Université Constantine 2
جامعة قسنطينة 2

TDSAI

Chapitre 2 : SPARK
-Partie 1-

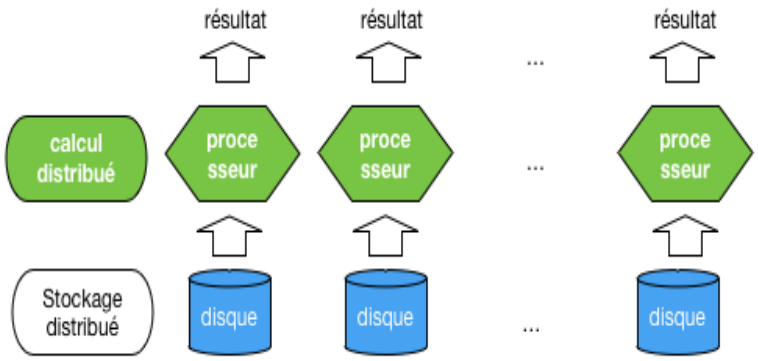
Dr. S.ZERABI
Faculté des NTIC
Soumeya.zerabi@univ-constantine2.dz

Etudiants concernés

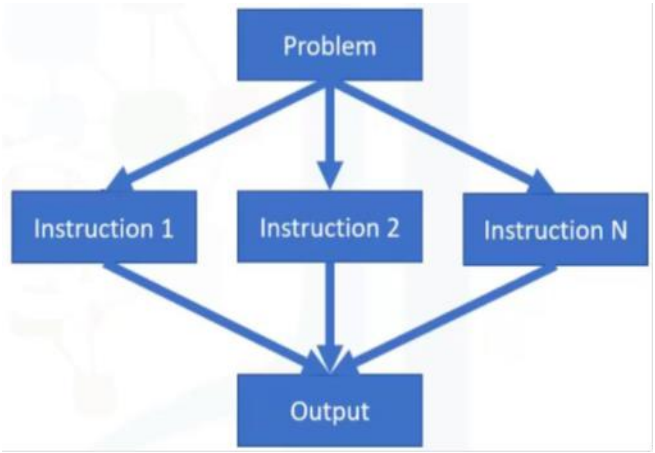
Faculté/Institut	Département	Niveau	Spécialité
Nouvelles technologies	IFA	M1	SDIA

Introduction

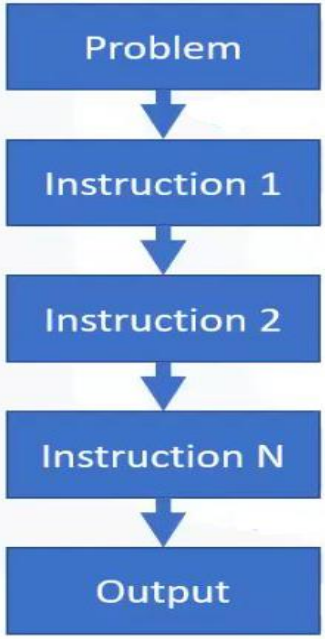
Calcul distribué

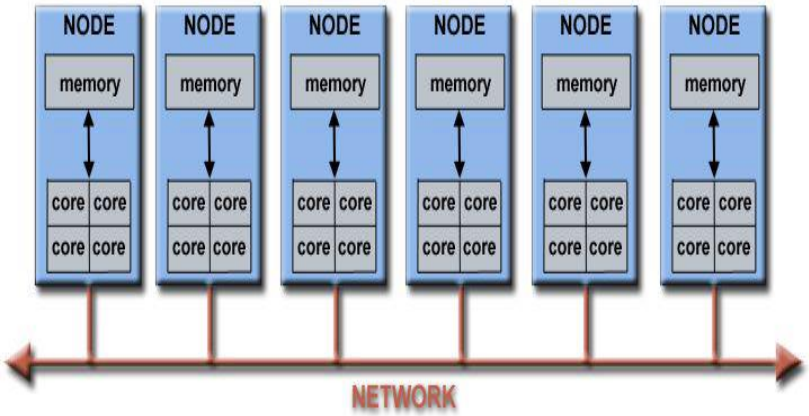


Calcul parallèle



Calcul en série





Calcul parallèle VS calcul distribué

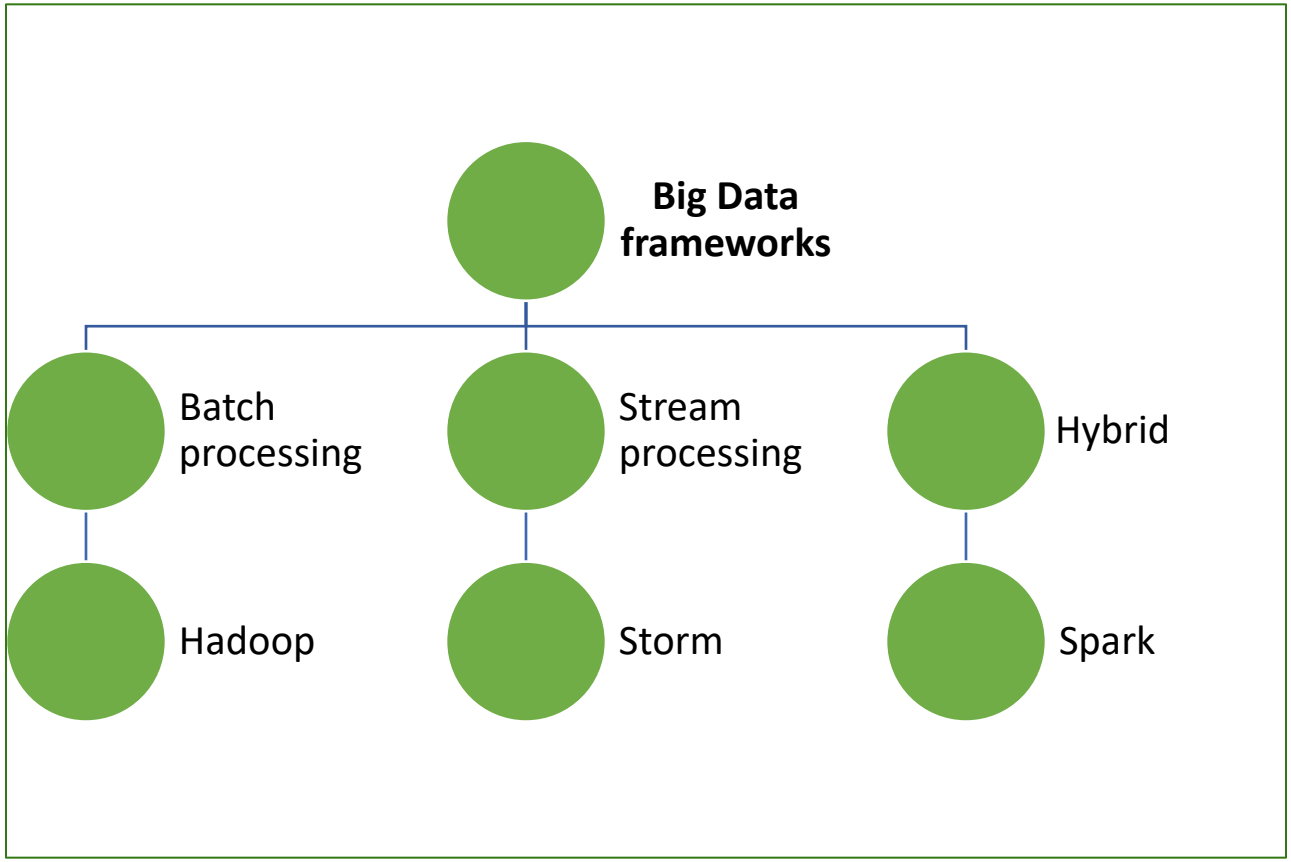
Calcul parallèle	Calcul distribué
Le calcul est effectué sur des threads	Les calculs sont distribués sur des nœuds autonomes
Les threads sont exécutés en parallèle	Aucune ressource n'est partagée
Les threads partagent une mémoire commune	Les nœuds sont distribués sur un cluster et communiquent par l'envoie de message
Le passage a l'échelle est **vertical** (augmenter la puissance des processeurs)	Le passage a l'échelle est **horizontal** (ajouter des nœuds)
La tolérance aux pannes est complexe	La tolérance aux pannes est facile

Un cluster (grappe de calcul)

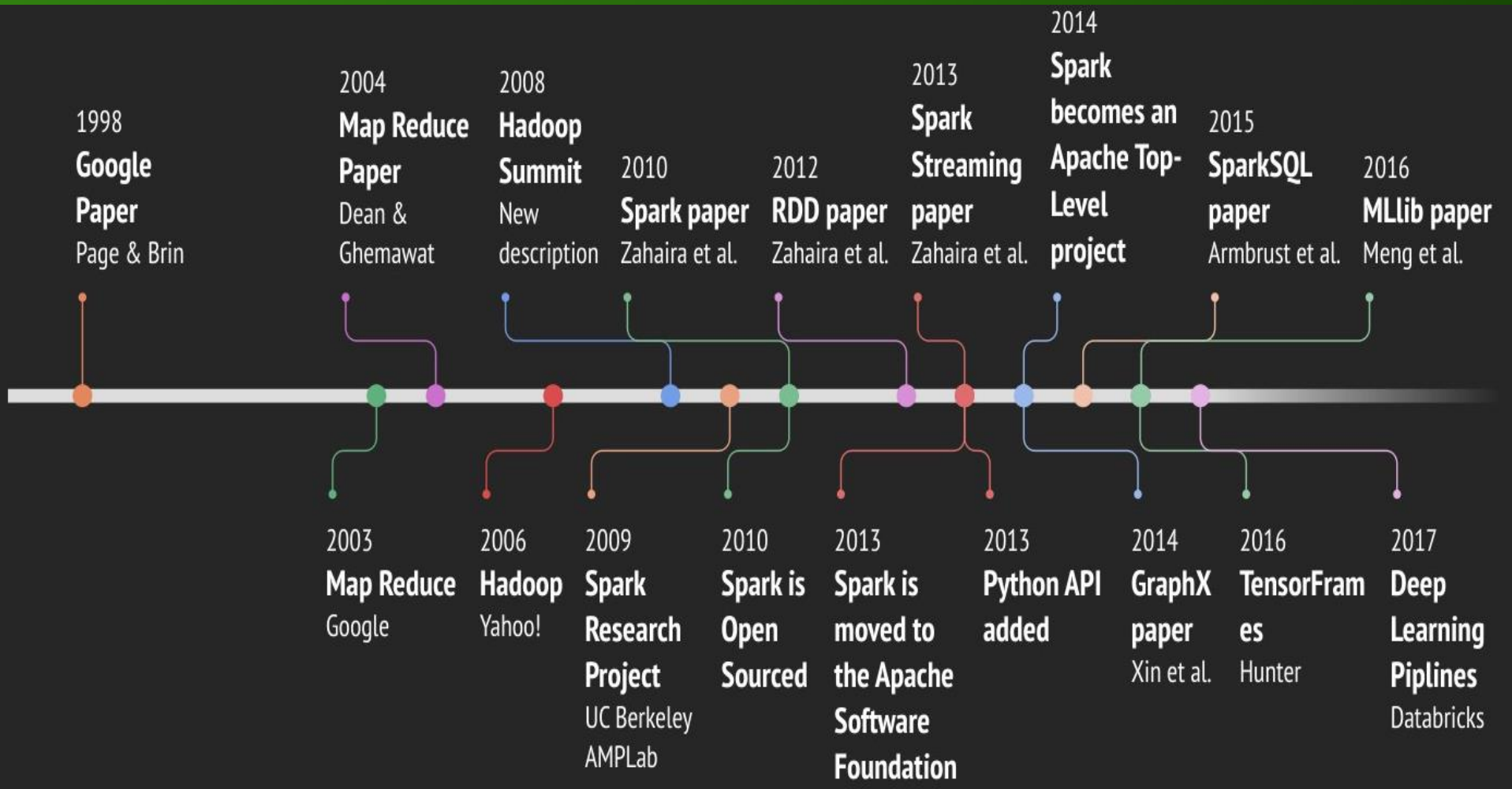
Collection de grand nombre d'ordinateurs (appelés nœuds) **autonomes** (CPU, RAM, DD, système…), extrêmement coûteux, relies entre eux afin d'effectuer des calculs à haute performance (HPC).

Ecosystème des Big Data

Introduction

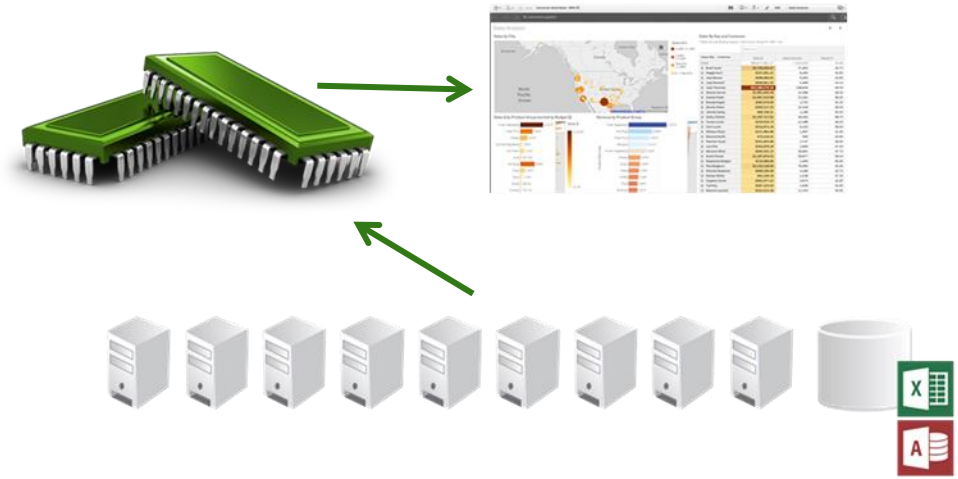


Historique de Spark



C’est quoi Spark?

- **Spark** est un framework libre, écrit en **Scala**.
- En 2009, développé par *Matei Zaharia, AMP Lab, Berkeley, Californie.*
- En 2010, il est open source.
- En 2013, la fondation Apache.
- Destiné à faciliter la création d'applications **parallèles** et **distribuées** .
- Traitement **en mémoire.**



Pourquoi Spark?

Rapidité
- 100x plus rapide que Hadoop (en mémoire)
- 10x plus rapide que Hadoop (en disque)

Traitement
- Supporte les taches itératives et interactives.
- Traitement en mémoire

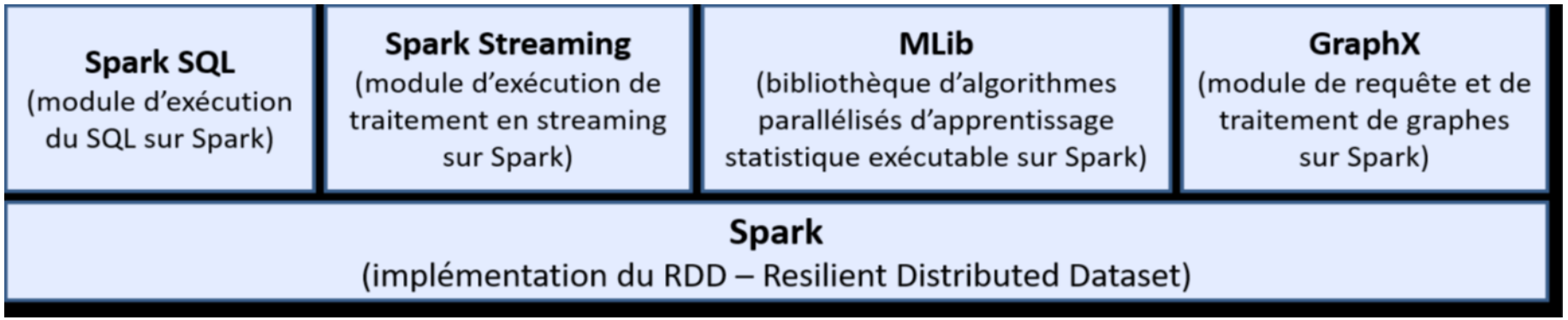
Poly glot
- APIs en Scala, Java, Python et R.

Déploiement
- YARN (Hadoop), Mesos, cluster Spark autonome (standalone).

General
- Librairies spécialisées

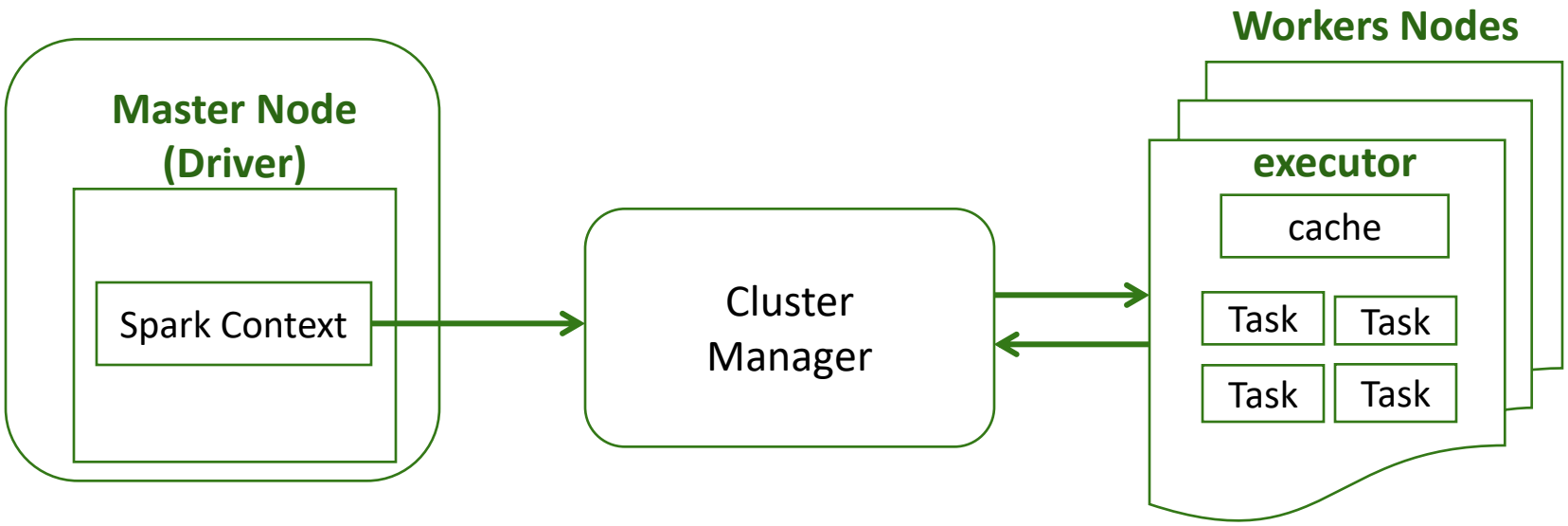
Ecosystème de Spark

- Spark contient 4 modules chacun est spécialisé dans une problématique particulière.



- Ces modules peuvent être exploitées via les langages de programmation: Java, Scala ou Python.

Architecture de Spark



Structures de données dans Spark

Il existe trois types de structures de données dans Spark:

- Les **RDDs.**
- Les **dataframes**
- Les **datasets.**

RDD (Resilient Distributed Datasets)

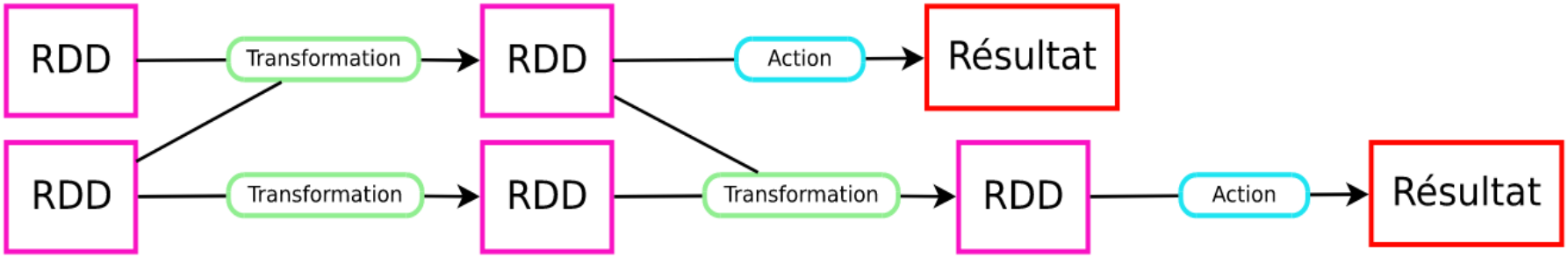
- Abstraction de base de données distribuées dans Spark.
- **Resilient:** supporte la tolérance aux pannes grace a la modélisation du processus d'exécution par DAG et le recalcule des partitions manquantes.
- **Distributed:** distribuée sur les nœuds du clusters ce qui permet aux développeurs d'effectuer des calculs parallèles en mémoire sur le cluster.
- **Dataset:** collection de données partitionnées selon le nombre de machines ou de cœurs disponibles.

Avantages des RDDs

Les RDDs ont hérité les caractéristiques des collections (tableaux, listes, tuples) de scala:

- **Calcul paresseux (Lazy computation):** les transformations exécutées sur les RDDs sont paresseux c.à.d. ne sont pas exécutées telles qu'elles apparaissent au driver mais lorsqu'une action est effectuée.
- **Immuabilité (Immutability) :** en lecture seule (ne pas les modifier après leur création).
- **En mémoire (In-memory):** les RDDs sont exécutés en mémoire centrale (soit en mémoire cache) évitant ainsi la réplication des données sur disque.
- **Tolérance aux pannes**

RDD-Tolérance aux pannes



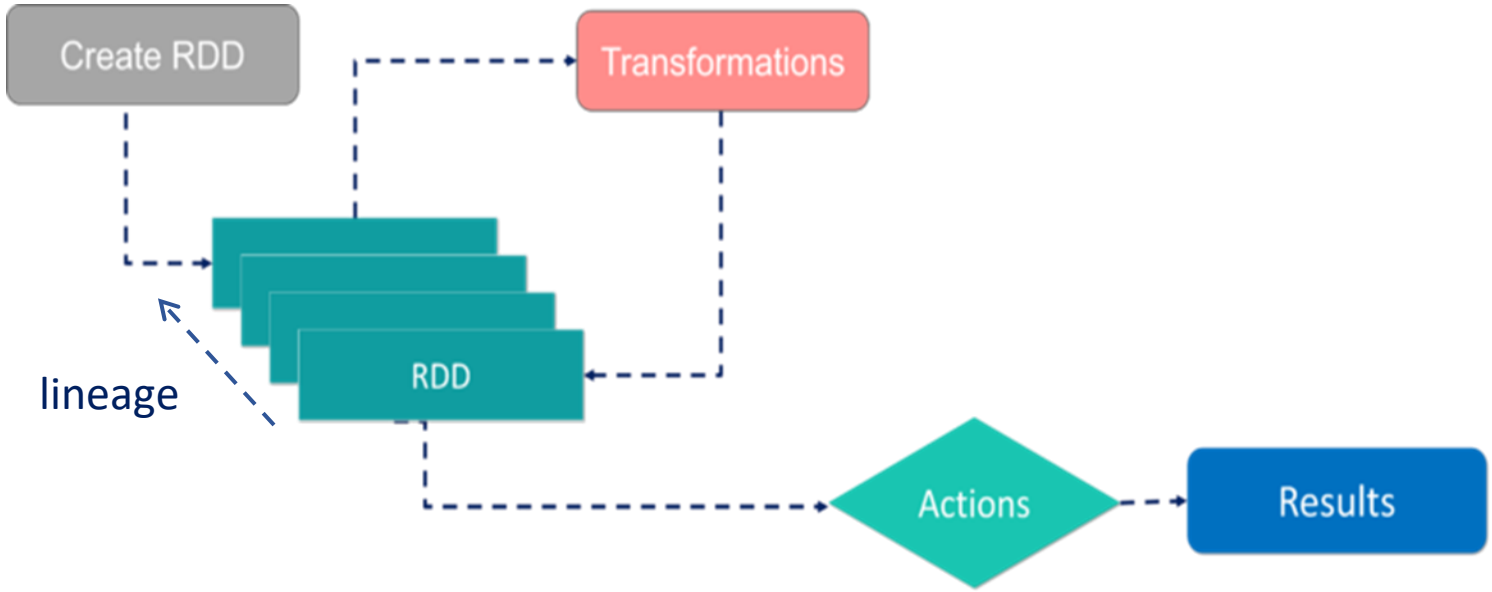
- Dans une application Spark, les transformations et les actions réalisées sur les RDDs permettent de construire un **graphe acyclique orienté** (**DAG** : "directed acyclic graph").
- Lorsqu'un nœud devient indisponible ou une opération RDD échoue, il peut être régénéré à partir de ses nœuds parents grâce aux DAG (**lignage**).
- La tolérance aux pannes dans Spark s'appuie essentiellement sur la préservation de cet historique afin de reconstituer le RDD.

Opérations sur les RDDs

- **Transformations**
- **Actions**

Opérations sur les RDDs

- **Les transformations:** créer un nouveau RDD (fichier/RDD) en transformant le RDD en entrée en un RDD en sortie (RDD to RDD).
- **Les actions:** effectue une action sur un RDD et retourne une valeur ou une collection à partir d'un RDD.



Comment executer un code Spark ?

- **Installation sous Windows**
- **Installation sous Ubuntu**
- **Utilisation du cloud (Google Colab, Databricks, etc).**

Création de RDD avec pyspark

```
#import SparkSession
from pyspark.sql import SparkSession
#create SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

A partir d'une liste

```
list = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
```

A partir d'une paire (clé, valeur)

```
list = spark.sparkContext.parallelize([('a', 3), ('b', 5), ('c', 4), ('d',10)])
```

A partir d'un fichier existant

```
list = spark.sparkContext.textFile("input.txt")
```

Création de RDD avec pyspark

- **SparkSession**

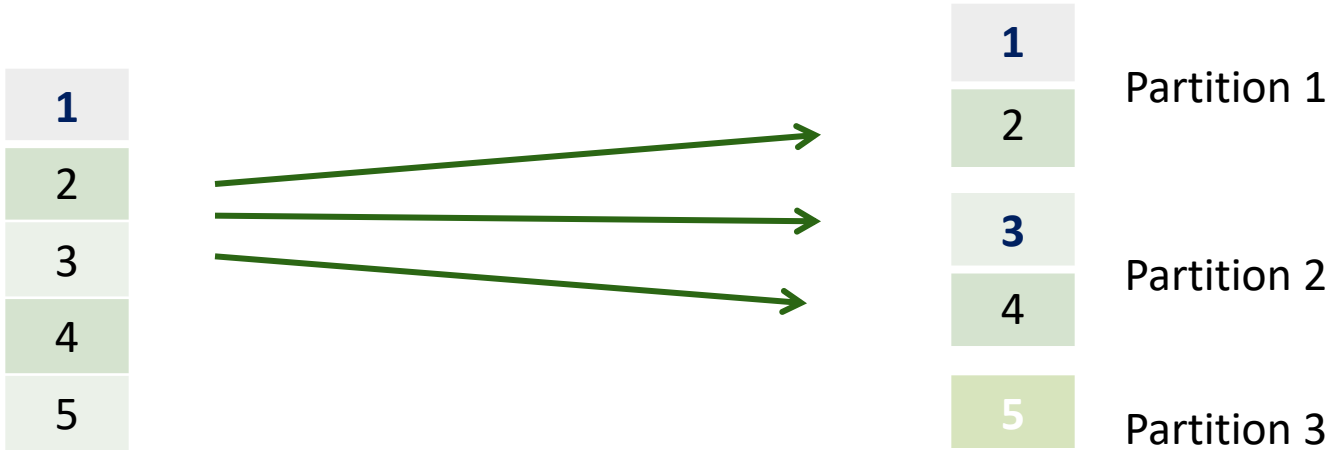
La session Spark est le point d'entrée pour interagir avec Spark.

- **master:** indique l'URL du nœud maitre

  (**Yarn, Mesos** (cluster), **local [n]** (standalone)).

- **getOrCreate()**

Créer la session Spark sur la base des paramètres ou appeler une session Spark existante.



Création de RDD à partir d'une liste

Transformations sur les RDDs

Transformation	signification
map()	Crée un nouveau RDD avec les conditions spécifiées
filter()	Crée un nouveau RDD contenant les données répondant aux conditions du filtre
flatmap()	Crée un nouveau RDD après l'application de map() puis aplatit les données
mapValues()	Transforme chaque **valeur** via la fonction map() tout en conservant la partie clé.
distinct()	Renvoie un nouveau RDD contenant les éléments distincts du RDD
groupBy()	Collecte les données identiques dans des groupes
groupByKey()	Regroupe les valeurs de chaque clé du RDD puis créer un nouveau RDD en appliquant une fonction d'aggrégation
reduceByKey()	Fusionne les valeurs de chaque clé du RDD à l'aide d'une fonction reduce ()

Transformations sur les RDDs

Transformation	signification
union()	Union de deux ou plusieurs RDD
join()	Jointure de deux ou plusieurs RDD
sample()	Crée un nouveau RDD constitué d'un échantillon de données du RDD source.
sortBy()	Tri un RDD selon la fonction spécifiée.
sortByKey()	Tri du RDD par clé.

map()

Permet d'appliquer les opérations sur les enregistrements du RDD, le RDD sorti aura toujours le même nombre d'enregistrements que le RDD initial.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()

list=spark.sparkContext.parallelize([1,2,3,4,5])

list.map(lambda x:x+1).collect()
```

```
[2, 3, 4, 5, 6]
```

filter()

Permet de filtrer les enregistrements du RDD.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()

list=spark.sparkContext.parallelize([1,2,3,4,5])

list.filter(lambda x: x>3).collect()
```

```
[4, 5]
```

flatMap()

Renvoyez un nouveau RDD en appliquant d'abord la fonction map() à tous les éléments de ce RDD, puis en aplatissant les résultats.

```
rdd=spark.sparkContext.parallelize([3,4,5])
rdd.flatMap(lambda x:range(1,x)).collect()
```

```
[1, 2, 1, 2, 3, 1, 2, 3, 4]
```

mapValues()

Renvoie un nouveau RDD en effectuant la fonction map() sur les valeurs tout en conservant les clés des paires (clé, valeur).

```
list=spark.sparkContext.parallelize([('A',["pomme","ananas","raisin"]),('B',["citron","orange"])])

def f(x):
  return(len(x))

list.mapValues(f).collect()
```

```
[('A', 3), ('B', 2)]
```

```
list=spark.sparkContext.parallelize([('A',[4,5]),('B',[10,15])])

list.mapValues(sum).collect()
```

```
[('A', 9), ('B', 25)]
```

distinct()

Renvoie un nouveau RDD sans les éléments doublons.

```
spark.sparkContext.parallelize([1, 1, 2, 3]).distinct().collect()
```

```
[2, 1, 3]
```

groupBy()

Renvoie un nouveau RDD en groupant ses éléments selon une condition.

```
x=spark.sparkContext.parallelize([1,2,3,4,5,6])
y=x.groupBy(lambda x: x%2==0).collect()
y
```

```
[(False, <pyspark.resultiterable.ResultIterable at 0x7cae10f9b250>),
 (True, <pyspark.resultiterable.ResultIterable at 0x7cae10f9b1f0>)]
```

```
x=spark.sparkContext.parallelize([1,2,3,4,5,6])
y=x.groupBy(lambda x: x%2==0).mapValues(set).collect()
y
```

```
[(False, {1, 3, 5}), (True, {2, 4, 6})]
```

groupByKey()

Regroupe les valeurs de chaque clé du RDD en une seule séquence (afin d'effectuer une agrégation).

```
x = spark.sparkContext.parallelize([("a", 1), ("b", 1), ("a", 1)])
y=x.groupByKey().collect()
print(y)
```

```
[('b', <pyspark.resultiterable.ResultIterable object at 0x7cae10f9a260>), ('a', <pyspark.resultiterable.ResultIterable object at 0x7cae10f9b910>
```

```
rdd = spark.sparkContext.parallelize([("a", 1), ("b", 1), ("a", 1)])
y=rdd.groupByKey().mapValues(sum).collect()
print(y)
```

```
[('b', 1), ('a', 2)]
```

reduceByKey()

Effectue pour chaque clé unique du RDD une opération reduce() **séparément** sur les valeurs du RDD.

```
rdd = spark.sparkContext.parallelize ([(1,10),(4,50),(1,-2),(3,20),(3,2)])
rdd.reduceByKey(lambda x,y: x+y).collect()
```

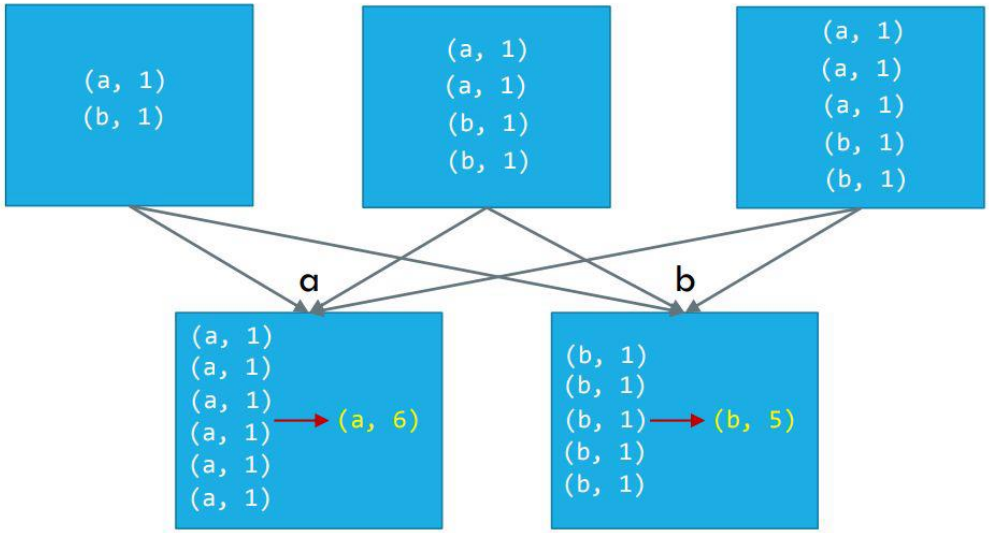
```
[(4, 50), (1, 8), (3, 22)]
```

```
from operator import add
rdd = spark.sparkContext.parallelize ([(1,10),(4,50),(1,-2),(3,20),(3,2)])
rdd.reduceByKey(add).collect()
```

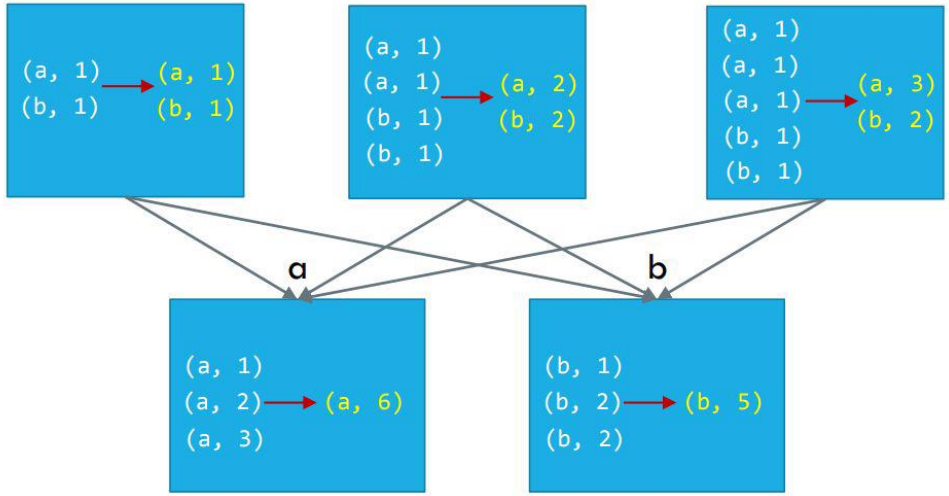
```
[(4, 50), (1, 8), (3, 22)]
```

groupByKey() VS reduceByKey()

groupByKey()



reduceByKey()



groupByKey() VS reduceByKey()

```
rdd1 = spark.sparkContext.parallelize ([(1,1),(4,1)])
rdd2 = spark.sparkContext.parallelize ([(1,1),(1,1),(4,1),(4,1)])
rdd3 = spark.sparkContext.parallelize ([(1,1),(1,1),(4,1),(4,1),(4,1),(4,1)])
rdd4=rdd1.union(rdd2).union(rdd3)
rdd4.reduceByKey(lambda x,y: x+y).collect()
```

```
[(1, 5), (4, 7)]
```

```
rdd1 = spark.sparkContext.parallelize ([(1,1),(4,1)])
rdd2 = spark.sparkContext.parallelize ([(1,1),(1,1),(4,1),(4,1)])
rdd3 = spark.sparkContext.parallelize ([(1,1),(1,1),(4,1),(4,1),(4,1),(4,1)])
rdd4=rdd1.union(rdd2).union(rdd3)
rdd4.groupByKey().mapValues(sum).collect()
```

```
[(1, 5), (4, 7)]
```

union()

Renvoie un nouveau RDD qui contient les éléments des deux RDDs originaux.

```
rdd1 = spark.sparkContext.parallelize([1, 2, 3, 4])
rdd2 = spark.sparkContext.parallelize([4,5, 6, 7])
rdd3=rdd1.union(rdd2)
rdd3.collect()
```

```
[1, 2, 3, 4, 4, 5, 6, 7]
```

join()

Renvoie un nouveau RDD contenant toutes les paires d'éléments ayants la même clé dans les RDDs d'origine.

```
x = spark.sparkContext.parallelize([("a", 1), ("b", 2)])
y = spark.sparkContext.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
z.collect()
```

```
[('b', (2, 5)), ('a', (1, 3)), ('a', (1, 4))]
```

sample(withReplacement, fraction, seed=None)

Renvoie un nouveau RDD qui contient un échantillon du RDD original.

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
rdd2=rdd.sample(withReplacement= True, fraction=0.5)
rdd2.collect()
```

```
[2, 2, 4, 4]
```

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
rdd2=rdd.sample(withReplacement= False, fraction=0.25)
rdd2.collect()
```

```
[3, 5]
```

sortByKey()

Renvoie un nouveau RDD trié sachant que le RDD original est sous la forme de paire (clé, valeur).

```
x = spark.sparkContext.parallelize([("b", 5), ("d", 4), ("a", 3), ('c',10) ])
x.sortByKey().collect()
```

```
[('a', 3), ('b', 5), ('c', 10), ('d', 4)]
```

sortBy()

Renvoie un nouveau RDD trié selon la fonction spécifiée.

```
x = spark.sparkContext.parallelize([("b", 5), ("d", 4), ("a", 3), ("3",15), ('c',10), ('1',20) ])
x.sortBy(lambda x: x[1]).collect()
```

```
[('a', 3), ('d', 4), ('b', 5), ('c', 10), ('3', 15), ('1', 20)]
```