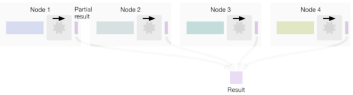
Big Data

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

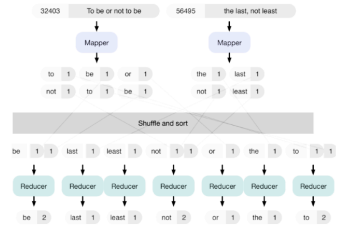
MapReduce manipuler grandes quantités de données en distribuant dans un cluster. **Apache Hadoop** permet de le faire. Utilise une architecture **maître-esclave**.

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

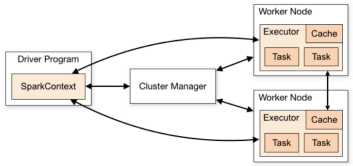
collect
```
val x = sc.parallelize(List(1,2,3))
val y = x.collect()
```

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.

PairRDDs

Un **Pair RDD** est un RDD où chaque élément est une **paire clé-valeur** `(K, V)`.
- Permet des opérations spécifiques : `groupByKey`, `reduceByKey`, `join`, etc.
- Utilisé pour des **agrégations**, **regroupements**, ou **jointures**.

Création d'un Pair RDD
```
val lines = List("one", "two", "two")
val linesRDD = sc.parallelize(lines)
val kvRDD = linesRDD.map(x=>(x,1))
// RDD[(String, Int)]
```

groupByKey

Regroupe les valeurs par clé et retourne un `RDD[(K, Iterable[V])]`.
- **Coûteux** : nécessite un **shuffle** (transfert réseau).
```
val data = Array((1, 2), (3, 4), (3, 6))
val myRdd = sc.parallelize(data)
val groupedRdd = myRdd.groupByKey()
// Résultat : (1, Seq(2)), (3, Seq(4, 6))
```

reduceByKey

Agrège les valeurs par clé avec une fonction associative (ex: `+`, `max`).
- **Plus efficace** que `groupByKey` : réduit les données avant le **shuffle**.
```
val eventsRdd =
sc.parallelize(List(("HEIG-VD", 42000),
("HEIG-VD", 14000)))
val totalBudget = eventsRdd.reduceByKey(_
+ _)
// Résultat : ("HEIG-VD", 56000)
```

mapValues

Applique une fonction **uniquement aux valeurs** d'un Pair RDD.
```
val rdd = sc.parallelize(List(("a", 1),
("b", 2)))
val mappedRdd = rdd.mapValues(v => v * 2)
// Résultat : (("a", 2), ("b", 4))
```

countByKey

Compte le nombre d'éléments par clé et retourne un **Map[K, Long]**.
```
val rdd = sc.parallelize(List(("a", 1),
("b", 2), ("a", 3)))
val counts = rdd.countByKey()
// Résultat : Map("a" -> 2, "b" -> 1)
```

join

Combine deux Pair RDDs sur leurs clés communes.
```
val abos = sc.parallelize(List((101,
"AG"), (102, "DemiTarif")))
val locations = sc.parallelize(List((101,
"Bern"), (102, "Lausanne")))
val joinedRdd = abos.join(locations)
// Résultat : (101, ("AG", "Bern")), (102,
("DemiTarif", "Lausanne"))
```

Optimisations : reduceByKey vs groupByKey

- **`groupByKey`** :
  - Transfère **toutes les valeurs** d'une clé sur un seul nœud (**shuffle** coûteux).
- **`reduceByKey`** :
  - **Réduit les données localement** avant le **shuffle** (moins de données transférées).
  - **3x plus rapide** que `groupByKey` pour les agrégations.

Exemple : Budget moyen par organisateur
```
val eventsKvRdd = eventsRdd.map(event =>
(event.organizer, event.budget))
val intermediate = eventsKvRdd.mapValues(b
=> (b, 1))
  .reduceByKey((v1, v2) => (v1._1 + v2._1,
v1._2 + v2._2))
val avgBudgets = intermediate.mapValues
{ case (budget, count) => budget / count }
```

Word Count (exemple classique)
```
val rdd = spark.textFile("hdfs://...")
val counts = rdd.flatMap(line =>
line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```

Spark DataFrames/Spark SQL

Spark SQL permet de manipuler des données **structurées** avec une API relationnelle.
- **3 APIs principales** : SQL literals, DataFrames, Datasets.
- **Optimisé** : Utilise le Catalyst Optimizer pour des performances élevées.

Spark SQL

- **Module Spark** pour le traitement de données structurées.
- **Sources de données** : JDBC, fichiers JSON/CSV, Hive, etc.
- **Intégration** : Fonctionne au-dessus des RDDs.

DataFrames

- **Tableau distribué** avec un **schéma** (comme une table SQL).
- **Optimisé** : Utilise le Catalyst Optimizer pour les requêtes.
- **Création** : À partir d'un RDD ou de fichiers (JSON, CSV, etc.).

Création de DataFrames

À partir d'un RDD (schéma inféré)
```
val tupleSeq = Seq((100, "Braga", "Porto",
"Portugal"))
import spark.implicits._
val tupleRDD = sc.parallelize(tupleSeq)
val tupleDF = tupleRDD.toDF("id", "name",
"city", "country")
```

À partir d'une case class (schéma inféré)
```
case class Person(id: Int, name: String,
city: String, country: String)
val caseSeq = Seq(Person(100, "Braga",
"Porto", "Portugal"))
val caseRDD = sc.parallelize(caseSeq)
val caseDF = caseRDD.toDF
```

Avec schéma explicite
```
val schemaString = "id name city country"
val fields = schemaString.split("
").map(fieldName =>
  StructField(fieldName, StringType,
nullable = true))
val schema = StructType(fields)
val rowRDD =
peopleRDD.map(_.split(",")).map(attrs =>
Row(attrs: _*))
val peopleDF =
spark.createDataFrame(rowRDD, schema)
```

À partir de fichiers
```
val peopleDF = spark.read.json("tmp/
people.json")
```

Requêtes SQL

- **Créer une vue temporaire** :
```
peopleDF.createOrReplaceTempView("people")
```
- **Exécuter une requête SQL** :
```
val frenchDF = spark.sql("SELECT * FROM
people WHERE country = 'France'")
```

API DataFrame

- **Méthodes principales** : `select`, `where`, `groupBy`, `orderBy`, `join`.
```
val sydneyEmployeesDF =
employeeDF.select("id", "lname")
  .where(col("city") === "Sydney")
  .orderBy(col("id"))
```

Transformations DataFrames

- **`select`** : Sélectionne des colonnes.
- **`where`/`filter`** : Filtre les lignes.
- **`groupBy`** : Regroupe les données.
- **`orderBy`** : Trie les résultats.
```
employeeDF.filter(col("age") > 30).show()
```

Agrégations

- **`agg`** : Applique des fonctions d'agrégation (`sum`, `avg`, `count`, etc.).
```
val rankedDF =
postsDF.groupBy(col("subforum"))
  .agg(min(col("likes")) as "minimum",
max(col("likes")) as "maximum")
```

Jointures

- **Types** : `inner`, `left_outer`, `right_outer`, `full_outer`.
```
val trackedCustomersDF =
abosDF.join(locationsDF, abosDF("id") ===
locationsDF("id"))
```

Actions DataFrames

- **`collect`** : Récupère tous les résultats sous forme de tableau.
- **`show`** : Affiche les résultats sous forme de tableau.
- **`count`** : Compte le nombre de lignes.