Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Centre Universitaire de Mila
Institut des Mathématiques et Informatique
Département de l'Informatique

# Big Data

## – Chapitre 3 –

# Apache Spark

# Plan du chapitre 3

- **Présentation de Spark**
- **RDD- Concepts et opérations**
- **Exécution**
- **Exemples de programmation**
- **Spark MLlib**

# Présentation (1)


- **Framework de conception et d'exécution Map/Reduce.**

- **Originellement (2014) un projet de l'université de Berkeley en Californie, désormais un logiciel libre de la fondation Apache.**

- **Licence Apache.**

- **Sensiblement plus rapide que Hadoop, notamment pour les tâches impliquant de multiples Maps et/ou Reduce.**

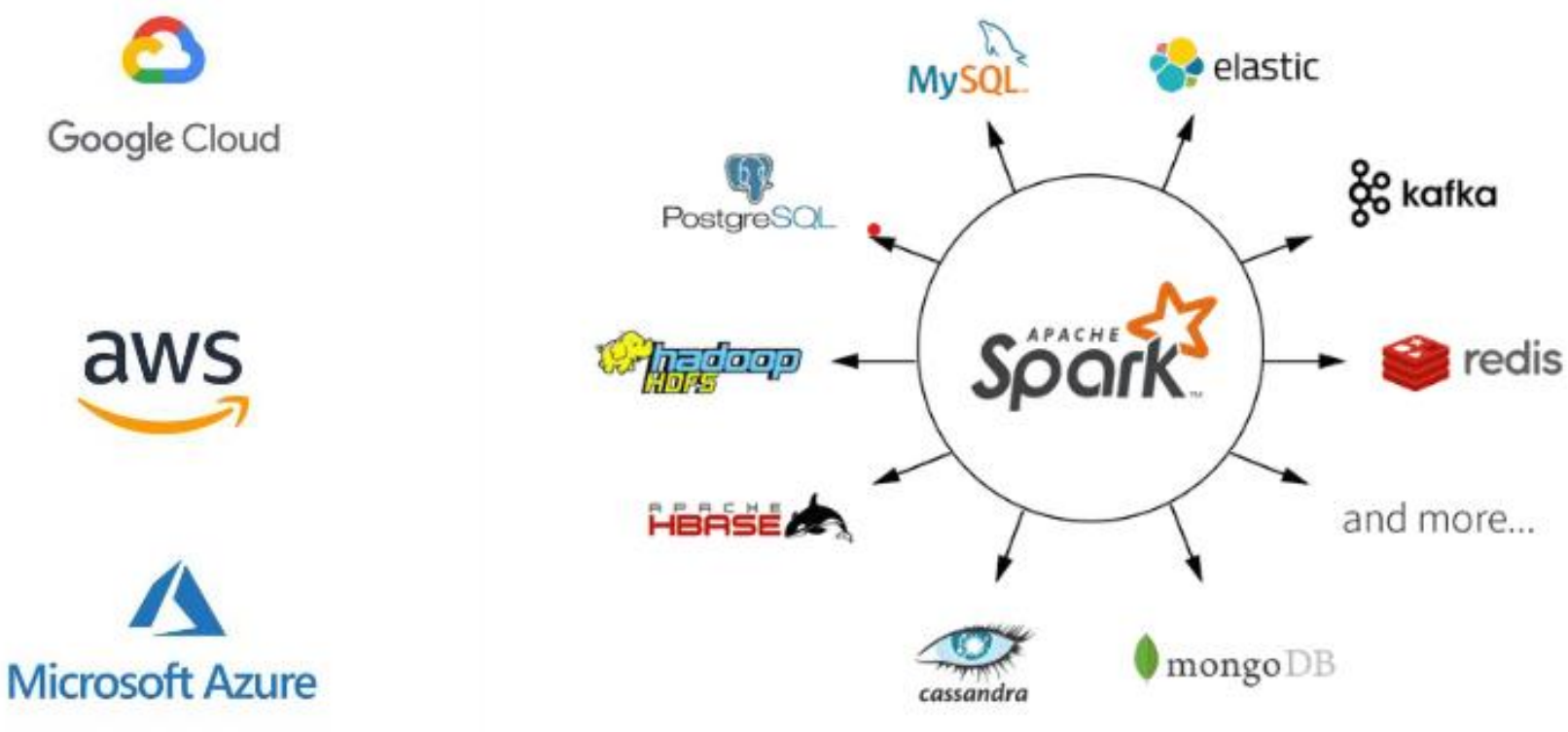
# Présentation (2)

**Spark** a été conçu :

- Pour exécuter efficacement des applications **itératives** et **interactives**
  - en conservant les données **en mémoire** entre les **opérations**
- Pour fournir un mécanisme de **tolérance aux pannes** à faible coût
  - Une faible **surcharge** lors des exécutions **sécurisées**
  - Une **récupération** rapide après une **panne**
- Pour être **simple** et **rapide** à utiliser dans un environnement **interactif**
  - En utilisant un langage de programmation **Scala** compact
- Pour être « **Scalable**»
  - Capable de traiter efficacement des données plus **volumineuses** sur des **clusters** de **calcul** plus importants

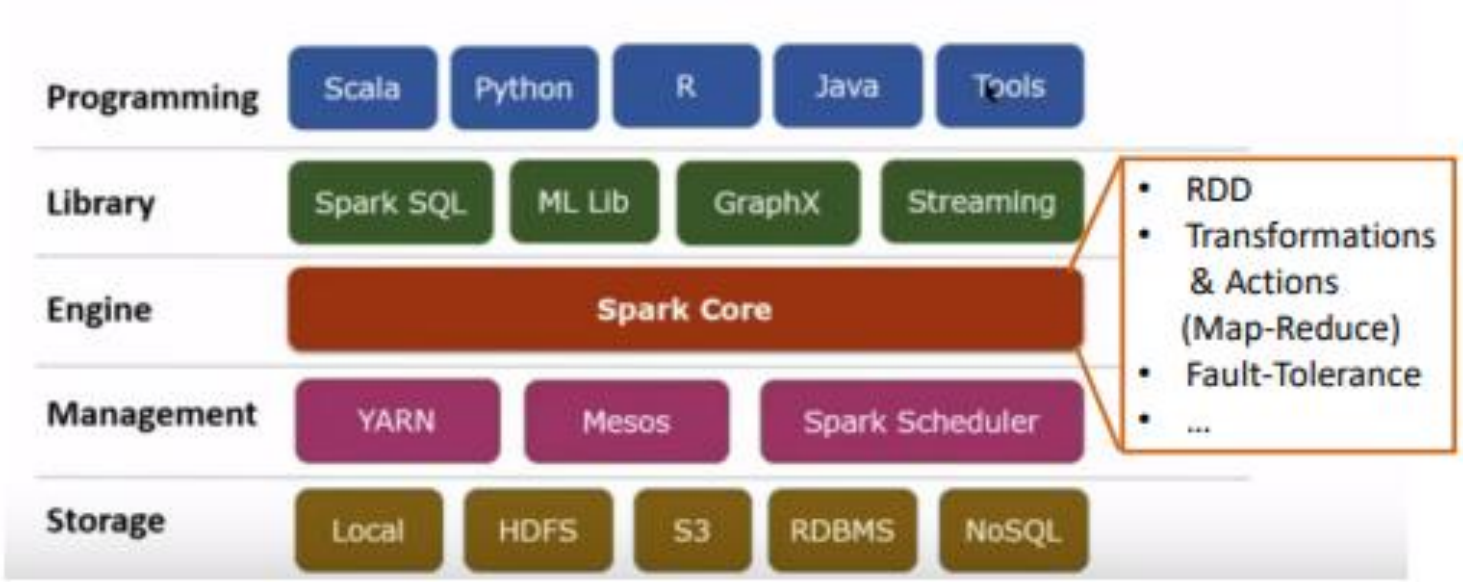
# Présentation (3)

**Spark** est connecté avec la majorité des technologies big data:



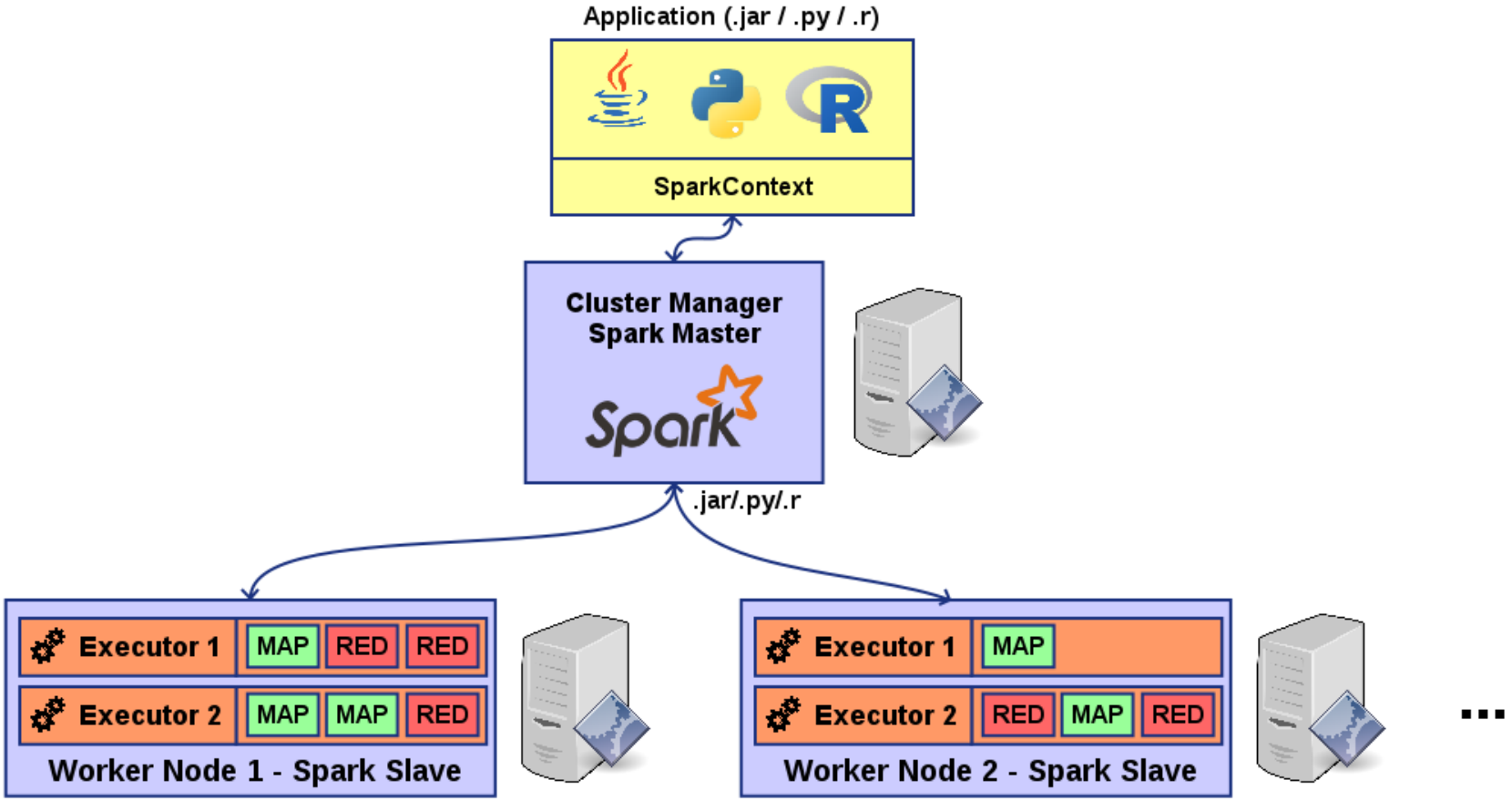
# Présentation (4)

● **Développé en Scala (langage orienté objet dérivé de Java et incluant de nombreux aspects des langages fonctionnels).**



# Architecture – Haut niveau

**En mode Spark natif : Les noeuds « Worker » tournent en permanence.**
**Ils disposent tous d'un cache.**



# RDD- Concepts et opérations (1)

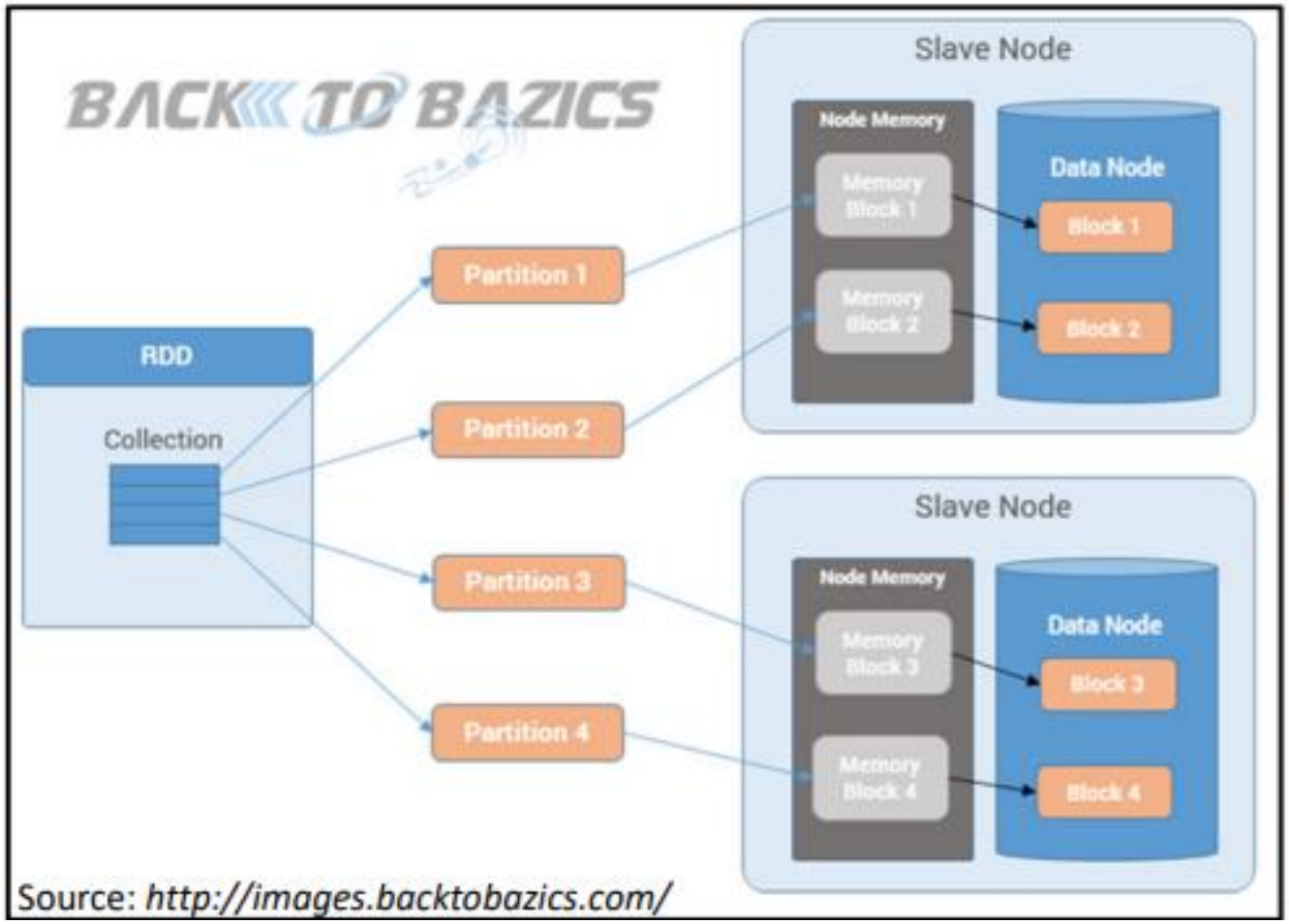
- Au centre du paradigme employé par **Spark**, on trouve la notion de **RDD**, pour **R**esilient **D**istributed **D**atasets.

- Il s'agit de larges ***hashmaps*** stockées en **mémoire** et sur lesquelles on peut appliquer des traitements.

- Ils sont:
  - **Distribués**.
  - **Partitionnés** (pour permettre à plusieurs noeuds de traiter les données).
  - **Redondés** (limite le risque de perte de données).
  - En **lecture seule**; un traitement appliqué à un **RDD** donne lieu à la création d'un nouveau **RDD**.

# RDD- Concepts et opérations (2)

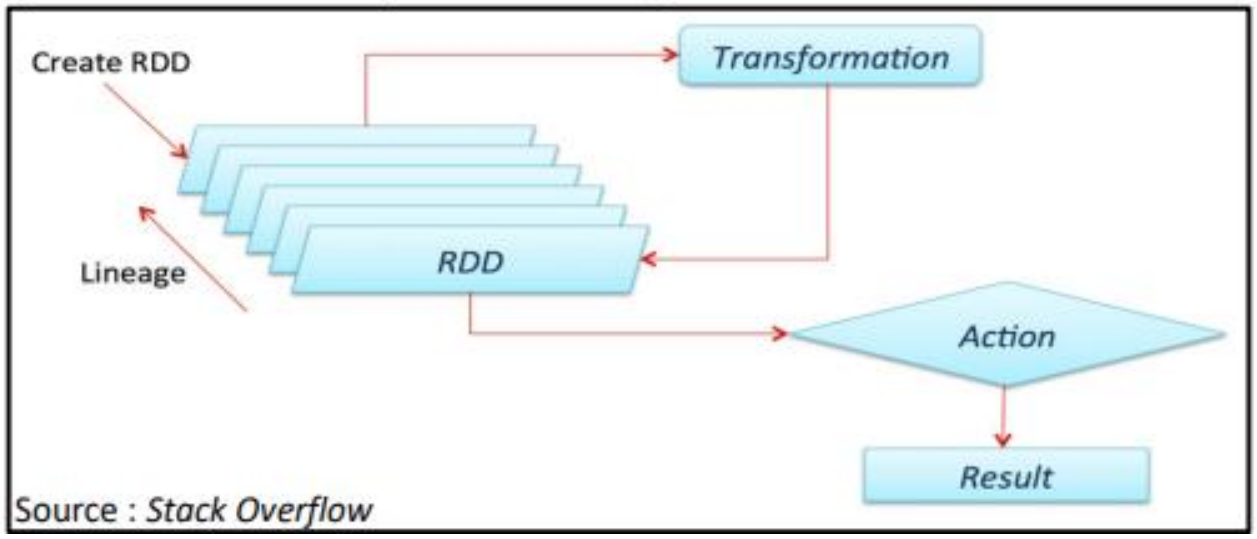
- Exemple de **blocs** de **4 partitions** stockés sur **2 nœuds** de données (pas de réplication)



# RDD- Concepts et opérations (3)

- Deux types d'opérations sur les **RDDs**:
  - Une **transformation**: une opération qui **modifie** les données d'un **RDD**. Elle donne lieu à la création d'un **nouveau RDD**.
    « **map** » est un exemple de transformation.

  - Une **action**: elles accèdent aux données d'un **RDD**, et nécessitent donc son évaluation,
    « **saveAsTextFile** » qui permet de **sauver** le contenu d'un RDD,



# RDD- Concepts et opérations (4)

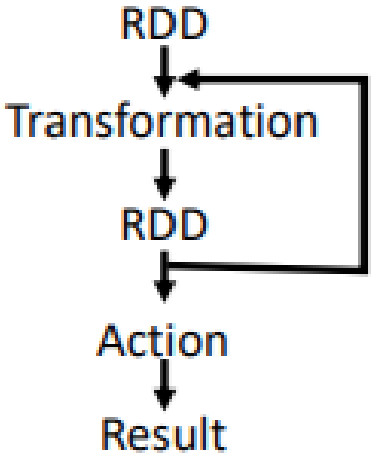
- Exemple d'**actions** et **transformations** sur les **RDDs**:

| | | | |
|---|---|---|---|
| **Transformations** | *map*(*f* : T ⇒ U) | : | RDD[T] ⇒ RDD[U] |
| | *filter*(*f* : T ⇒ Bool) | : | RDD[T] ⇒ RDD[T] |
| | *flatMap*(*f* : T ⇒ Seq[U]) | : | RDD[T] ⇒ RDD[U] |
| | *sample*(*fraction* : Float) | : | RDD[T] ⇒ RDD[T] (Deterministic sampling) |
| | *groupByKey*() | : | RDD[(K, V)] ⇒ RDD[(K, Seq[V])] |
| | *reduceByKey*(*f* : (V, V) ⇒ V) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *union*() | : | (RDD[T], RDD[T]) ⇒ RDD[T] |
| | *join*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))] |
| | *cogroup*() | : | (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))] |
| | *crossProduct*() | : | (RDD[T], RDD[U]) ⇒ RDD[(T, U)] |
| | *mapValues*(*f* : V ⇒ W) | : | RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning) |
| | *sort*(*c* : Comparator[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| | *partitionBy*(*p* : Partitioner[K]) | : | RDD[(K, V)] ⇒ RDD[(K, V)] |
| **Actions** | *count*() | : | RDD[T] ⇒ Long |
| | *collect*() | : | RDD[T] ⇒ Seq[T] |
| | *reduce*(*f* : (T, T) ⇒ T) | : | RDD[T] ⇒ T |
| | *lookup*(*k* : K) | : | RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs) |
| | *save*(*path* : String) | : | Outputs RDD to a storage system, *e.g.*, HDFS |

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

# Exécution (1)

- Les transformations sont des opérations **paresseuses** : enregistrées et exécutées **ultérieurement**,
- Les actions **déclenchent** l'exécution de la séquence de transformations
- Un job est une séquence de transformations **RDD**, terminée par une **action**




- Une application Spark est un ensemble de tâches (jobs) à exécuter ***séquentiellement*** ou en ***parallèle,***

# Exemple de compteur d'occurrence de mots

- Transformation:
- **lines = sc.textFile("bible.txt")**
- **words = lines.flatMap(lambda line: line.split(" "))**
- **items = words.map(lambda word: (word, 1))**
- **counts = items.reduceByKey(lambda a, b: a + b)**

- Action:
- **counts.take(5)**

# Exécution : Driver (2)

- Le **driver** d'application Spark **contrôle** l'exécution de l'application
  - Il crée le **contexte** Spark
  - Il analyse le **programme** Spark

  - Il **crée** un **DAG (Directed Acyclic Graph)** de tâches pour chaque **Job** (**représentant les *actions* et *opérations* effectuées dans le programme)**
  - Il optimise le **DAG**
    - Pipeline de transformations **étroites**
    - Identifie les **tâches** qui peuvent être exécutées **en parallèle**

- Il planifie le **DAG** des tâches sur les nœuds de travail disponibles (*Spark Executors*) afin de maximiser le **parallélisme** (et de **réduire** le temps d'exécution)

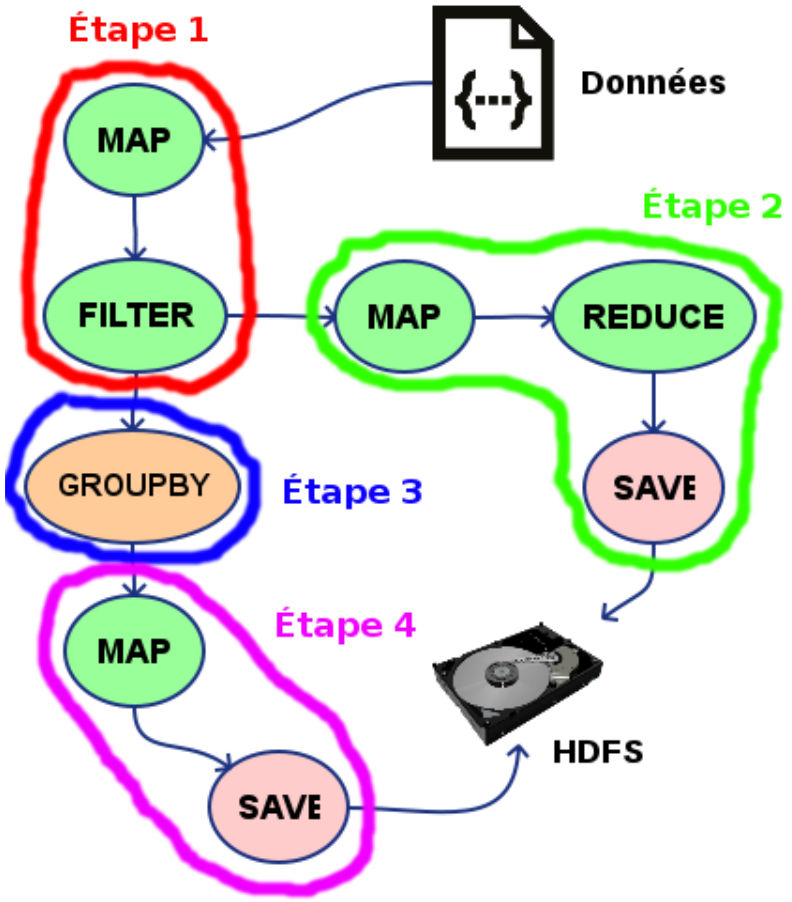
- Un **RDD** obtenu à la fin d'une **transformation** peut être explicitement conservé en **mémoire**, lors de l'appel de la méthode **persist()** de ce **RDD** (intéressant s'il est **réutilisé** ultérieurement).

# Exécution (3)

**Exemple :**

- Ici, on imagine qu'on a **persisté** le **RDD** en sortie de « **filter** »; le découpage est par conséquent différent.

- A noter que les **opérations** « **larges** » (comme **groupBy**) sont plus **coûteuses**, parce qu'elles nécessitent de réorganiser (repartitioner) les données entre les **noeuds** d'exécution, ce qui peut impliquer un certain nombre d'échanges.

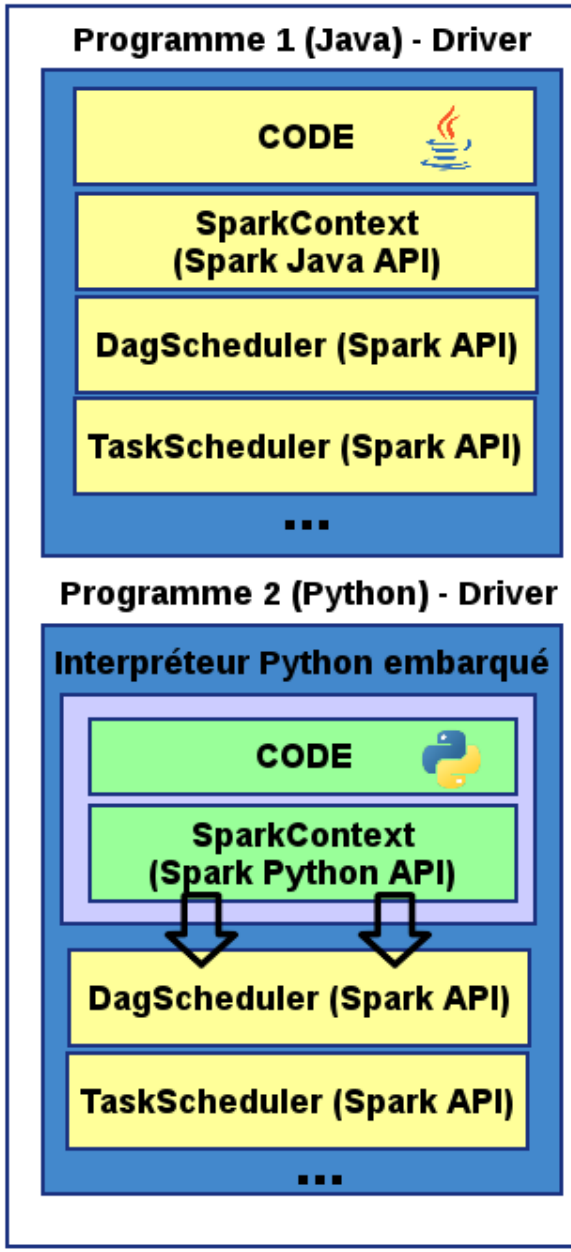


# Exécution (4)

- Un programme **Spark** fait appel à un objet, le **SparkContext**, qui permet d'utiliser l'**API Spark** et de communiquer avec un **cluster** Spark

- On parle de ***driver*** pour désigner l'ensemble composé du programme, du **SparkContext**, et de plusieurs autres composants, parmi lesquels:

  - Le **DagScheduler**, qui est en charge de construire les **graphes** acycliques à partir du code du programme et de les **découper** en **tâche**.
  - Le **TaskScheduler**, qui **lance l'exécution** des tâches, **surveille** qu'elles s'exécutent bien, et **relancent** celles qui ont posé **problème** au besoin.



# Exemples de programmation

**Ex. of transformations on one RDD:**

rdd : {1, 2, 3, 3}

Python: `rdd.map(lambda x: x+1)` → rdd: {2, 3, 4, 4}

Scala : `rdd.map(x => x+1)` → rdd: {2, 3, 4, 4}

Scala : `rdd.map(x => x.to(3))` → rdd: {(1,2,3), (2,3), (3), (3)}

Scala : `rdd.flatMap(x => x.to(3))` → rdd: {1, 2, 3, 2, 3, 3, 3}

Scala : `rdd.filter(x => x != 1)` → rdd: {2, 3, 3}

Scala : `rdd.distinct()` → rdd: {1, 2, 3}

Some sampling functions exist:

Scala : `rdd.sample(false,0.5)` → rdd: {1} or {2,3} or …

*with replacement = false*

**Sequence of transformations:**

Scala: `rdd.filter(x => x != 1).map(x => x+1)` → rdd: {3, 4, 4}

# Exemples de programmation

**Ex. of transformations on two RDDs:**

rdd : {1, 2, 3}
rdd2: {3, 4, 5}

Scala : `rdd.union(rdd2)` → rdd: {1, 2, 3, 3, 4, 5}

Scala : `rdd.intersection(rdd2)` → rdd: {3}

Scala : `rdd.subtract(rdd2)` → rdd: {1, 2}

Scala : `rdd.cartesian(rdd2)` → rdd: {(1,3), (1,4), (1,5),
(2,3), (2,4), (2,5),
(3,3), (3,4), (3,5)}

# Exemples de programmation

## Ex. of actions on a RDD:

Examples of « aggregations »: **computing a sum**

rdd : {1, 2, 3, 3}

Computing the sum of the RDD values:

Python : `rdd.reduce(lambda x,y: x+y)` → 9

Scala : `rdd.reduce((x,y) => x+y)` → 9

**Results are NOT RDD**

Specifying the initial value of the accumulator:

Scala : `rdd.fold(0)((accu,value) => accu+value)` → 9

Specifying to start to accumulate from Left or from Right:

Scala : `rdd.foldLeft(0)((accu,value) => accu+value)` → 9

Scala : `rdd.foldRight(0)((accu,value) => accu+value)` → 9

# Exemples de programmation

**Ex. of actions on a RDD:**

rdd : {1, 2, 3, 3}

Scala : `rdd.collect()` → {1, 2, 3, 3}

Scala : `rdd.count()` → 4

Scala : `rdd.countByValue()` → {(1,1), (2,1), (3,2)}

Scala : `rdd.take(2)` → {1, 2}

Scala : `rdd.top(2)` → {3, 3}

Scala : `rdd.takeOrdered(3,Ordering[Int].reverse)` → {3,3,2}

Scala : `rdd.takeSample(false,2)` → {?,?}

*takeSample(withReplacement, NbEltToGet, [seed])*

Scala :
```
var sum = 0
rdd.foreach(sum += _)
println(sum)
```
→ does not return any value

→ 9

# Exemples de programmation : pair RDDs

**Ex. of transformations on one RDD:**

rdd : {(1, 2), (3, 3), (3, 4)}

Scala : `rdd.reduceByKey((x,y) => x+y)` → rdd: {(1, 2), (3, 7)}

*Reduce values associated to the same key*

Scala : `rdd.groupByKey((x,y) => x+y)` → rdd: {(1, [2]), (3, [3, 4])}

*Group values associated to the same key*

Scala : `rdd.mapValues(x => x+1)` → rdd: {(1, 3), (3, 4), (3, 5)}

*Apply to each value (keys do not change)*

Scala : `rdd.flatMapValues(x => x to 3)` → rdd: {(1,2), (1,3), (3,3)}

key: 1, 2 to 3 → (2, 3) → (1, 2), (1, 3),
key: 3, 3 to 3 → (3) → (3, 3)
key: 3, 4 to 3 → () → nothing

→ (1,2), (1,3), (3,3)

*Apply to each value (keys do not change) and flatten*

# Exemples de programmation : pair RDDs

**Ex. of transformations on one RDD:**

rdd : {(1, 2), (3, 3), (3, 4)}

Scala : `rdd.keys()` → rdd: {1, 3, 3}

*Return an RDD of just the keys*

Scala : `rdd.values()` → rdd: {2, 3, 4}

*Return an RDD of just the values*

Scala : `rdd.sortByKeys()` → rdd: {(1, 2), (3, 3), (3, 4)}

*Return a pair RDD sorted by the keys*

Scala :
```
rdd. combineByKey(
    …, // createCombiner function
    …, // mergeValue function          ≈ Hadoop Combiner
    …, // mergeCombiners function)    ≈ Hadoop Reduce
```

*Voir plus loin…*

# Exemples de programmation : pair RDDs

## Ex. of transformations on two pair RDDs

rdd : {(1, 2), (3, 4), (3, 6)}
rdd2: {(3, 9)}

Scala : `rdd.subtractByKey(rdd2)` → rdd: {(1, 2)}

*Remove pairs with key present in the 2nd pairRDD*

Scala : `rdd.join(rdd2)` → rdd: {(3, (4, 9)), (3, (6, 9))}

*Inner Join between the two pair RDDs*

Scala : `rdd.cogroup(rdd2)` → rdd: {(1, ([2], [])),
(3, ([4, 6], [9]))}

*Group data from both RDDs sharing the same key*

# Exemples de programmation : pair RDDs

## Ex. of classic transformations applied on a pair RDD

rdd : {(1, 2), (3, 4), (3, 6)}

A pair RDD remains a RDD of tuples (key, values)

→ Classic transformations can be applied

Scala : `rdd.filter{case (k,v) => v < 5}` → rdd: {(1, 2), (3, 4)}

Scala : `rdd.map{case (k,v) => (k,v*10)}` → rdd: {(1, 20),
(3, 40),
(3, 60)}

# Exemples de programmation : pair RDDs

## Ex. of actions on pair RDDs

rdd : {(1, 2), (3, 4), (3, 6)}

Scala : `rdd.countByKey()` → {(1, 1), (3, 2)}

*Return a tuple of couple, counting the number of pairs per key*

Scala : `rdd.collectAsMap()` → Map{(1, 2), (3, 4), (3, 6)}

*Return a 'Map' datastructure containing the RDD*

Scala : `rdd.lookup(3)` → [4, 6]

*Return an array containing all values associated with the provided key*

# Spark Mllib

- **Spark MLLib** est la librairie d'**apprentissage automatique** de Spark.
- Son but est de rendre son utilisation facile et **scalable**.
- Elle fournit des outils tel que:

  -Des algorithmes de **machine learning**

  -**L'extraction de caractéristiques** (*Features*), transformation, réduction de dimensions et sélection

  -Les **pipelines** pour construire, évaluer et régler les pipelines ML.

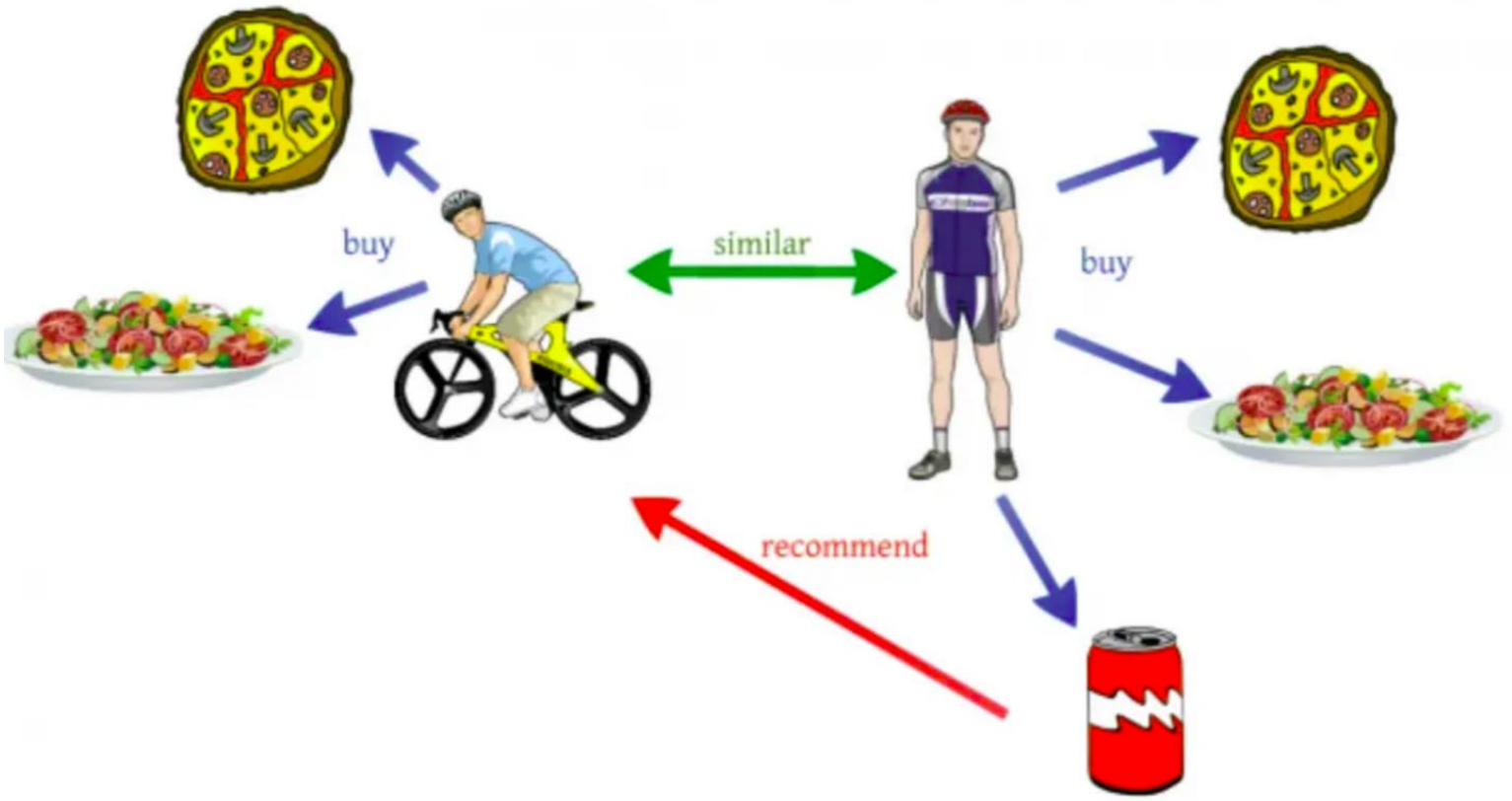
  -La **persistence**, pour sauvegarder et charger des algorithmes, modèles et pipelines.

  - Des **utilitaires** tel que l'**algèbre linéaire**, **statistiques**, manipulation des données, etc.

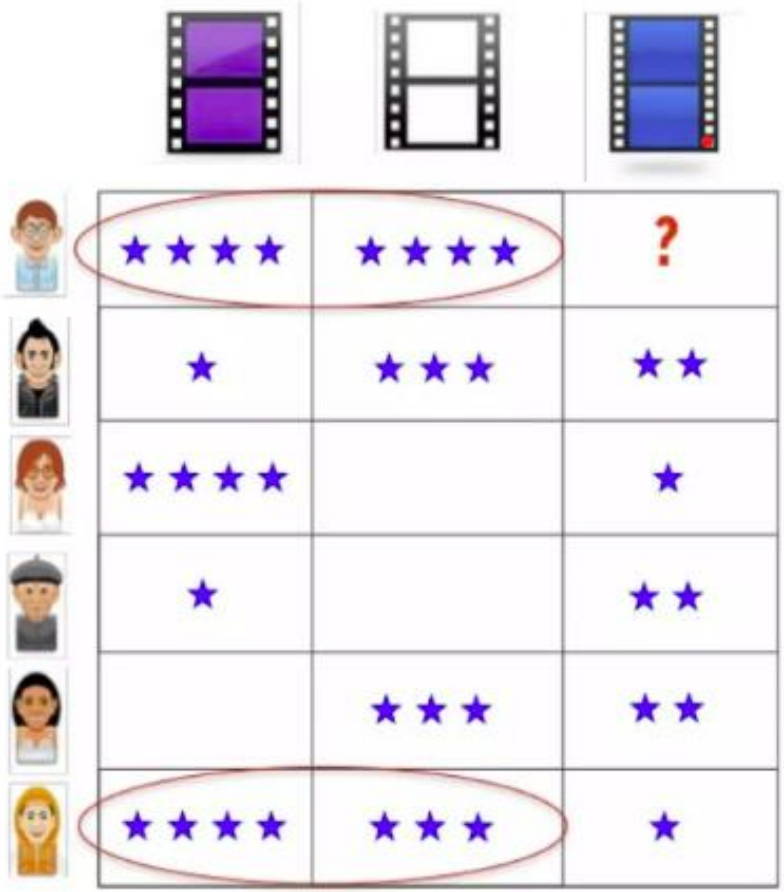
# Algorithmes d'apprentissage automatique dans Spark MLlib

- **Régression** :
  - Régression linéaire
  - Régression par arbres de décision
  - Régression par arbres à gradient renforcé
- **Classification :**
  - Régression logistique
  - Classification par forêts aléatoires
  - Machines à vecteurs de support (SVM)
  - Bayes naïf
- **Clustering** :
  - Clustering K-means
  - Allocation de Dirichlet latente (LDA)
- **Recommandation** :
  - Filtrage collaboratif

# Exemple sur la Recommandation des films : **Filtrage collaboratif**



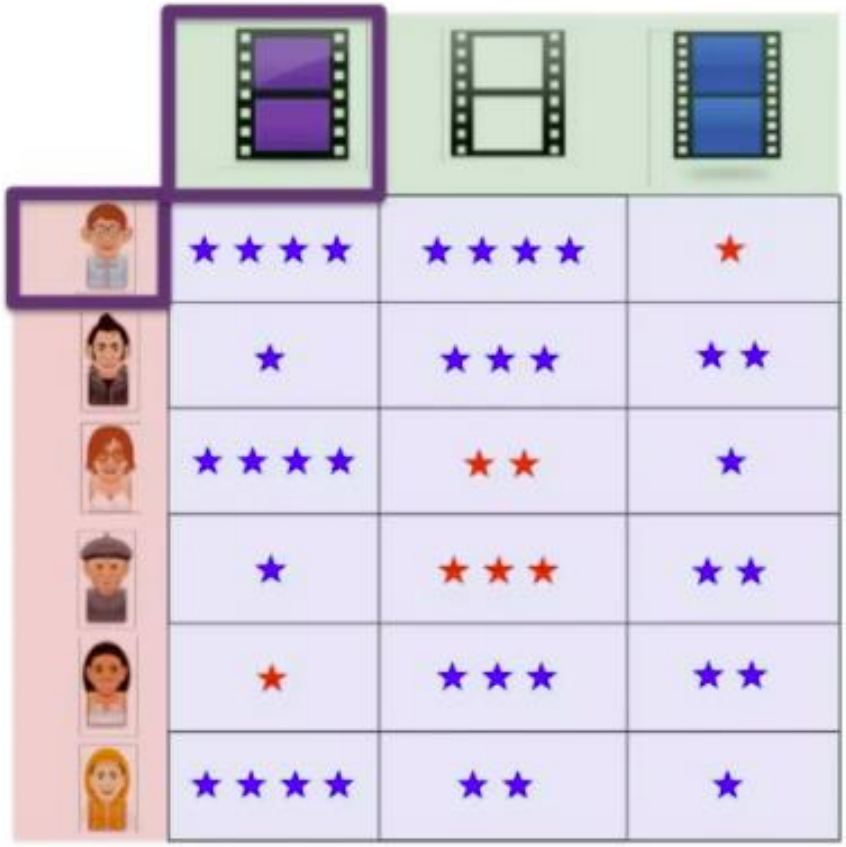
# Example sur la Recommandation des films : **Filtrage collaboratif**



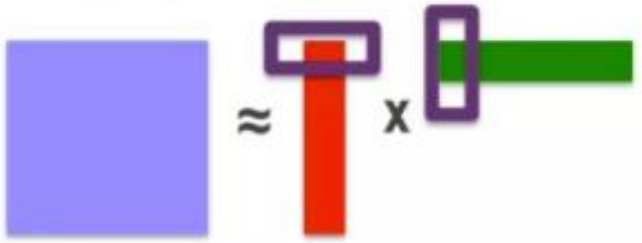

**Goal**: Recommend movies to users

NETFLIX

# Exemple sur la Recommandation des films : **Filtrage collaboratif**



Solution: Assume ratings are determined by a small number of factors.



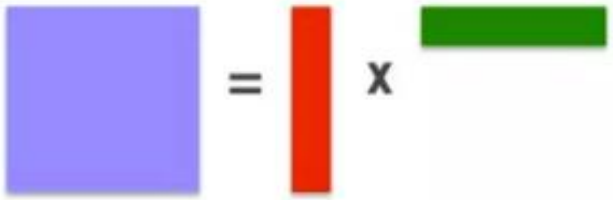
25M Users, 100K Movies
→ 2.5 trillion ratings
With 10 factors/user
→ 250M parameters

# Example sur la Recommandation des films : Filtrage collaboratif

## Recommendation: ALS

Algorithm

Alternating update of user/movie factors



**Can update factors in parallel**

**Must be careful about communication**

NETFLIX last·fm amazon.com P LinkedIn

# Spark Pipeline (1)

- Les **pipelines** permettent d'enchaîner plusieurs **Transformateurs** et **Estimateurs** dans un seul **flux de travail**.

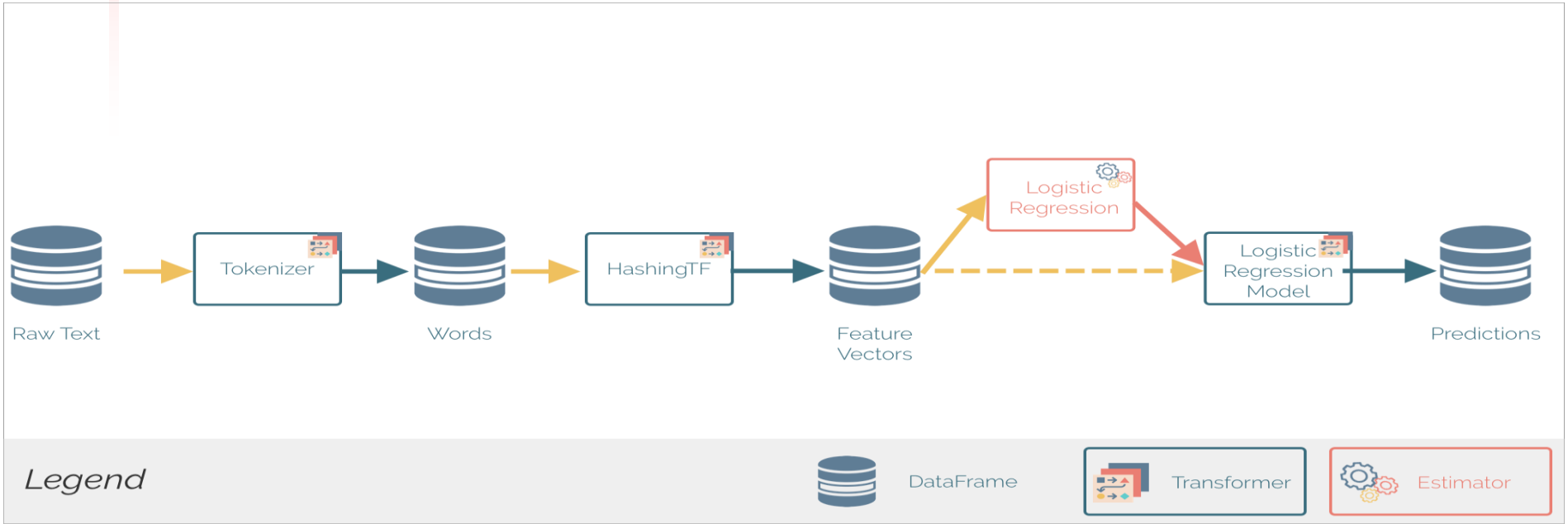
  - **Transformer**: Algorithme qui peut transformer un **DataFrame** en un autre **DataFrame**.
  - **Estimator**: Algorithme qui peut être appliqué sur un DataFrame pour produire un Transformer.

- Cela permet d'organiser et d'exécuter de manière séquentielle les **étapes** de **prétraitement** des données, d'**entrainement** des modèles et de **prédiction**

  **Par exemple**, un flux de traitement d'un document texte peut inclure les étapes suivantes:
  - Diviser chaque document en mots
  - Convertir chaque mot en vecteur de caractéristiques numériques
  - Créer un modèle prédictif en utiliser les vecteurs et les labels

# Spark Pipeline (2)

- Cette **pipeline** montre l'application d'un modèle de **régression linéaire** pour la prédiction de la **valeur** d'un **label** à partir d'un texte.
- La méthode **Pipeline.fit()** est initialement appelée sur le DataFrame originel (*Raw Text*), qui contient des documents textes bruts et des labels.
- La méthode **Tokenizer.transform()** divise les documents texte en mots, en rajoutant une nouvelle colonne contenant ces mots à la DataFrame.
- La méthode **HashingTF.transform()** convertit les mots en vecteurs de **features**.
- Ensuite, puisque **LogisticRegression** est un **Estimator**, la pipeline appelle **LogisticRegression.fit()** pour créer le Transformer: *LogisticRegressionModel*, qui est à son tour utilisé pour produire un nouveau DataFrame contenant les **prédictions**.



# RéférencesSpark Mllib

- http://www.metz.supelec.fr/metz/personnel/vialle/course/BigData-2A-CS/slides-pdf/
- https://cours.tokidev.fr/bigdata/