fonction **schema**

Les Datasets

- Pour utiliser un objet de type Dataset, il ne faut plus SparkContext mais une **SparkSession** qui permet d'avoir accès à cet API, la SparkSession contient de plus un SparkContext permettant l'utilisation des RDD (même si cela peut s'avérer fastidieux dans certains langages comme Java)
- Il existe généralement deux façons de créer un Dataset :
    - La manière la plus courante est de faire pointer Spark vers certains fichiers sur les systèmes de stockage, en utilisant la **fonction de lecture** disponible sur un SparkSession.
    - Les ensembles de données peuvent également être créés par le biais de **transformations** disponibles sur les ensembles de données **existants**
- Les RDD sont créés principalement de la même façon.

Les DataFrames

- Un DataFrame est un ensemble de données organisé en colonnes nommées. Il est conceptuellement équivalent à une table dans une base de données relationnelle ou à un DataFrame dans R/Python, mais avec des optimisations plus riches sous le capot.
- Les DataFrames peuvent être construits à partir d'un large éventail de sources telles que : des fichiers de données structurés, des tables dans Hive, des bases de données externes ou des RDDs existants.
- L'API DataFrame est disponible en Scala, Java, Python et R. En Scala et Java, un DataFrame est représenté par Dataset typé par des Row.
- Dans l'API Scala, DataFrame est simplement un alias de type de Dataset[Row]. Alors que, dans l'API Java, les utilisateurs doivent utiliser Dataset<Row> pour représenter un DataFrame.

Les DataFrames

- Depuis la mise à jour 2.0 et la fusion des API DataFrame et Dataset, les Datasets n'existent plus ni en Python ni en R (ces langages de programmation ne disposant pas de typage pour leurs variables).
- Dans Java et Scala, les objets Row sont utilisés pour donner la possibilité à un DataFrame d'avoir des colonnes de types différents. On possède alors dans les DataFrame aussi un schéma indiquant le type de chacune des colonnes et d'autres informations
- Il est aussi possible de construire des RDD dont le type correspond à des Row, on perd cependant la structure des DataFrames et leur organisation par colonne.

3-

Quelques commandes en **Scala**/**Python** pour la manipulation des **données**

À titre informatif

L’installation de **Spark** est assez **fastidieuse** et demande de suivre quelques étapes précises, vous pouvez trouver un guide pour utiliser **Spark** dans **Python** avec des **Notebooks Jupyter** ici.

Principales manipulations sur les RDD

Afin d'utiliser la structure de données RDD, nous devons tout d'abord créer a minima un SparkContext, le SparkContext étant le point d'entrée entre votre langage de programmation et Spark. Pour importer le SparkContext vous pouvez utiliser les commandes suivantes:

En Scala:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
```

En Java:

```
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf;
```

En Python:

```
from pyspark import SparkContext, SparkConf
```

Principales manipulations sur les RDD

Une fois les imports réalisés, on peut créer un SparkConf puis un SparkContext.
Le SparkConf sert à configurer l'application (nom, type de cluster, options...), et le SparkContext est le point d'entrée vers Spark.

En Scala:

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val sc = new SparkContext(conf)
```

En Java:

```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

En Python:

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

Principales manipulations sur les RDD

On peut maintenant utiliser l'objet **sc** pour créer ou lire des fichiers dans des RDD:

- avec parallelize : transformer une collection locale (liste, tableau...) en RDD
- avec textFile : lire n'importe quel fichier texte (fichier log, CSV brut, JSON ligne par ligne...) et renvoie un RDD de chaînes de caractères, une ligne par élément.

En Scala:	`val data = Array(1, 2, 3, 4, 5)` `val distData = sc.parallelize(data)`	`val distFile = sc.textFile("data.txt")`
En Java:	`List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);` `JavaRDD<Integer> distData = sc.parallelize(data);`	`JavaRDD<String> distFile = sc.textFile("data.txt");`
En Python:	`data = [1, 2, 3, 4, 5]` `distData = sc.parallelize(data)`	`distFile = sc.textFile("data.txt")`

Pour Java: `import org.apache.spark.api.java.JavaRDD;`.

Principales manipulations sur les RDD

Maintenant que notre RDD est définie on peut utiliser différentes méthodes classiques sur celui-ci. On peut par exemple sélectionner une fraction du RDD avec ou compter le nombre de lignes que celui contient avec :

En Scala:	`distFile.take(10)` `println(distFile.take(10).mkString("\n"))`	`println(distFile.count())`
En Java:	`distFile.take(10);` `distFile.take(10).forEach(line -> System.out.println(line));`	`System.out.println(distFile.count());`
En Python:	`distFile.take(10)` `print(distFile.take(10))`	`print(distFile.count())`

On constate que les méthodes ont les mêmes noms qu'importe le langage de programmation, cependant la nature des objets renvoyés n'est pas forcément la même !

Opérations sur les RDD

Pour rappel, un RDD supporte deux type d'opérations, les opérations de **Transformation** et les opérations d'**Action**.

- On retrouve parmi les opérations sur les RDD `map` qui est une transformation appliquant à **chaque élément** du RDD une fonction retournant ensuite un nouveau RDD avec les résultats.
- On a d'un autre côté l'opération `reduce` qui est une action qui va **agréger** les éléments d'un RDD en utilisant des fonctions et retourner le résultat dans le **programme pilote**.
- Il existe aussi une autre action `reduceByKey` qui va effectuer la même opération que `reduce` mais selon des clés spécifiques.

On retrouve le schéma **MapReduce** que nous avions déjà vu précédemment, on va donc reprendre ce concept et le détaillé dans un exemple que nous coderons ensuite en **pratique** avec Spark.

Exemple d’opérations: comptage de mots dans Spark

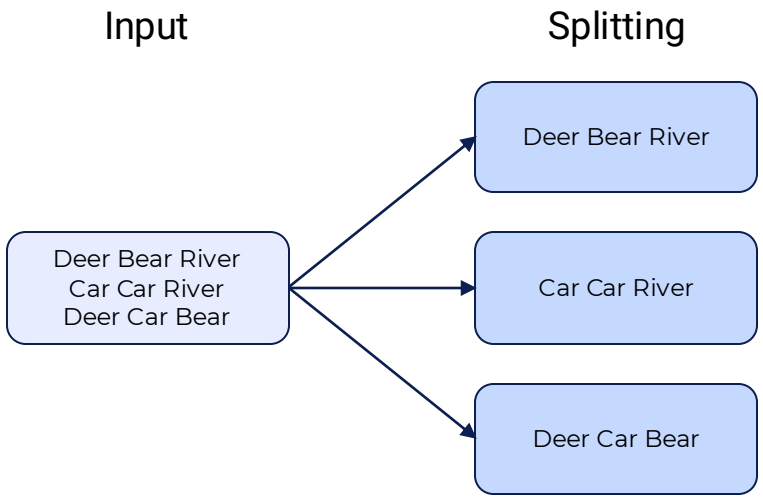
L’un des exemples classiques d’application de Spark que l’on retrouve dans de nombreux cours est celui du dénombrage des mots d’un texte. Cette tâche assez simple est hautement parallélisable en utilisant le fonctionnement de Spark, dans l’exemple que nous allons traiter, le texte dont nous voudrions dénombrer les mots est le suivant:

Deer Bear River
Car Car River
Deer Car Bear

Nous allons donc dans un premier temps présenter le fonctionnement du patron d’architecture dans cet exemple précis.

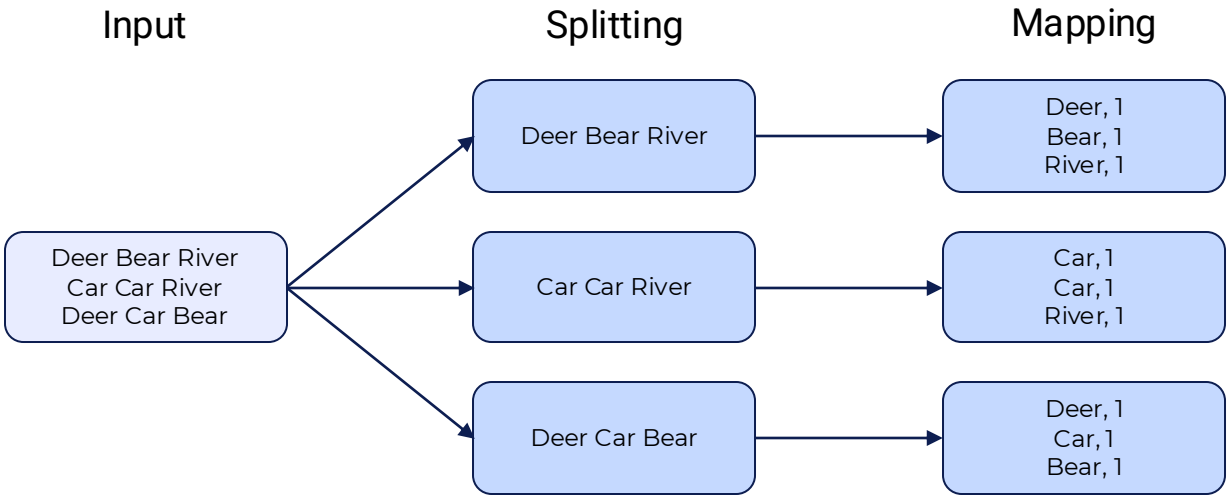
Exemple du comptage de mots dans Spark

Etape 1 : Splitting sur les noeuds



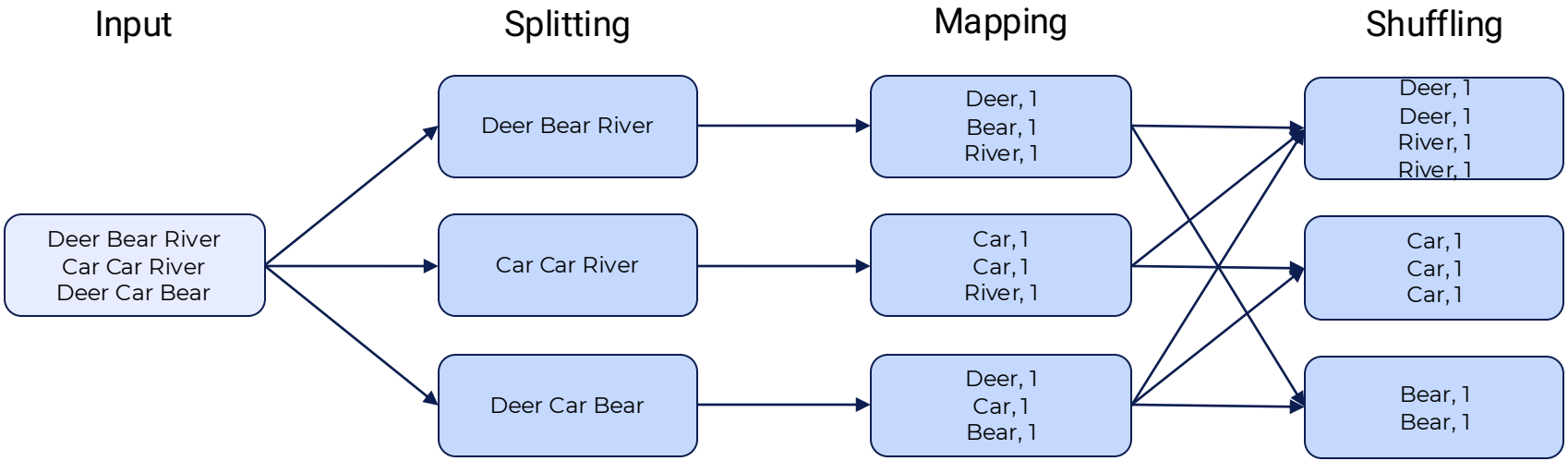
Exemple du comptage de mots dans Spark

Etape 2 : Mapping sur les noeuds



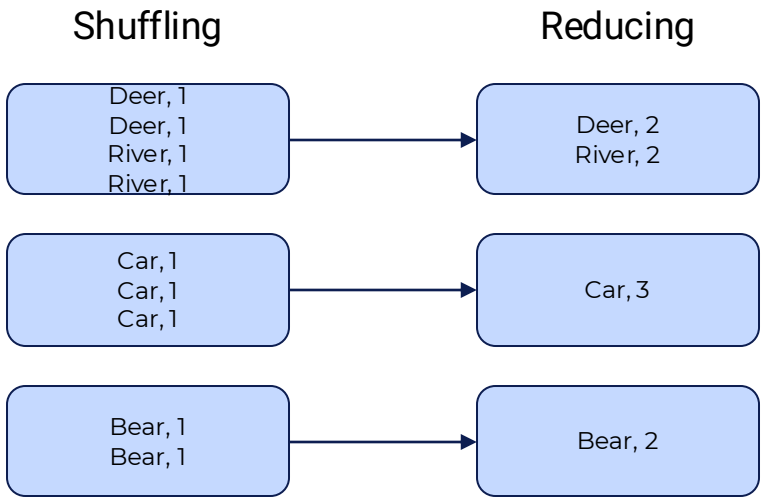
Exemple du comptage de mots dans Spark

Etape 3 : Shuffling entre les noeuds



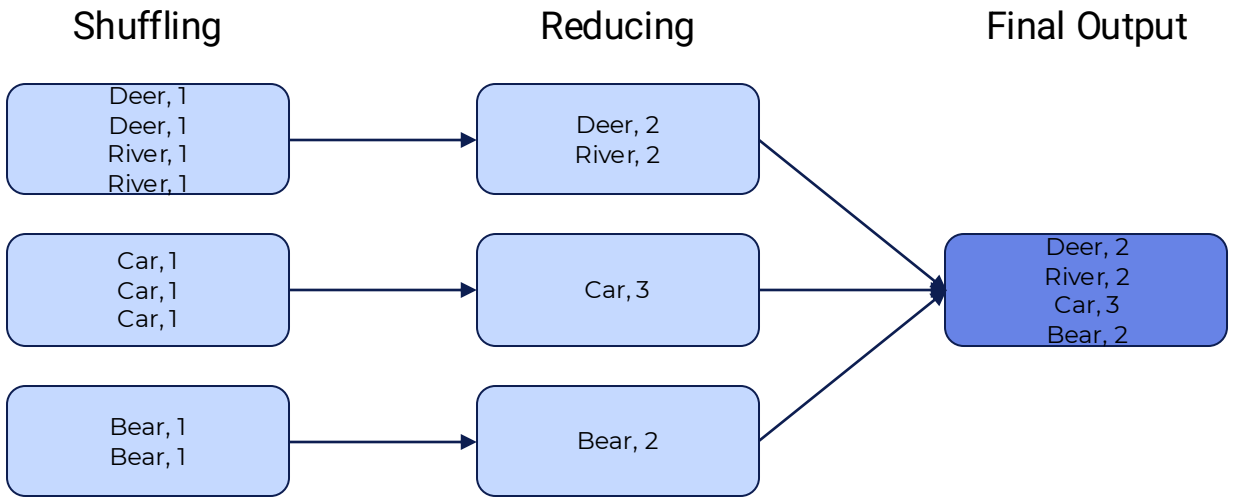
Exemple du comptage de mots dans Spark

Etape 4 : Reducing sur les noeuds

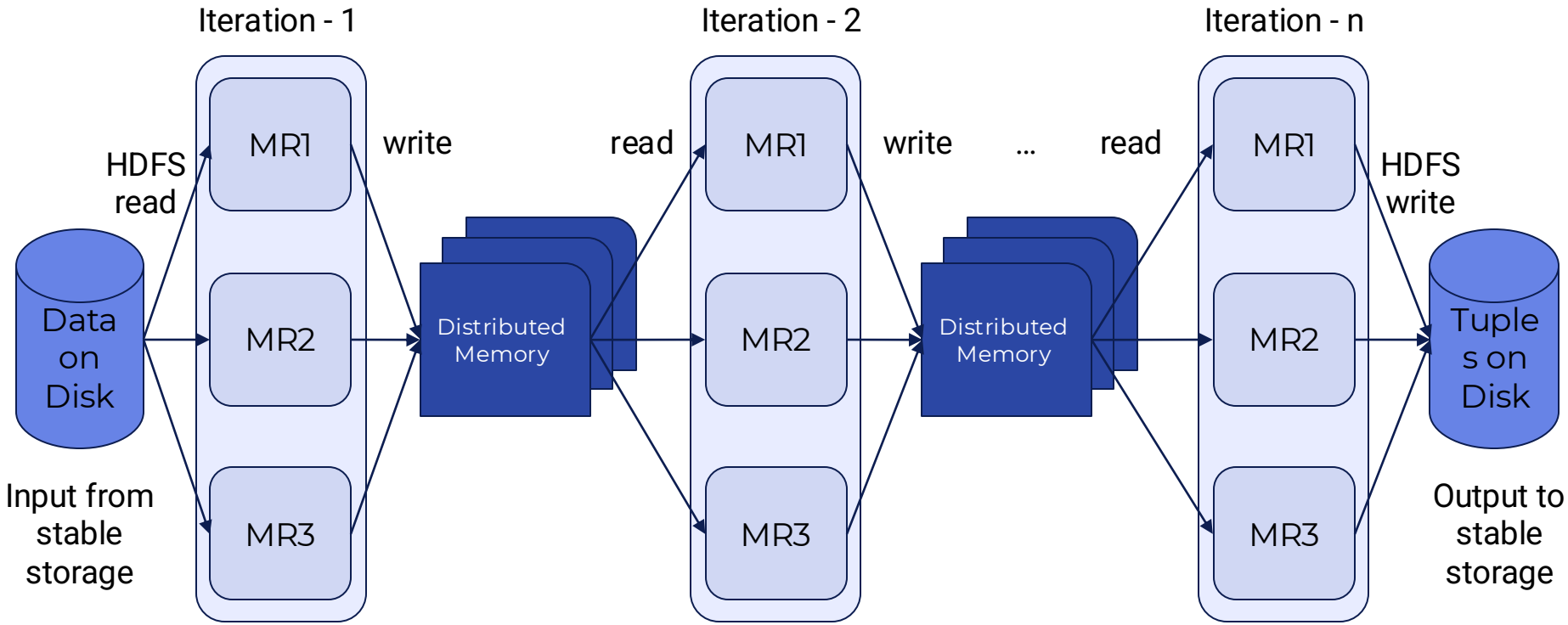


Exemple du comptage de mots dans Spark

Etape 5 : Résultat final



Traitement des données en mémoire



Principales manipulations sur les Datasets/DataFrames

Pour utiliser les Datasets et DataFrames, à l'image des RDD il nous faut un point d'entrée, la SparkSession (celui-ci contient d'ailleurs le point d'entrée des RDD). Les commandes suivantes servent à l'importer:

En Scala/Java:

```
import org.apache.spark.sql.SparkSession
```

En Python:

```
from pyspark.sql import SparkSession
```

Principales manipulations sur les Datasets/DataFrames

Vous pouvez ensuite définir votre SparkSession comme suit:

En Scala:

```scala
val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()
```

En Java:

```java
SparkSession spark = SparkSession
  .builder()
  .appName("Java Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();
```

En Python:

```python
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Vous avez maintenant les outils nécessaires pour travailler avec des Datasets et des DataFrames !