



**Adotmob**  
CONNECTING THE DOTS

Cours MS BGD : Spark

22 Septembre 2017

Nicolas Cosson  
Maxime Kubryk



# Présentation générale

# Adotmob

Qui sommes-nous ?

- Fondée en 2014
- Publicité ciblée achetée en RTB
- Présent à Paris, Londres, New York
- Croissance de 300% par an
- Investissement de 1M€ en 2015 puis 10M€ en 2016
- 45 employés dont 8 dans la data

## Quelques chiffres

- Volume :
  - ➔ Environ 10 To de données brutes par jour
  - ➔ Environ 10 milliards de requêtes par jour
- Infrastructure :
  - ➔ 100% sur AWS
  - ➔ Entre 100 et 400 serveurs
  - ➔ Temps de réponse inférieurs à 100ms

# Apache Spark

## Programme du cours

Cours :

- Généralités sur Spark
- Spark vs. MapReduce
- Composants :
  - Librairies
  - APIs
  - Spark UI
- Infrastructure :
  - Ecosystème
  - Cluster
  - Fault-Tolerance
- Spark Internals :
  - RDDs
  - Partitionnement
  - Actions/Transformations
  - Shuffling
  - Persistence

TP : introduction à Spark puis projet guidé (classification supervisée d'exoplanètes) :

- Introduction
  - Lancement d'un serveur Spark
  - Lancement d'un premier job et analyse de logs
- Projet
  - Récupération de données brutes
  - Traitement et formatage des données
  - Optimisation du script
  - Entraînement d'un algorithme de classification
  - Sauvegarde d'un modèle

Evaluation : projet en groupe

- Mise en place d'un job de classification en streaming

1

# Généralités sur Spark

# Qu'est-ce que Spark ?

Généralités sur Spark

- Outil de **processing** et **d'analyse de données** à grande échelle, codé en Scala, et open source.
- Différents modes de déploiement :
  - en single machine pour des tests ou pour traiter de petits échantillons de données
  - en **cluster** (multi-machines) pour traiter de gros volumes de données
- Fonctionne aisément sur un laptop en local (sans cluster manager)
- Accessible en plusieurs langages de programmation via des API **Scala**, Python, Java et R

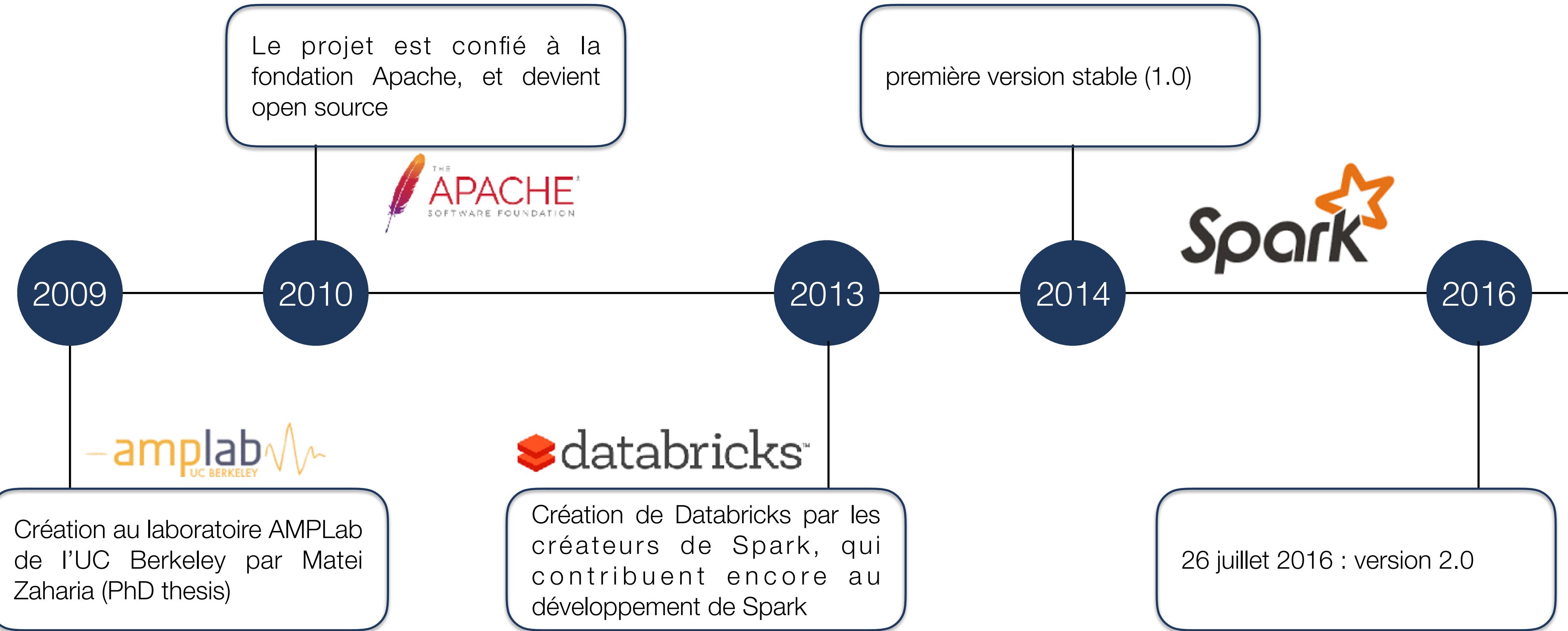
# Spark, outil polyvalent

## Généralités sur Spark

- Compatible avec la plupart des bases de données et systèmes de fichiers distribués (ou non)
  - **S3** (AWS), Google Storage
  - HDFS
  - Cassandra, HBase, Redshift,...
- Possède plusieurs librairies, à savoir :
  - un moteur **SQL**
  - du **Machine Learning**
  - du calcul sur graphes
  - du **streaming** (ou presque !)

# Historique

## Généralités sur Spark

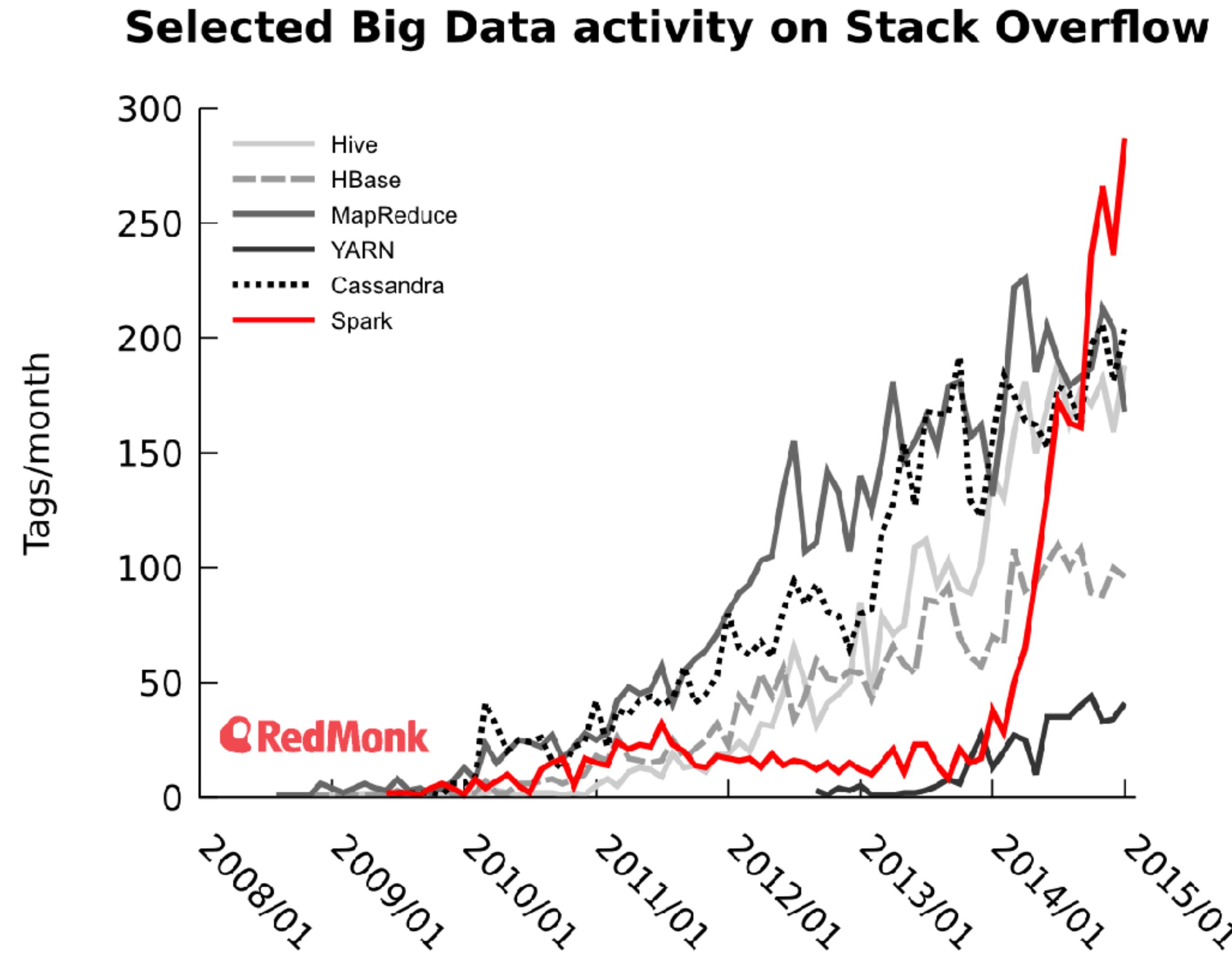


Source : <http://blog.madhukaraphatak.com/history-of-spark/>

# Spark largement adopté

Généralités sur Spark

- Projet Big Data le plus activement développé actuellement :
  - ~1000 contributeurs
  - ~100 commits/semaine
- Rapidement et largement adopté par la communauté Big Data
- Utilisé en **production** dans de nombreuses entreprises (Netflix, Uber, Spotify,...)
- Disponible en **SaaS** chez les cloud providers majeurs



# 1. Spark Adoption Is Growing Rapidly



Adoption of Spark has spread beyond the technology industry, and Spark is fast becoming the Big Data technology for everyone, not just for Big Data experts.



## SPARK IS THE MOST ACTIVE OPEN SOURCE PROJECT IN BIG DATA.

### Spark Summit conferences

\*Based on Spark Summit East and Spark Summit West,  
not including Spark Summit Europe

**1,164**  
attendees  
**453**  
companies

**2,986**  
attendees  
**1,144**  
companies

**2014**

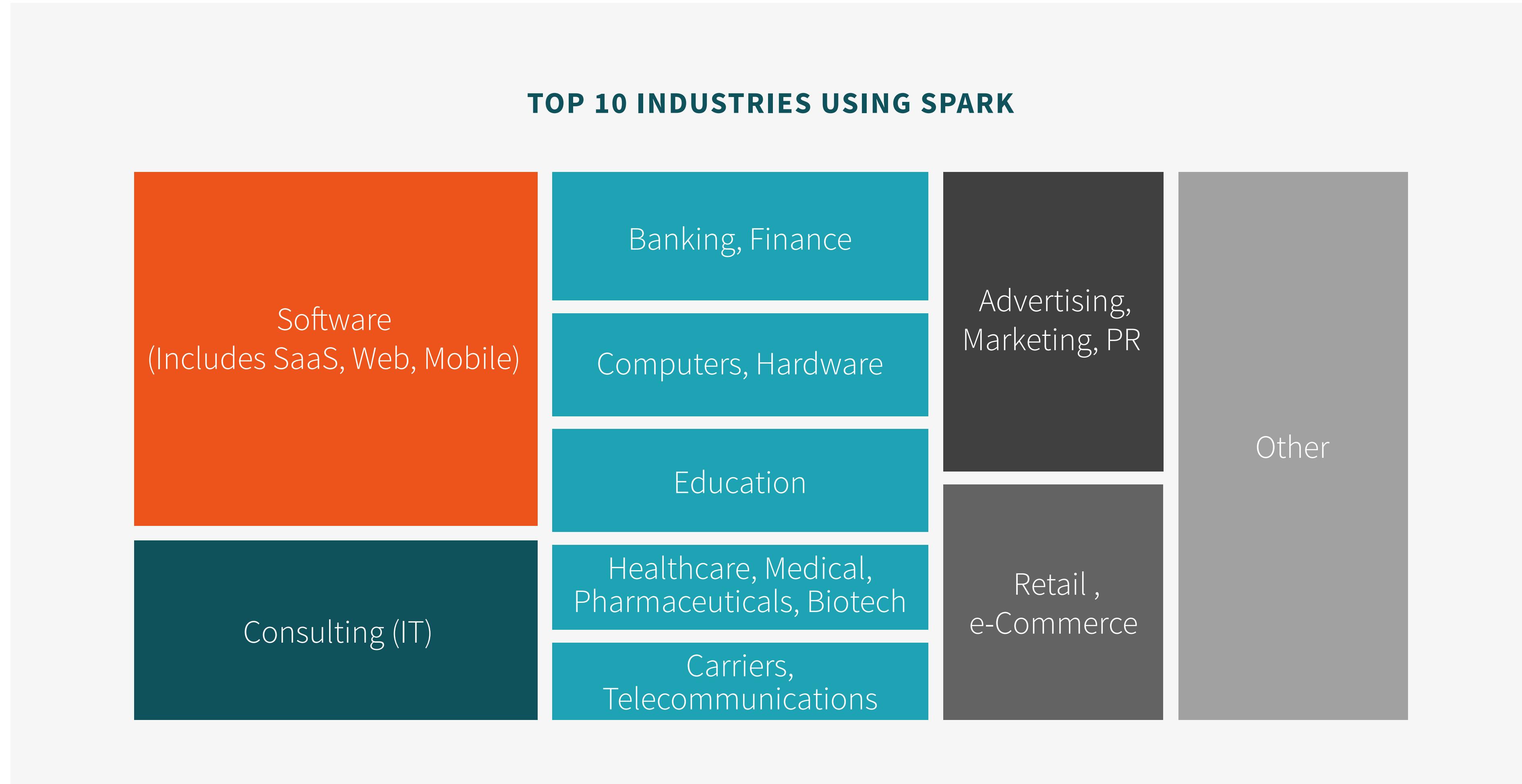
**2015\***

### Spark contributors

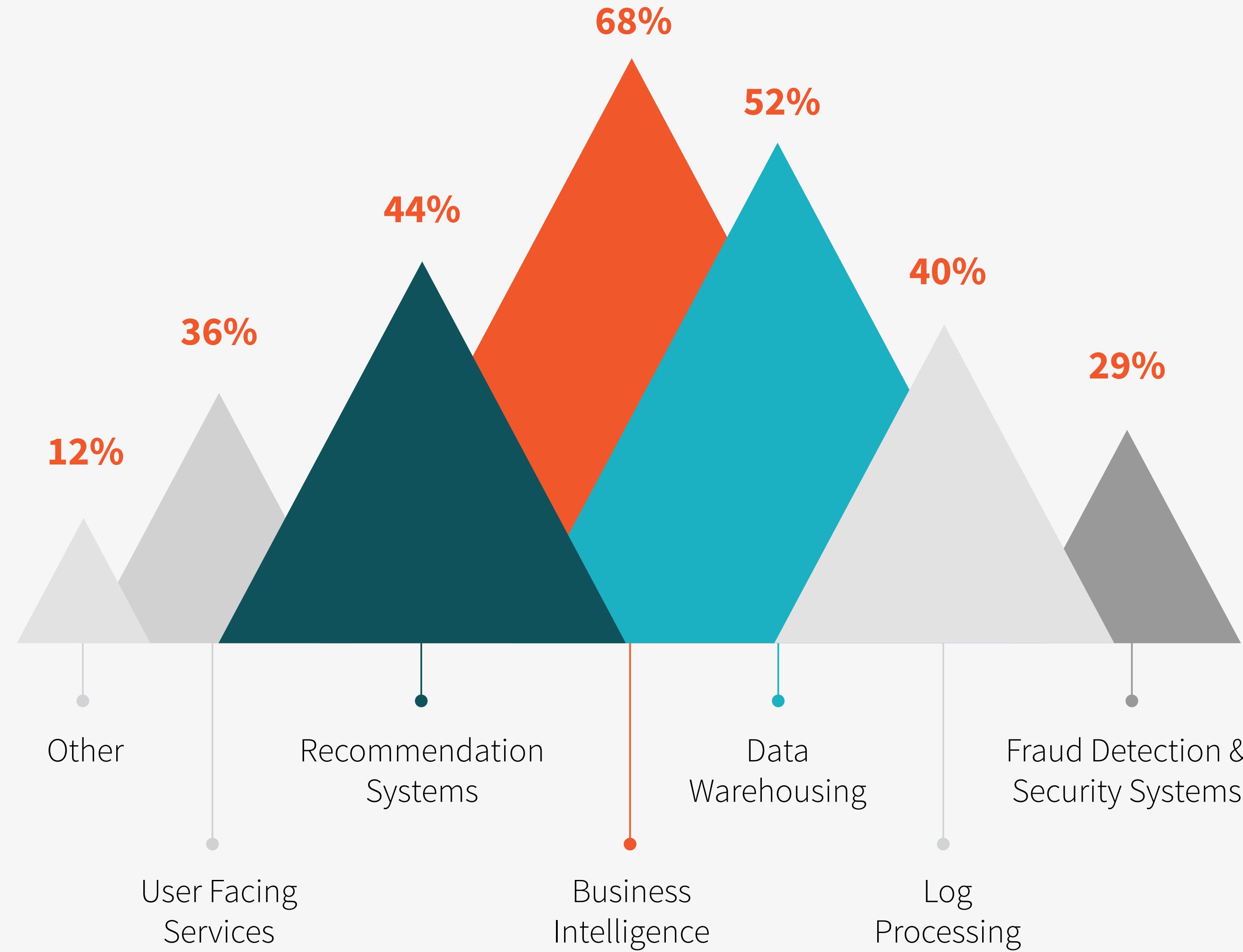


# Spark Survey 2015

## TOP 10 INDUSTRIES USING SPARK



## SPARK IS USED TO CREATE MANY TYPES OF PRODUCTS INSIDE OF DIFFERENT ORGANIZATIONS



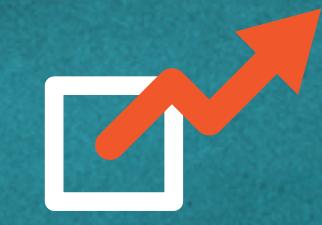
# Spark Survey 2015

## NOTABLE USERS THAT PRESENTED AT SPARK SUMMIT 2015 SAN FRANCISCO

Source: Slide 5 of Spark Community Update



## 2. Spark Is Growing Far Beyond Hadoop



While many users run Spark in on-premise Hadoop environments, they are not a majority of its users. Spark usage in the cloud and with Spark's own cluster manager have surged in the last year.



**MOST COMMON SPARK DEPLOYMENT ENVIRONMENTS (CLUSTER MANAGERS)**



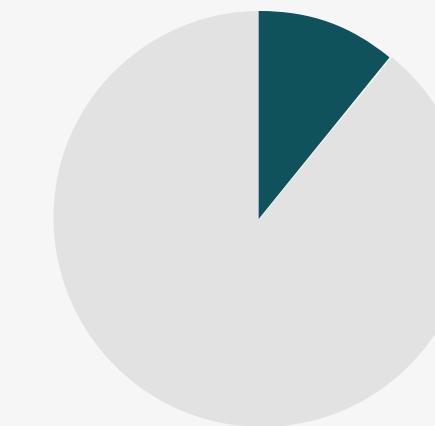
**48%**

Standalone mode



**40%**

YARN



**11%**

Mesos

**HOW RESPONDENTS ARE RUNNING SPARK**



**51%**

on a public cloud

## MOST USED SPARK COMPONENTS

**69%** | Spark SQL

**62%** | DataFrames

**58%** | MLlib + GraphX

**48%** | Streaming

**75%**

of Spark users are using two or more Spark components.

---

51% of Spark users are using three or more Spark components.

# Spark Survey 2015

## MOST IMPORTANT SPARK FEATURES

64%

Advanced analytics

47%

DataFrames

52%

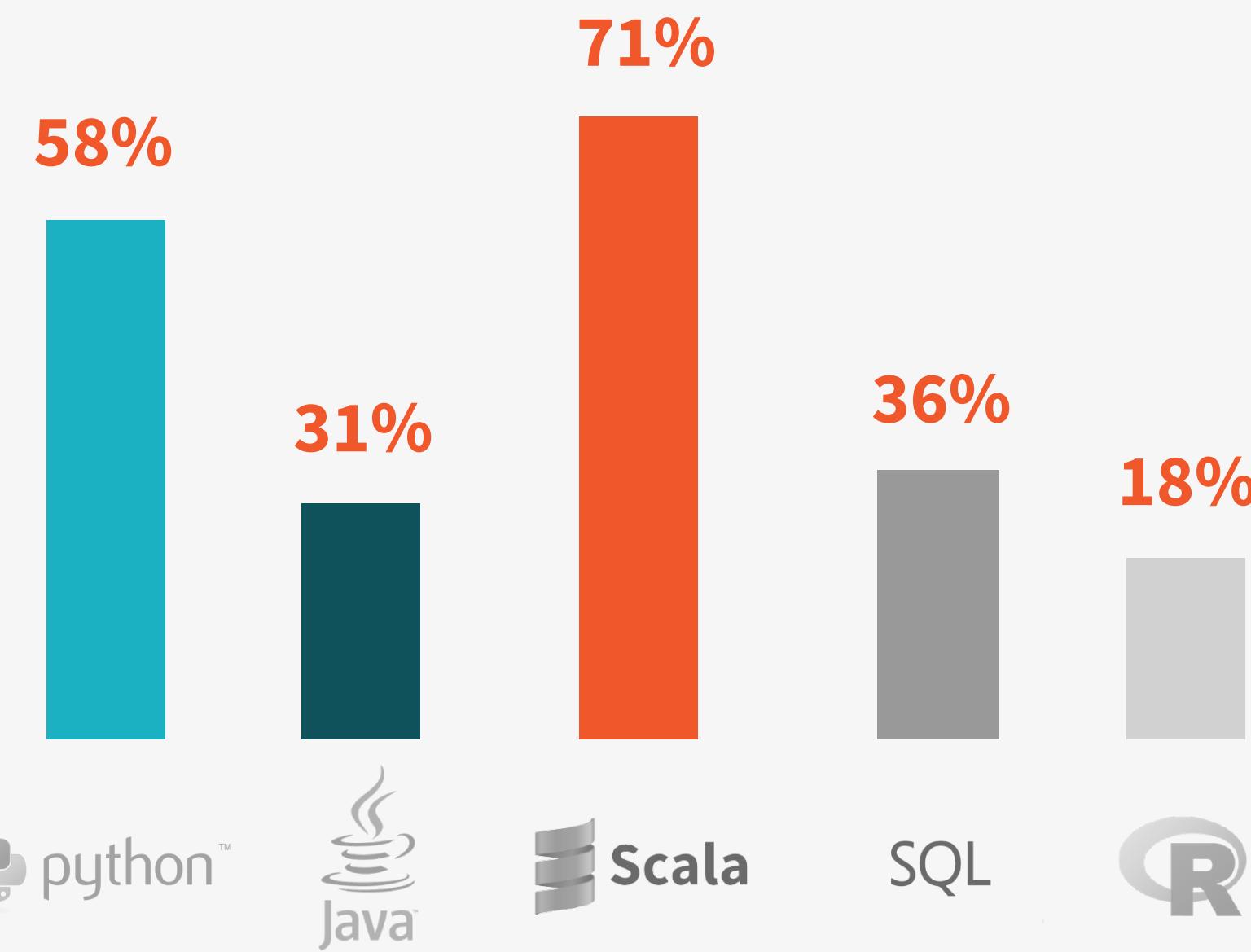
Real-time streaming

28%

SQL Standards

## PROGRAMMING LANGUAGES USED WITH SPARK

*Survey respondents can choose multiple languages.*



Spark users are expanding into the areas of advanced analytics and real-time streaming while building foundations on data warehousing and BI.

2

# Spark vs. MapReduce

# Hadoop MapReduce

Spark vs. MapReduce

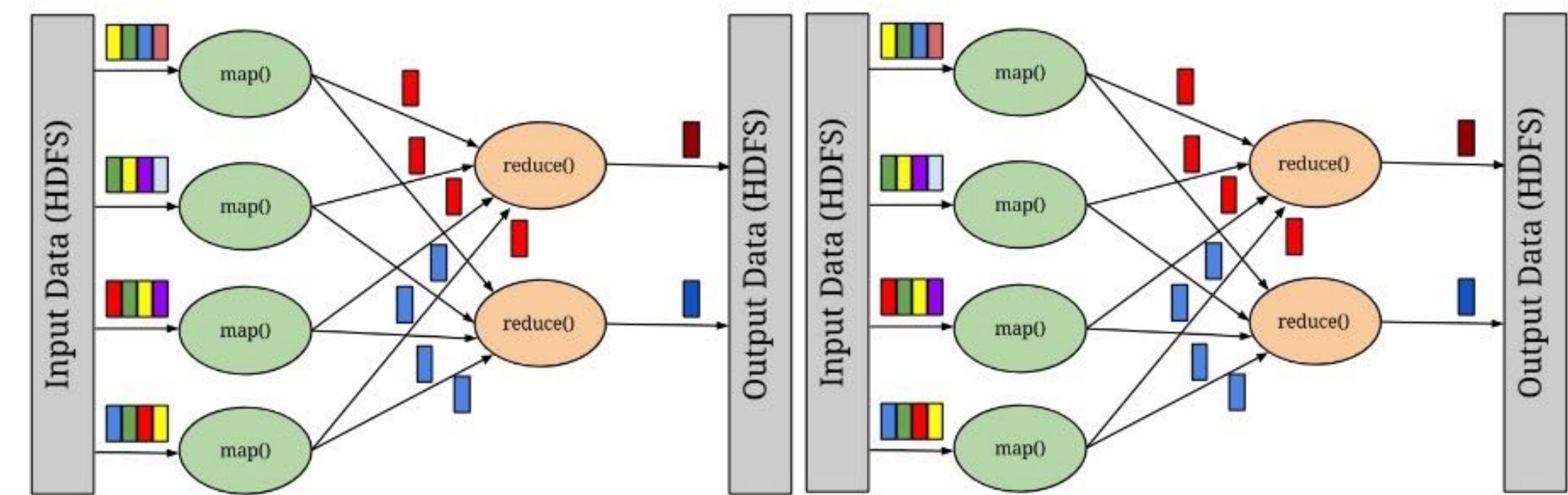
En **2009 et encore en 2014**, l'outil standard pour le traitement des grands volumes de données est **MapReduce** (composant de Hadoop):

- Modèle de programmation popularisé par **Google** (publication en **2004**).
- Une **version open source** est développée dans Hadoop.

Avec MapReduce on code des traitements de données sous forme de **séries de Map et de Reduce**.

Les résultats des Reduce sont **écrits sur HDFS** (système de fichier distribué, utilise les disques durs des machines).

## MapReduce Programming Model



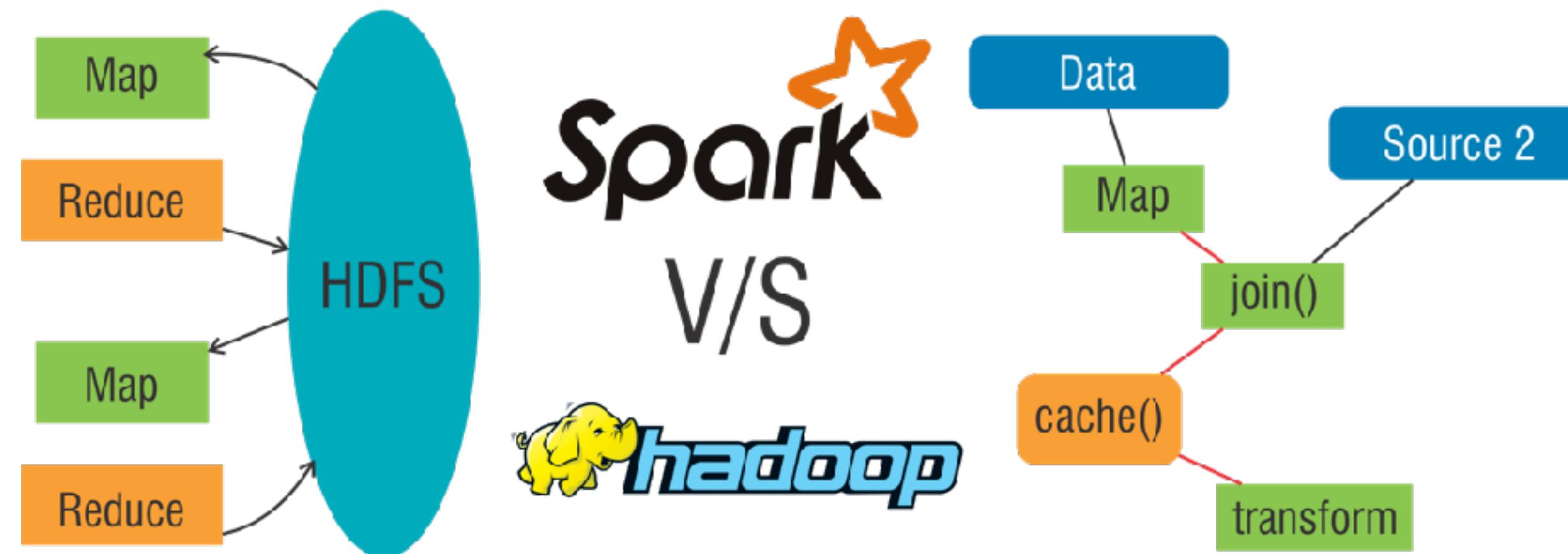
<http://dl.acm.org/citation.cfm?id=1251264>

# Limitations de Hadoop MapReduce

Spark vs. MapReduce

Problèmes :

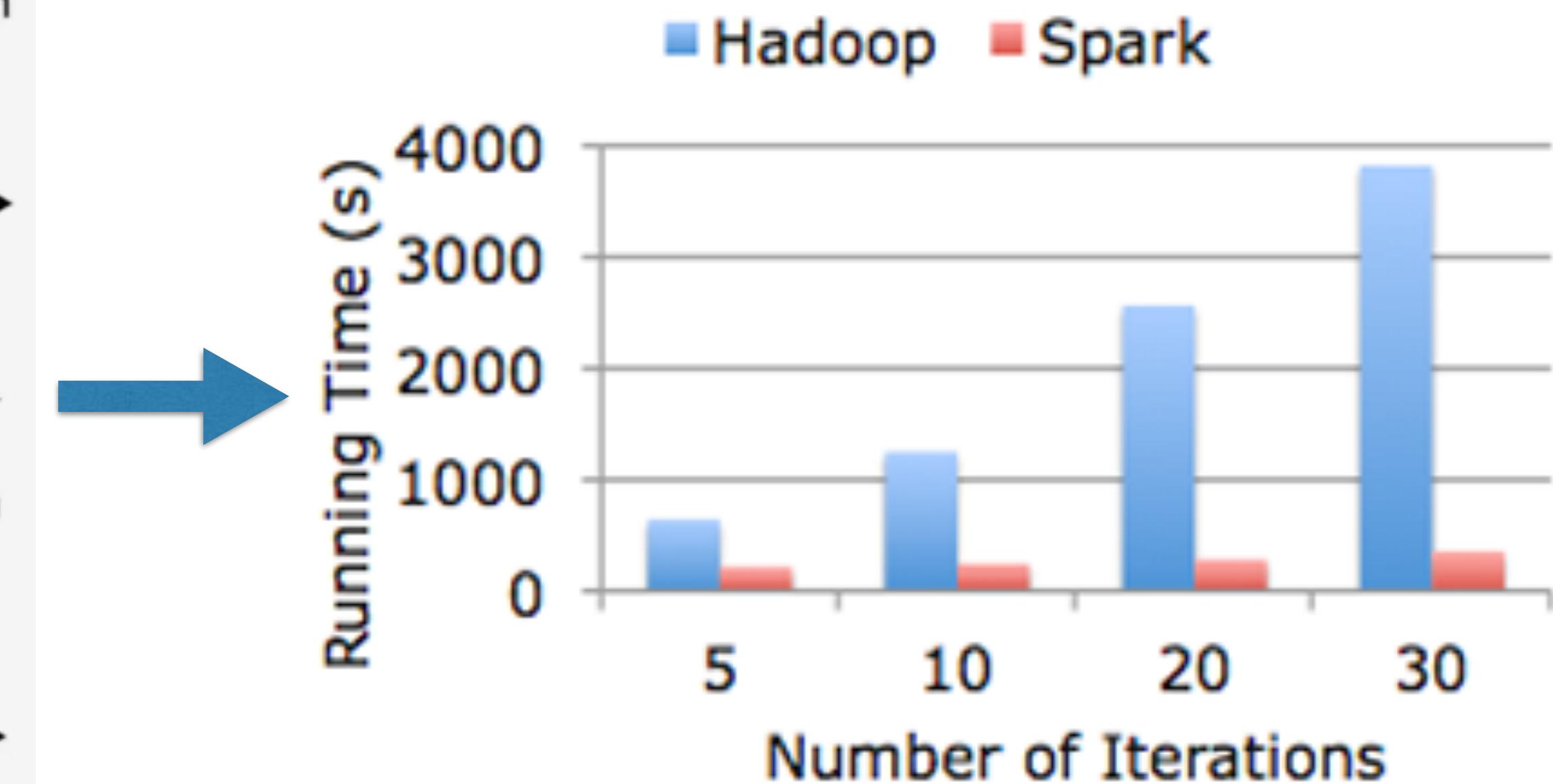
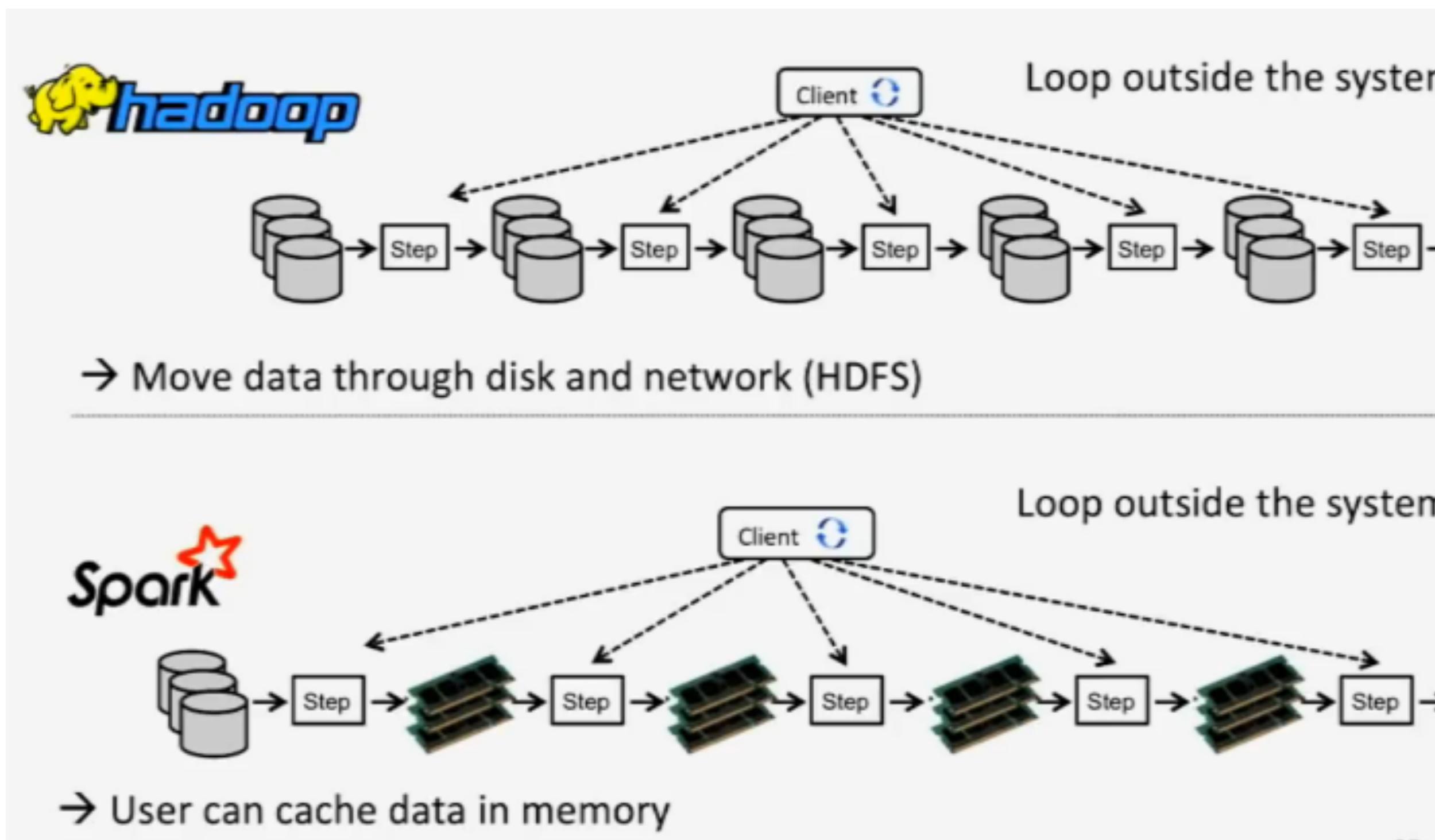
- **Les lectures/écritures de données sur disques** entre chaque étapes d'un job ralentissent les traitements:
  - temps de transfert des données vers les disques
  - temps d'écriture sur disque
  - temps de réplication (données copiées plusieurs fois en cas de panne d'une machine)
- **Le faible nombre d'opérations disponibles** :
  - fonctions Map, Reduce, et très peu d'autres fonctions.
  - le cluster exécute une seule fonction à la fois.



# Solution Spark: Mise en cache des données

Spark vs. MapReduce

- Solution proposée par Spark : tirer parti de la RAM, limiter les lectures/écritures sur disque:
- Les données peuvent être mises en cache => **accès plus rapide**.



# Le DAG

Directed Acyclic Graph

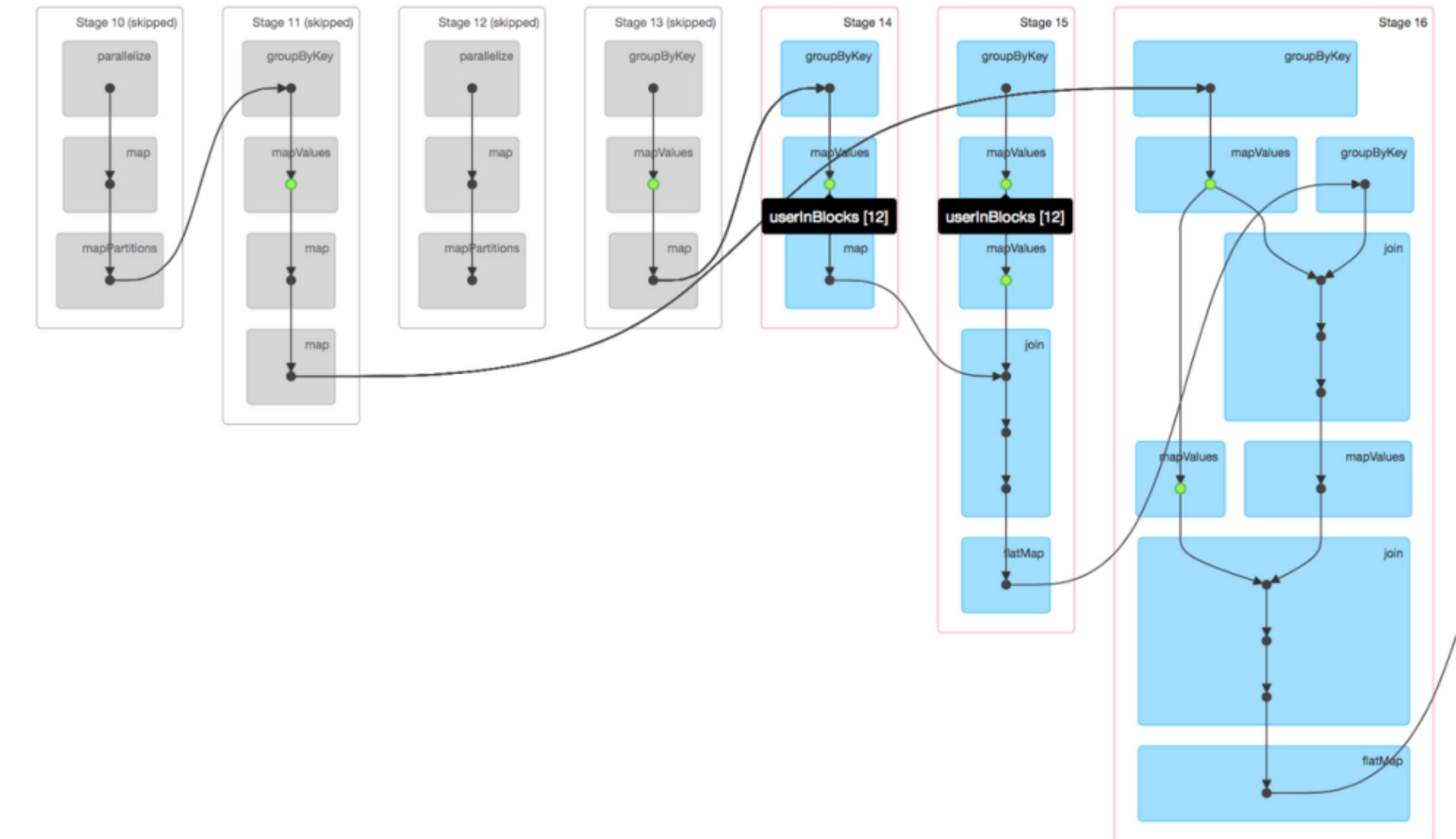
Le **DAGScheduler** permet:

- De découper un script en **Stages** (groupes de tâches qui ne requièrent pas de transfert de données entre les noeuds du cluster, ex: map, filter, ...).
- D'identifier les Stages qui peuvent être exécutés en **parallèle**.
- De déterminer où une tâche doit être exécutée (sur quel exécuteur) pour **minimiser les transferts de données** dans le cluster.
  - ➡ Optimisations du pipeline de traitement des données.
  - ➡ Optimisation de la récupération des données intermédiaires perdues.

## Details for Job 4

Status: SUCCEEDED  
Completed Stages: 22  
Skipped Stages: 4

▶ Event Timeline  
▼ DAG Visualization



# Facilité de programmation

Exemple du Word Count

Spark

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" ")).  
    .map(word => (word, 1))  
.reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

(En Scala)

→ Adoption massive de Spark !

MapReduce (Hadoop)

```
public class WordCount {  
  
    public static class TokenizerMapper  
        extends Mapper<Object, Text, Text, IntWritable> {  
  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(Object key, Text value, Context context  
                        ) throws IOException, InterruptedException {  
            StringTokenizer itr = new StringTokenizer(value.toString());  
            while (itr.hasMoreTokens()) {  
                word.set(itr.nextToken());  
                context.write(word, one);  
            }  
        }  
    }  
  
    public static class IntSumReducer  
        extends Reducer<Text, IntWritable, Text, IntWritable> {  
        private IntWritable result = new IntWritable();  
  
        public void reduce(Text key, Iterable<IntWritable> values,  
                          Context context  
                          ) throws IOException, InterruptedException {  
            int sum = 0;  
            for (IntWritable val : values) {  
                sum += val.get();  
            }  
            result.set(sum);  
            context.write(key, result);  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf, "word count");  
        job.setJarByClass(WordCount.class);  
        job.setMapperClass(TokenizerMapper.class);  
        job.setCombinerClass(IntSumReducer.class);  
        job.setReducerClass(IntSumReducer.class);  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

3

# Composants

# Stack

## Composants

Spark SQL

MLlib/ML

Spark Streaming

GraphX

Spark Core

Spark Standalone

Yarn

Mesos

Storage and Data Sources (HDFS, S3, HBase, Kafka, ...)

- Ce module permet de traiter des **données structurées** (sous forme de table).
- Utilise la structure des données pour optimiser les traitements (façon optimisation de requêtes SQL).
- Les différents moyens d'interagir avec ce module:
  - Requêtes SQL.
  - DataFrames et DataSets.

<http://spark.apache.org/docs/latest/sql-programming-guide.html>

Deux librairies de **machine learning distribué** pour les grands volumes de données.

- MLlib est plus ancien que ML, sera maintenu mais pas développé => **utilisez ML**
- Les DataFrames sont supportés par ML, pas par MLlib (qui supporte uniquement les RDD).
- Outils de **preprocessing** des données (ml.feature)
- Les **algorithmes d'apprentissage** distribués les plus courants sont implémentés dans ml et mllib  
(ex: ml.classification, ml.regression, ml.clustering, ml.recommendation)
- Outils de **tuning** des algorithmes (ml.evaluation et ml.tuning)

<https://spark.apache.org/docs/latest/ml-guide.html>

## Module Streaming

Composants

Spark SQL

MLlib/ML

Spark Streaming

GraphX

Librairie permettant de traiter en **temps réel** des données

- Pas exactement du streaming mais du **micro-batch**
- Nombreux connecteurs pré-existants (Kafka, Flume, Kinesis, ...)
- Crée des **RDD** à chaque pas de temps et les traite ensuite normalement
- Possibilité de créer des fonctions sur des **fenêtres de temps** (ex : moyenne glissante du nombre de tweets)

<http://spark.apache.org/docs/latest/streaming-programming-guide.html>

## Librairie de **calcul distribué sur graphes**.

- Ne supporte que les RDD (pas les DataFrames/DataSets)
- **Uniquement pour les graphes dirigés (construire un graphe non-dirigé oblige à doubler les arêtes)**
- Librairie peu fournie en algorithmes pour l'instant (voir spark.graphx.lib)
- Mais toujours en développement ! (voir les derniers commits sur github)

<https://spark.apache.org/docs/latest/graphx-programming-guide.html>

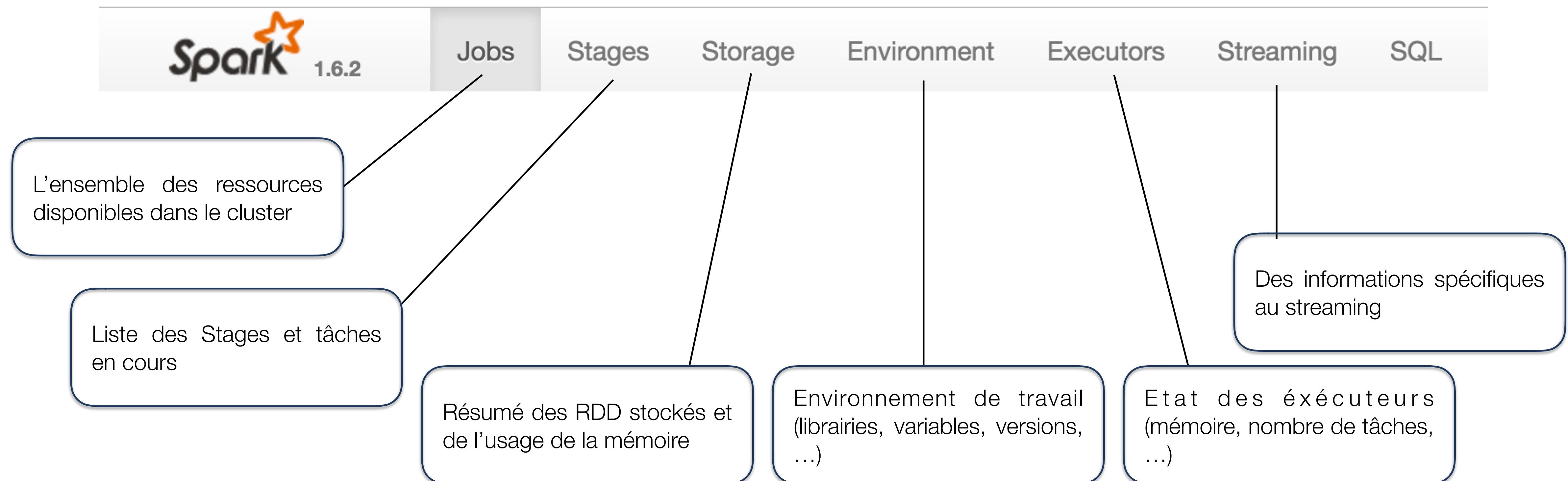
# API (Application Programming Interface)

|        | % utilisation | Commentaires   |
|--------|---------------|--|
| Scala  | 71%           | <ul style="list-style-type: none"><li>• Langage <b>typé et compilé</b></li><li>• Pas besoin d'interface (Spark est codé en Scala)<ul style="list-style-type: none"><li>• une source de bugs en moins !</li><li>• plus rapide dans certains cas</li></ul></li><li>• API plus rapidement à jour</li><li>• Usage :<ul style="list-style-type: none"><li>• <b>utilisable en production</b>,</li><li>• langage utilisé dans de nombreux back-ends</li></ul></li></ul> |
| Python | 58%           | <ul style="list-style-type: none"><li>• Langage <b>non-typé non-compilé</b> il faut lancer le script pour vérifier qu'il n'y a pas d'erreurs.</li><li>• Requiert une interface python-scala.</li><li>• Usage : Python est un langage très utilisé dans la R&amp;D</li></ul>  |
| Java   | 31%           | <ul style="list-style-type: none"><li>• Langage typé et compilé.</li><li>• Pas particulièrement d'avantages sur Scala.</li><li>• Usage: souvent utilisé dans un système déjà développé en Java (legacy)</li></ul>  |
| R      | 18%           | <ul style="list-style-type: none"><li>• Peu recommandé, trop peu développé</li></ul>   |

# Spark UI

## Composants

L'Interface Web d'un cluster Spark est l'interface qui résume l'état du cluster ainsi que ses jobs associés.



4

# Infrastructure

# Ecosystème

## Infrastructure

Spark SQL

MLlib/ML

Spark Streaming

GraphX

Spark Core

Spark Standalone

Yarn

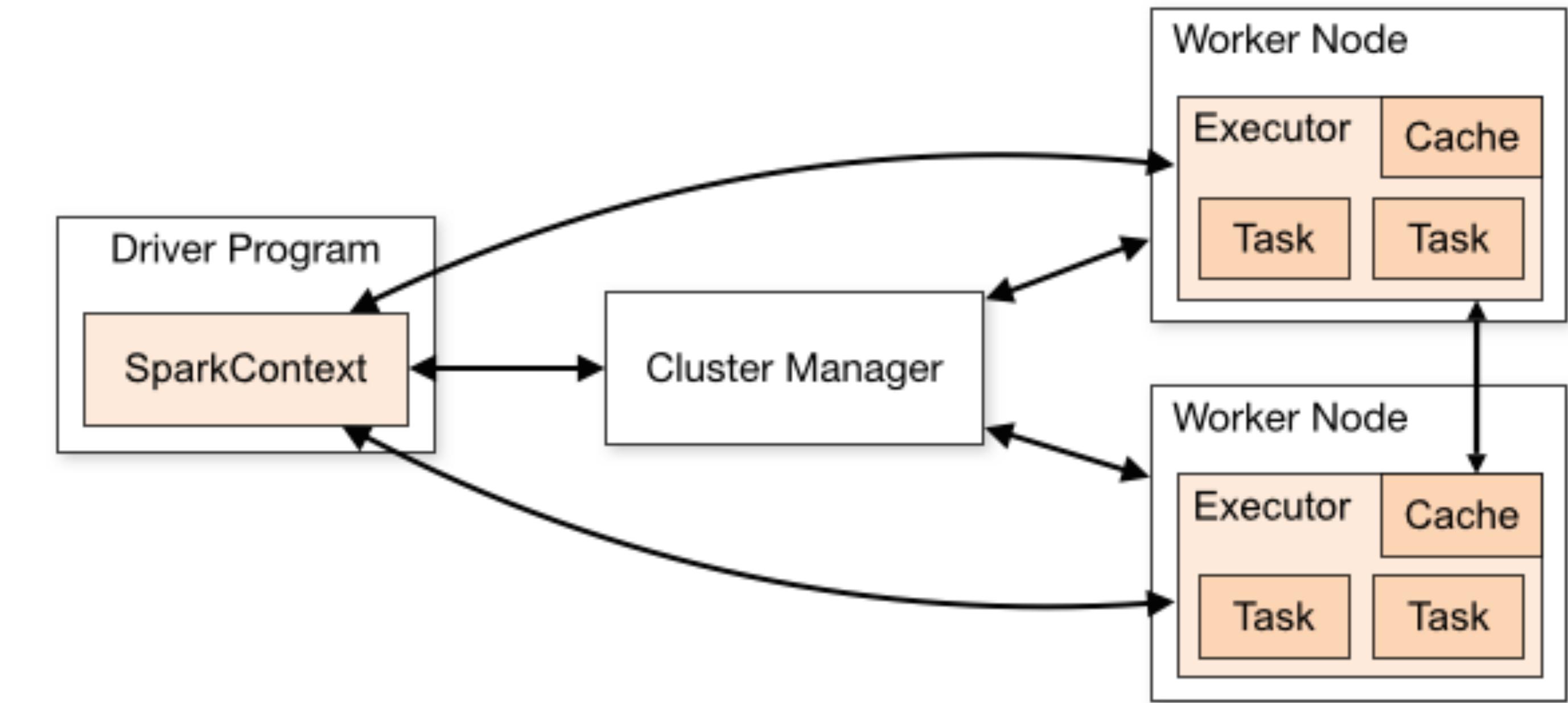
Mesos

Storage and Data Sources (HDFS, S3, HBase, Kafka, ...)

# Cluster

Infrastructure

|                  |                                       |
|------------------|---------------------------------------|
| Cluster Software | Master node                           |
|                  | Worker node                           |
|                  | Cluster manager                       |
|                  | Driver program                        |
|                  | Executor                              |
|                  | Application                           |
|                  | Job                                   |
|                  | Stage                                 |
|                  | Task                                  |
|                  | Application soumise par l'utilisateur |



# Cluster

## Infrastructure

| Component              | Description  |
|------------------------|--|
| <b>Cluster manager</b> | Service externe regroupant l'ensemble des ressources du cluster (i.e. standalone manager, Mesos, YARN)   |
| <b>Master node</b>     | Noeud sur lequel est installé le cluster manager en mode Standalone  |
| <b>Worker node</b>     | Noeud mis à disposition dans le cluster pour faire tourner une application   |
| <b>Driver program</b>  | Processus sur lequel tourne le <i>main()</i> de l'application. Il crée le <i>SparkContext</i>  |
| <b>Executor</b>        | Processus lancé pour une application sur un <i>Worker node</i> . C'est le processus qui exécute les tâches "lourdes" de l'application et stocke en mémoire les résultats intermédiaires. |
| <b>Application</b>     | Application Spark. Elle est constituée d'un <i>driver</i> et d' <i>executors</i> .   |
| <b>Job</b>             | Un calcul parallélisé au sein d'une application consistant en un ensemble de <i>stages</i> menant à une <i>Action</i> (ex : <i>collect</i> )   |
| <b>Stage</b>           | Ensemble de <i>tasks</i> interdépendantes (ex : <i>map</i> puis <i>reduce</i> )  |
| <b>Task</b>            | Unité de travail distribuée sur un <i>executor</i>   |

# Fault-Tolerance

## Infrastructure

Propriété permettant à un système de continuer à **fonctionner normalement** en cas de panne de l'un ou plusieurs de ses composants. L'idée sous-jacente est de ne jamais avoir de **Single Point of Failure** dans le système.

Dans le cas de Spark, cela s'applique à plusieurs parties du système :

- Si un *executor* tombe, le job continue normalement et relance les tâches perdues par l'*executor* perdu.
- Si un *worker* (hardware) vient à disparaître, les *executors* sont perdus (on revient au cas précédent).

# High-Availability

Infrastructure

L'**High Availability** est un concept très présent en systèmes distribués. Cela consiste à minimiser la probabilité et le temps de downtime d'un système.

Dans le cas de Spark, il existe plusieurs moyens de renforcer un cluster :

- Plusieurs machines distinctes peuvent jouer le rôle de *cluster manager* (l'une est **active**, les autres sont **passives**)
- Le *cluster manager* peut lancer le driver en mode “**supervise**” auquel cas un driver est automatiquement relancé après crash
- Possibilité de localiser les machines dans différents racks/**availability zones**/data centers pour être résistant à une panne système
- Possibilité de **duplicer** complètement un système **cross-datacenters**

5

# Quelques détails sur le code

# Scala : programmation fonctionnelle

Très brève introduction

- Les données sont **immuables**:
  - pas de modifications imprévues des données
- Les calculs sont codés dans des **fonctions** (ou dans des méthodes en OOP) :
  - facilite la **réutilisation** de code
  - **lisibilité** du code produit
- Une fonction peut être donnée en entrée d'une autre fonction:
  - utilisation de fonctions anonymes (**lambda functions**)
  - ex: data.map(word => (word, 1))
  - équivalent à : data.map((\_, 1))
- Les fonctions peuvent être « **chaînées** »:
  - permet les lazy evaluations (on y reviendra)

=> On ne fait qu'appliquer un pipeline de fonctions pour transformer les données.

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" ")).  
               .map(word => (word, 1)).  
               .reduceByKey(_ + _).  
counts.saveAsTextFile("hdfs://...")
```

```
// ----- CLEANING -----  
  
val df_os_country = df_raw.  
  .filter($"app_bundle".isNotNull)  
  .filter(length($"app_bundle") >= 1)  
  .filter($"app_bundle" != "%var_bundle%")  
  .withColumn("os", lower($"os"))  
  .filter($"os" === config.os)  
  .withColumn("country", upper($"country"))  
  .withColumn("country", regexp_replace($"country", "FRA", "FR"))  
  .filter($"country" === config.country)  
  .withColumn("idfa", upper($"idfa"))  
  .filter($"idfa".rlike(DataCleaning.idfaRegex))  
  .groupBy("idfa", "app_bundle", "app_name").count
```

```
val df_os_country_coalesced = df_os_country.  
  .coalesce(config.parallelism.toInt)  
  .persist()  
  
df_os_country_coalesced.orderBy($"count".desc).show()
```

# Spark : les RDD

Resilient Distributed Dataset

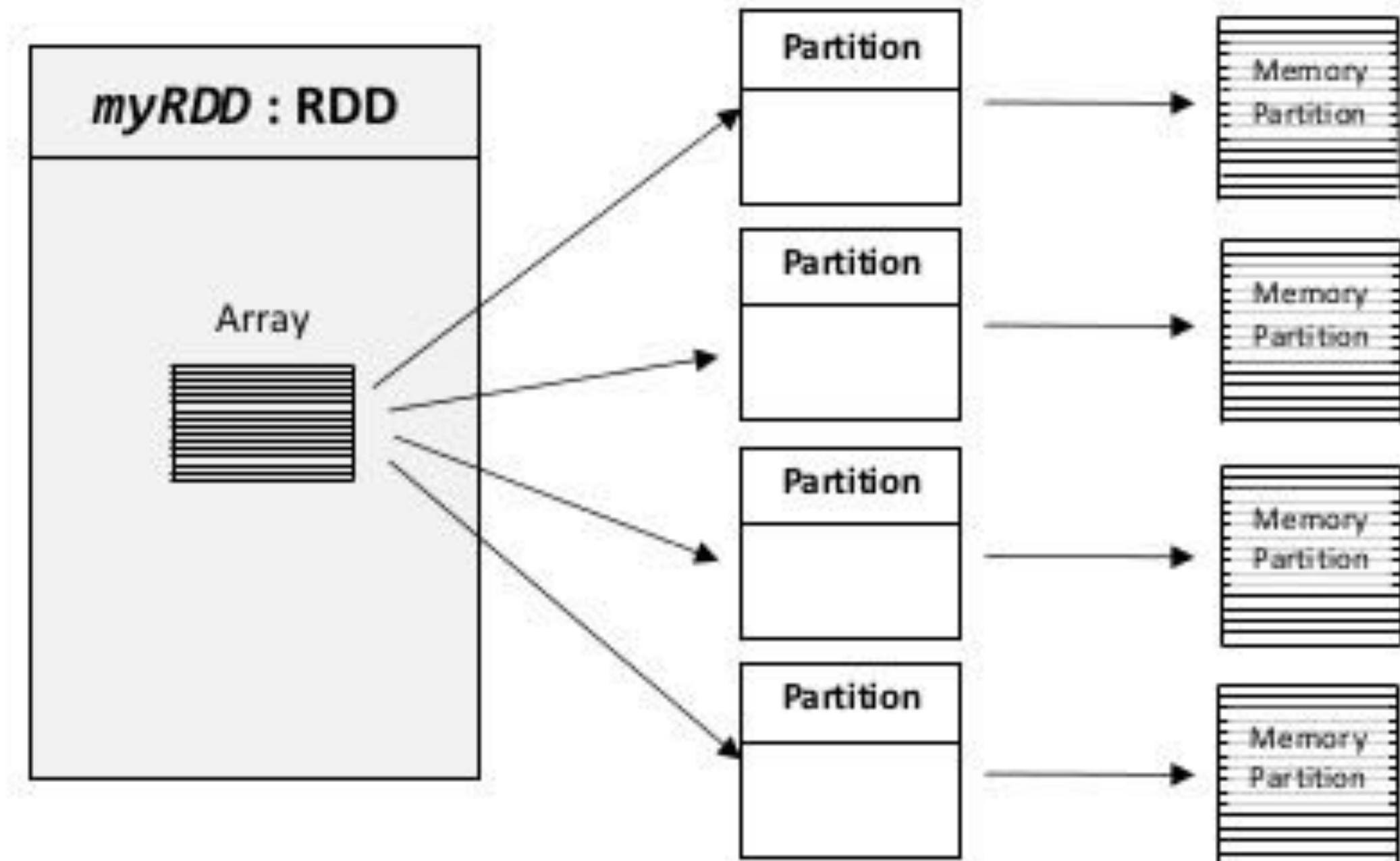
- Abstraction principale de Spark
- Respecte les règles de **fault-tolerance**
- Opérable en **parallèle**
- Représentation sous forme de **collection non ordonnée**

d'éléments

- un élément peut être aussi complexe que nécessaire
- souvent une String
- Crée de 2 manières :
  - en parallélisant une collection existante
    - ex : `sc.parallelize( Array(1,2,3,4,5) )`
  - à partir d'un ou plusieurs fichiers
    - ex : `sc.textFile("data.txt")`

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" ")).  
               .map(word => (word, 1)).  
               .reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

## What is an RDD?



# Details du Word Count

Spark

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" ")).  
    .map(word => (word, 1))  
    .reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

Importation du fichier découpé par ligne, dans un RDD :

[ («# Apache Spark»),  
 «Spark is a fast and general cluster computing system»), ....]

flatMap :

- iteration sur les lignes du RDD => pas de boucle « for » !!
- applique à chaque élément la fonction split( )
- retourne plusieurs éléments en sortie pour un seul élément en entrée
- retourne un nouveau RDD

[ (« # »), («Apache »), (« Spark »),(« Spark »), (« is »), (« a »), (« fast »),  
 (« and »), (« general »), (« cluster »), (« computing »), (« system »), .... ]

map :

- même chose que flatMap mais ne renvoie qu'un seul élément.
- applique la fonction anonyme f(word) = (word, 1)

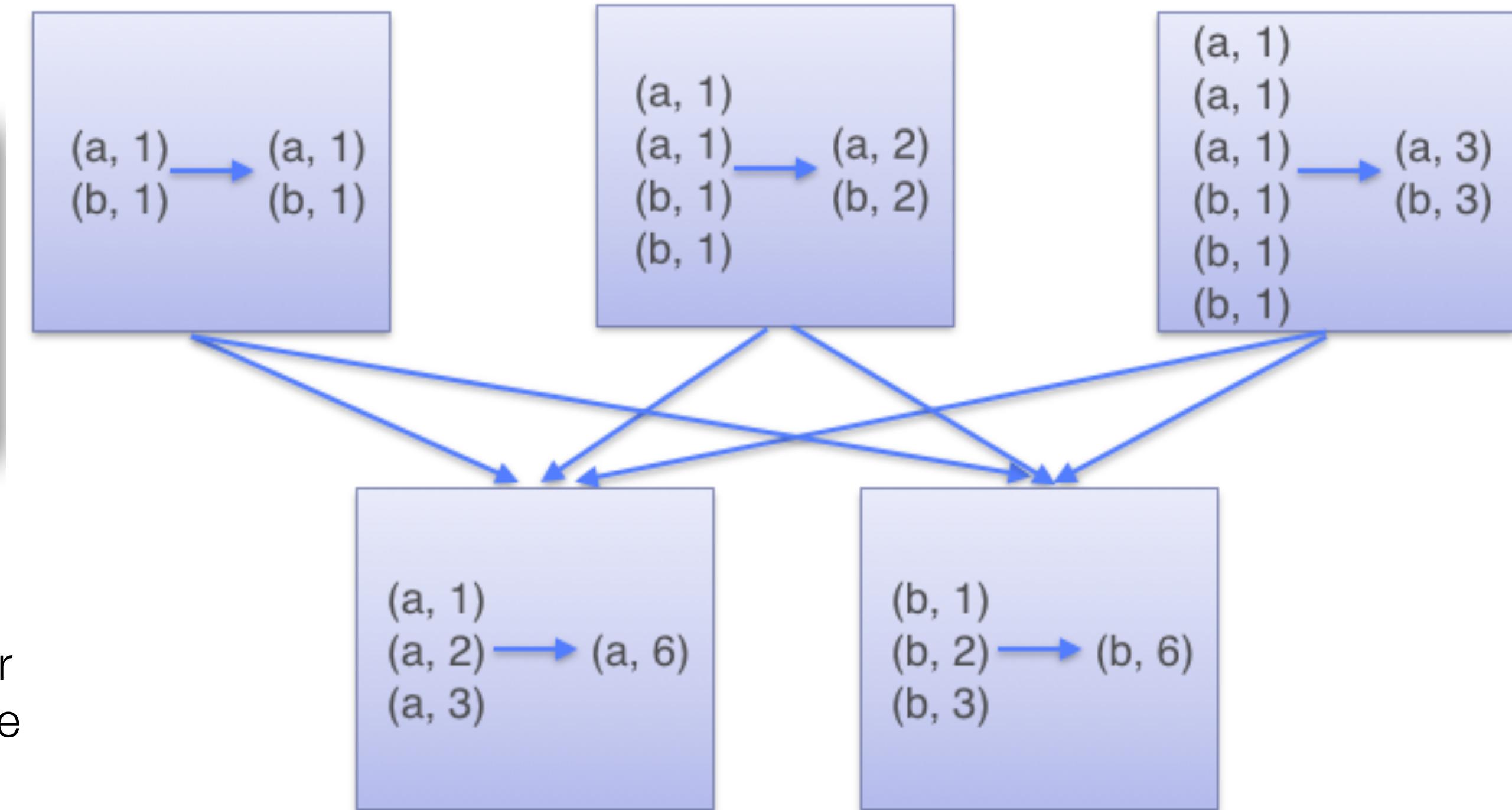
[ («Apache », 1 ), (« Spark », 1),(« Spark », 1), (« is », 1), (« a », 1), (« fast »,  
 1), (« and », 1), (« general », 1), (« cluster », 1), (« computing », 1),  
 (« system », 1), .... ]

# Details du Word Count

Spark

```
val textFile = sc.textFile("hdfs://...")  
val counts = textFile.flatMap(line => line.split(" "))  
    .map(word => (word, 1))  
    .reduceByKey(_ + _)  
counts.saveAsTextFile("hdfs://...")
```

ReduceByKey



fonction d'agrégation de tuples au format (clé, valeur):

- regroupe les tuples ayant la même clé sur un même exécuteur
- puis agrège les valeurs deux à deux, avec la fonction anonyme  $f(valeur1, valeur2) = valeur1 + valeur2$
- jusqu'à ce qu'il n'y ai plus qu'un tuple par clé

Retourne un RDD [(« Apache », 12), (« the », 20), ... ]

6

# Partie pratique

7

# Spark Internals

## Actions / Transformations

Spark Internals

Actions / Tranfo.

Partitionnement

Shuffling

Persistence

- Les **Transformations** sont des opérations chaînables qui produisent des Datasets à partir de Datasets (ex : drop, filter, select,...).
- Les **Actions** sont des opérations qui retournent des valeurs (ex : count, show, take,...).
- Seules les Actions provoquent le lancement d'un calcul à travers le cluster : il est très important d'en minimiser le nombre.

<https://spark.apache.org/docs/2.0.0/api/scala/index.html#org.apache.spark.sql.Dataset>

- Manière dont sont découpés les DataSets sur le cluster
- Une **clé de partitionnement** permet de diviser le DataSet, il faut bien la contrôler pour avoir une division correcte.
- Best practices :
  - Avoir environ **10 partitions par cœur**
  - Ne pas avoir de partitions trop petites (ie calculs ~100ms par partition)
  - **Expérimenter** est la seule manière sûre

- Etape de **redistribution** des données à travers le cluster.
- Plusieurs points négatifs :
  - Souvent très coûteux
  - Nécessite d'écrire sur disque
  - Le **coût augmente** avec la taille du cluster !!
- Best practices :
  - Essayer de **partitionner** au mieux les données initiales
  - Evaluer la possibilité d'utiliser du **Broadcasting**

- Opération permettant de **mettre en cache** un DataSet pour une réutilisation future. (opération **persist**)
- Très utile notamment en Machine Learning (nombreuses **itérations**)
- Faire attention cependant à bien vérifier la taille des données (elles doivent tenir en RAM/sur disque et laisser de la place pour les calculs)