Big Data

Big Data, quand un ensemble de données trop volumineuses, complexes et dynamique pour être gérer par des outils conventionnels.
- **Volume**: small, 10gb < medium < 1tb, big
- **Vitesse**: batch ou stream (flux) processing
- **Variété**: structurée, semi-struct, non struct
- **Véracité**: pré-traitement, qualité important

MapReduce

MapReduce manipuler grandes quantités de données en distribuant dans un cluster. **Apache Hadoop** permet de le faire. Utilise une architecture **maître-esclave**.

Avec MapReduce on déplace le traitement vers les données

Résultats partiels collectés et agrégés

Lire données paritionnées → **Map** extraire une partie → mélanger et trier (par le système) → **Reduce** agréger, résumer, filter, transformer → résultats

- Mapper: identifie les mots, sortie clé-val
- Shuffle and sort
- Reducer: lit le mot et le compte

Utilisateur choisi fonction de **Map**, fonction de **Reduce**, règle de shuffle, etc… Possible de **chaîner** des MapReduce

Spark

Apache Spark: framework open source combinant **infra distribuée de programmes** et **modèle pour écrire prog**

Spark consèrve seulement le graphe des transformations appliquées sur les données pour pas devoir sauvegarder tout le temps

Spark est construit autour du concept de RDD (données distribué résilient)

RDD

- **Resilient**: tolérence aux pannes
- **Distributed**: données sur plusieurs noeuds
- **Dataset**: données partit., valeur primitives

Utilisation

Initialiser session Spark
```
val spark: SparkSession =
SparkSession.builder
    .appName("Test")
    .master("local[*]")
    .getOrCreate()
val sc: SparkContext =
spark.sparkContext
```

Lire un fichier
```
val distFile = sc.textFile("link")
```

Résultat de type `RDD[String]`

Transformations et actions

- **Tranformation**: renvoient des nouveaux RDD en tant que résultat
  - Lazy → pas calculée immédiatement
  - `map`, `filter`,`flatMap`, `groupBy`, `sortBy`
- **Actions**: calculent un résultat basé sur un RDD et le renvoient ou enregistre dans un stockage (HDFS,…)
  - Eager → immédiatement calculée
  - `reduce`, `collect`, `foreach`, `count`, `max`

map

```
val x = sc.parallelize(List("a",...))
val y = x.map( z => (z,1))
```

`x` devient un `RDD[String]` avec les éléments
`y` devient un `RDD[(String, Int)]` donc on crée un **nouveau RDD transformé**

filter

```
val x = sc.parallelize(List(1,2,3))
val y = x.filter(n => n%2 == 1)
```

Ici le type ne change pas mais le RDD `y` contiendra des données en moins.

flatMap

```
val x = sc.parallelize(List(1,2,3))
val y = x.flatMap(n => List(n,
n*100))
```

Retournera `[1,100,2,200,3,300]` de manière applatie et pas `[[1,100],[2,200],[3,300]]`

groupBy

```
val x = sc.parallelize(List("John",
"Fred", "Anna", "James"))
val y = x.groupBy(w => w.charAt(0))
```

Regroupera les personnes par leur première lettre du prénom et donnera `(J,Seq(John, James)),(F,Seq(Fred)),(A,Seq(Anna))`

Autres

- `distinct()`: renvoie un nouvel ensemble sans les doublons
- `union()`: union de deux RDD
- `intersection()`: intersection de deux RDD

reduce

```
val wordsRdd =
sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(x =>
x.length)
val totalChars =
lengthsRdd.reduce((x,y)=> x+y)
```

`totalChars` contiendra la somme des longueurs de tous les mots

collect

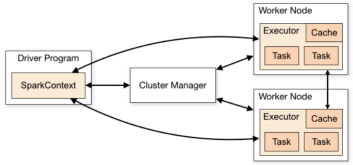
```
val x = sc.parallelize(List(1,2,3))
val y = x.collect()
```

`y` sera un `Array[Int]` et collect retournera `[1,2,3]`

Autres

- `take(n)`: retourne les `n` premiers éléments
- `first()`: retourne le premier élément
- `count()`: nombre d'éléments dans le RDD
- `foreach(f)`: applique la fonction `f` à chaque élément du RDD
- `saveAsTextFile(path)`: sauvegarde le RDD dans un fichier texte à l'emplacement spécifié

Execution de prog Spark



- Driver programme: crée contexte Spark, RDD, transformations, actions et récupère résultats
- Cluster manager: gère ressources du cluster
- Workers: exécutent tâches sur données

persist

Le mot clé `persist` permet de stocker un RDD en mémoire pour éviter de le recalculer à chaque fois.