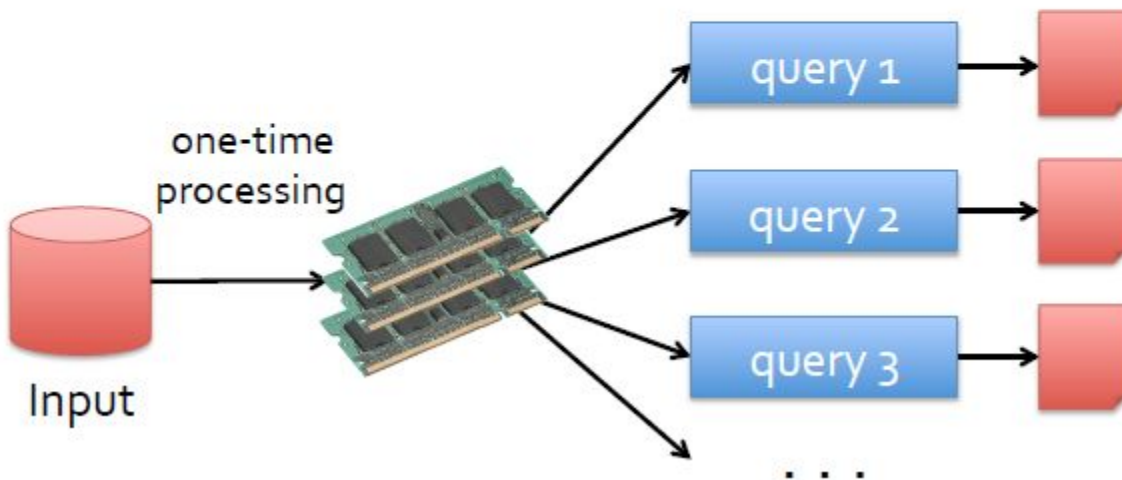
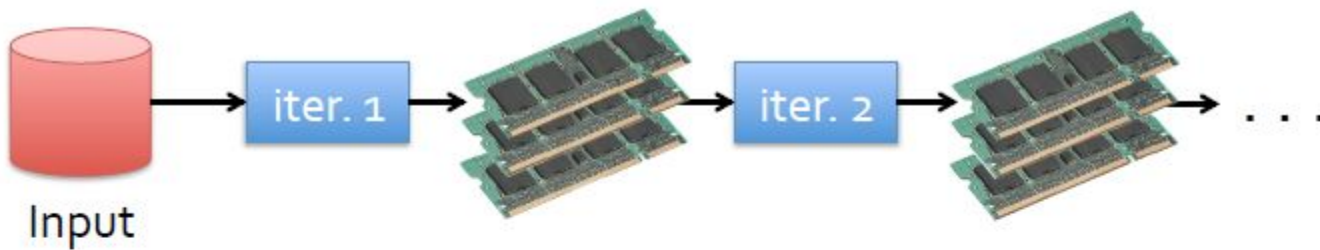


Spark

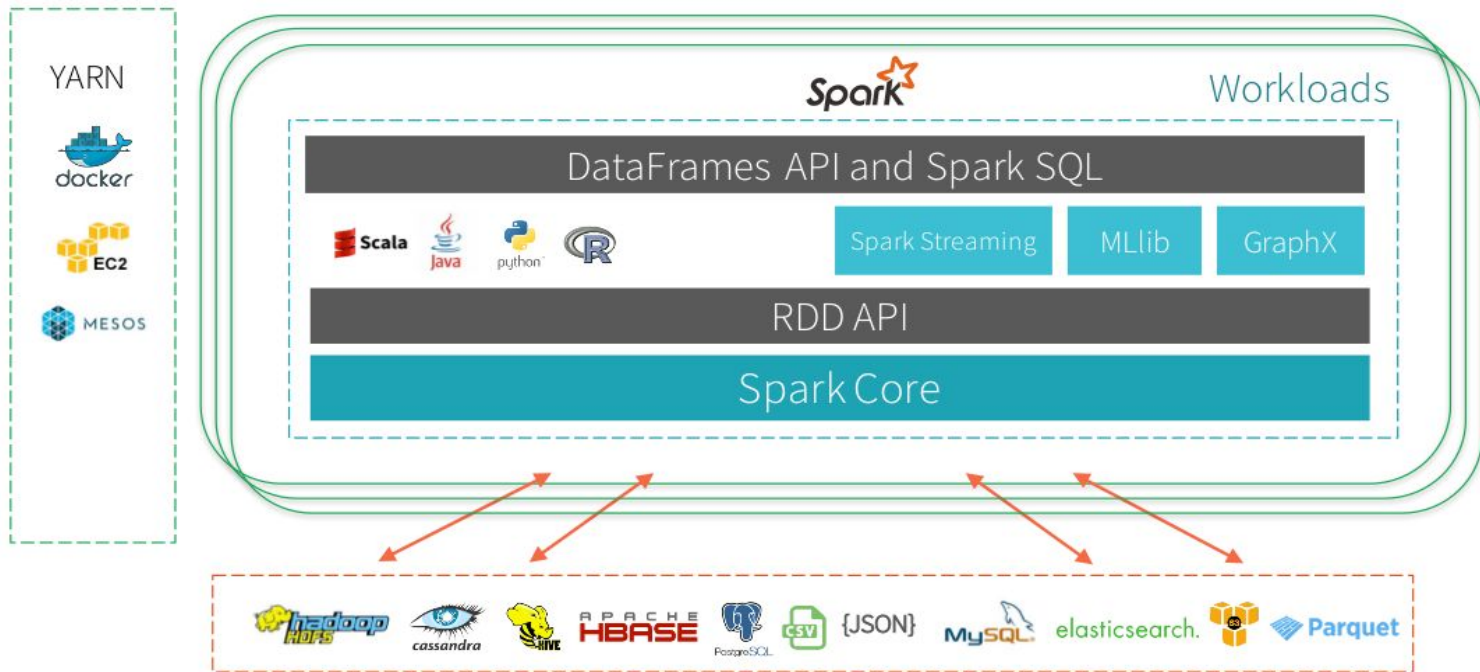


- Framework de traitements Big Data in-memory
- Développé par UC Berkeley en 2009
- Développé par plus de 800 développeurs de plus de 250 entreprises
- Projet Apache le plus actif
- 100 fois plus rapide que Map Reduce en mémoire, 10 fois plus rapide sur disque
- Api en Java, Scala, Python ou R
- Très populaire pour lancer des algorithmes de ML itératifs

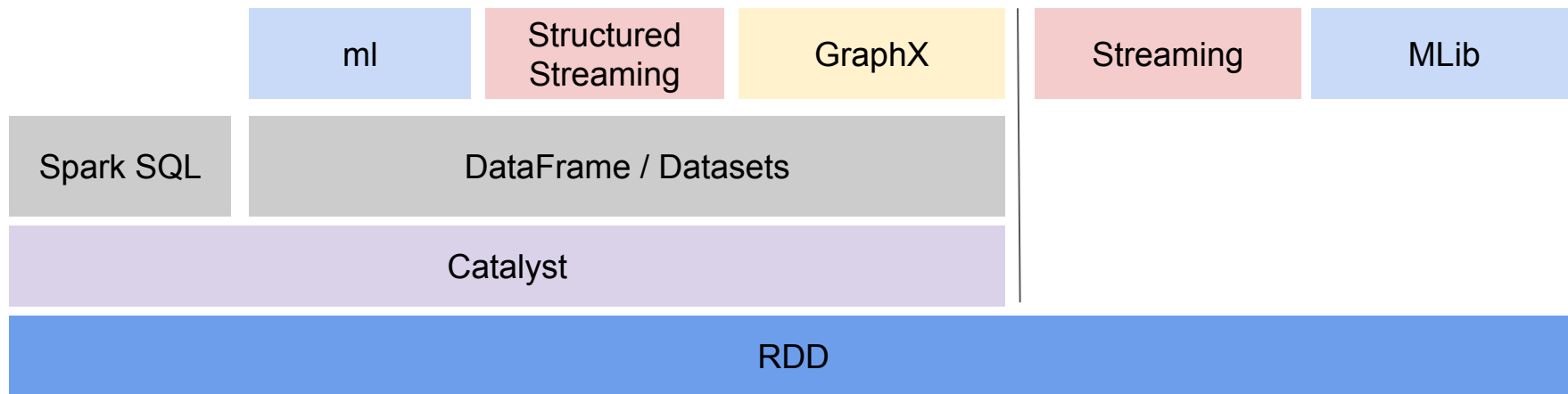


Moteur unifié, avec diverses sources, dans des environnements divers

Environments



Un framework complet:



- **Spark Streaming** : [Spark Streaming](#) peut être utilisé pour traitement temps-réel des données en flux. Il s'appuie sur un mode de traitement en "micro batch" est utilisé pour les données temps-réel DStream, c'est-à-dire une série de RDD (Resilient Distributed Dataset).
- **Structured Streaming** : [Structured Streaming](#) est un moteur de traitement tolérant aux pannes basé sur le moteur Spark SQL. Vous pouvez exprimer votre calcul en continu de la même manière que vous exprimeriez un calcul batch sur des données statiques.
- **Spark SQL** : [Spark SQL](#) permet d'exposer les jeux de données Spark via API JDBC et d'exécuter des requêtes de type SQL en utilisant les outils BI et de visualisation traditionnels. Spark SQL permet d'extraire, transformer et charger des données sous différents formats (JSON, Parquet, base de données) et les exposer pour des requêtes ad-hoc.
- **Spark MLlib** : [MLlib](#) est une librairie de machine learning qui contient tous les algorithmes et utilitaires d'apprentissage classiques, comme la classification, la régression, le clustering, le filtrage collaboratif, la réduction de dimensions, en plus des primitives d'optimisation sous-jacentes.
- **Spark GraphX** : [GraphX](#) est la nouvelle API (en version alpha) pour les traitements de graphes et de parallélisation de graphes. GraphX étend les RDD de Spark en introduisant le Resilient Distributed Dataset Graph, un multi-graphe orienté avec des propriétés attachées aux nœuds et aux arêtes.



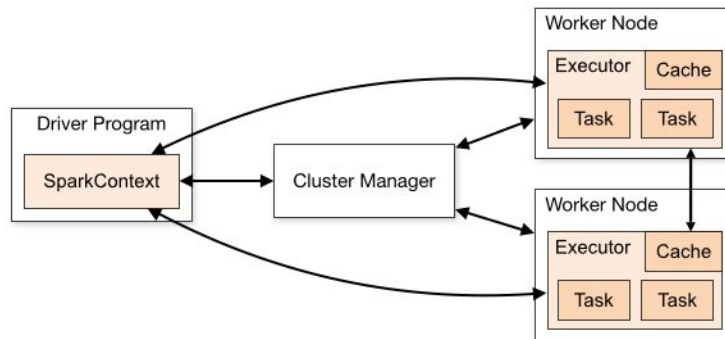
{ JSON }



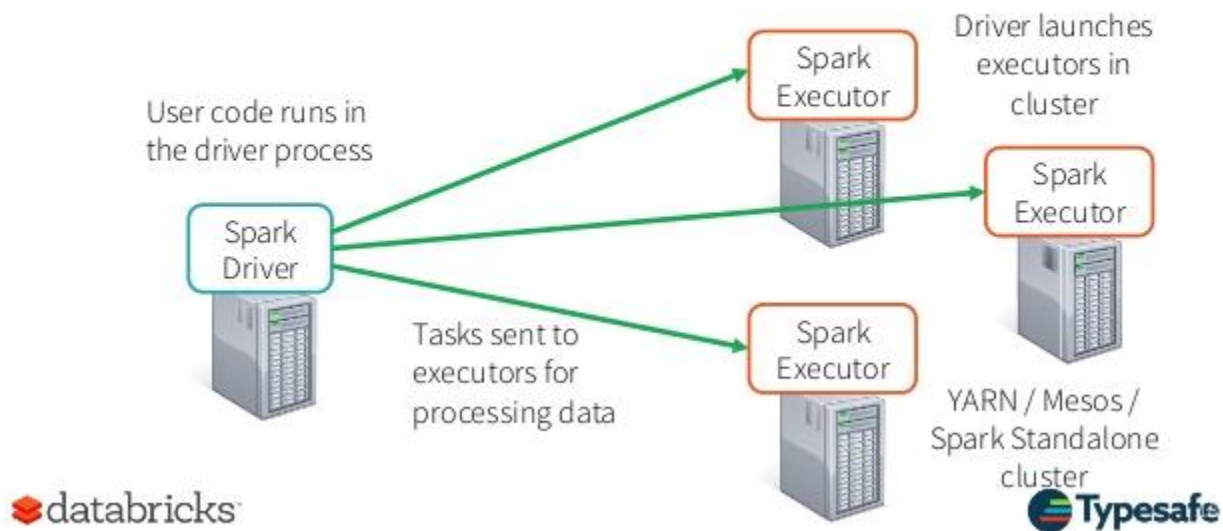
and more ...

Formats and Sources supported by DataFrames

- Les applications Spark se lancent comme un ensemble indépendant de processus sur un cluster qui sont coordonnées par le SparkContext (Driver Program)
- Peut fonctionner sur plusieurs types de cluster manager (Spark standalone, Mesos, Yarn) qui alloue des ressources aux applications
- Spark acquiert des executors sur des noeuds du cluster qui effectuent des calculs et stockent des données pour votre application
- Spark envoie le code de votre application sur les executors, et le Spark Context envoie des tâches aux executors qui les lancent



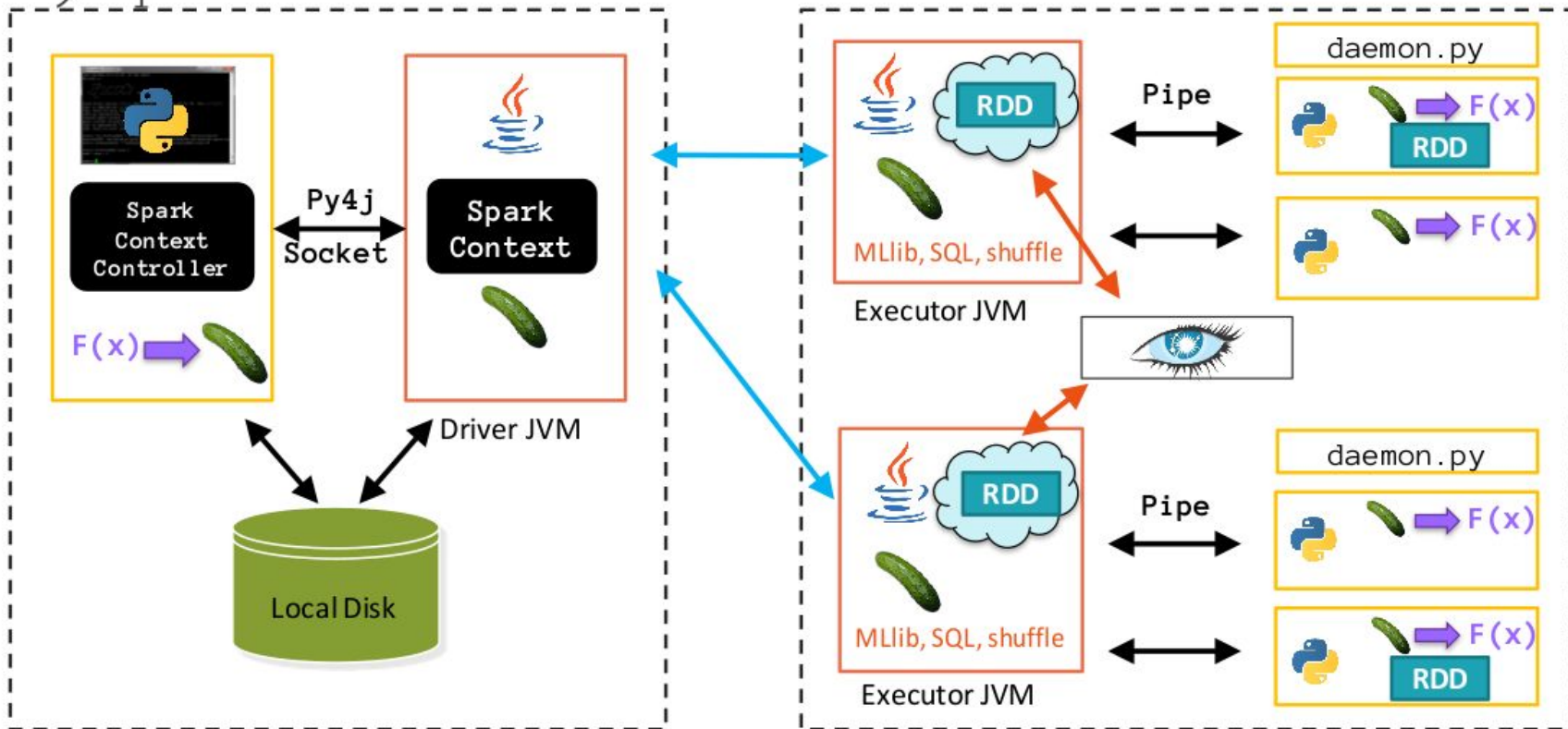
Any Spark Application



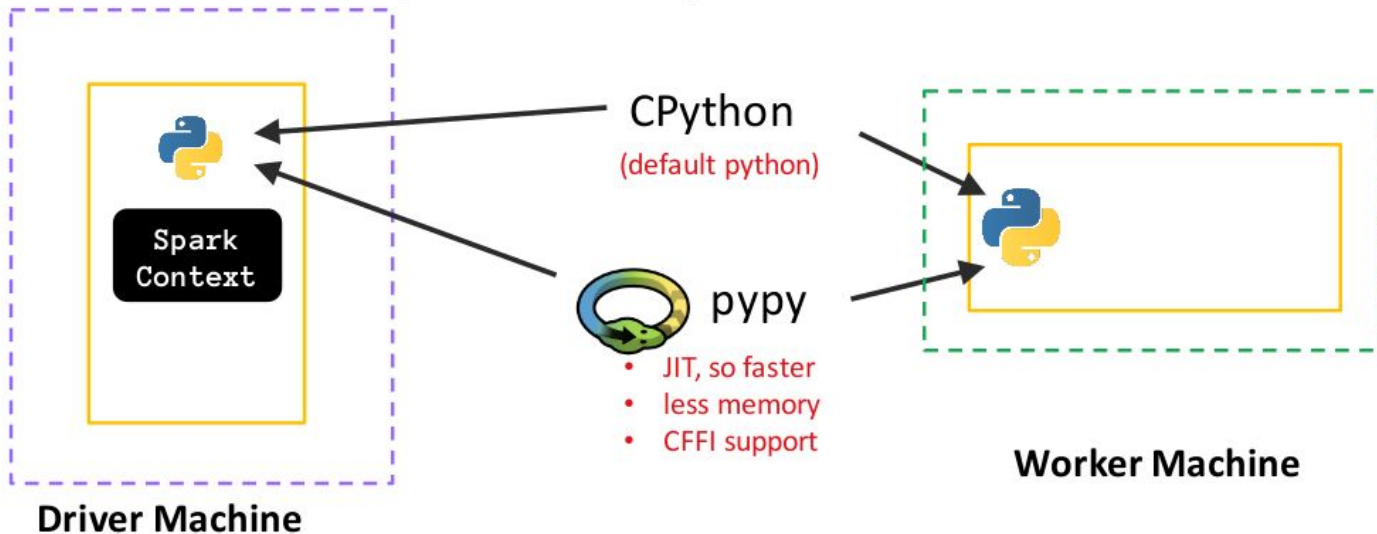
A propos de cette architecture :

- Chaque application obtient ses propres executors qui exécutent des tâches dans plusieurs threads (isolation des application, ordonnancement et exécution)
- Spark est agnostique du cluster manager sous-jacent
- Le Driver Program doit écouter et accepter les connexions provenant de ses executors durant sa durée de vie ce qui implique qu'il doit être adressable depuis les workers nodes
- Parce que le driver schedule des tâches sur le cluster, il doit être lancé au plus proche des workers nodes

PySpark Architecture



Choose Your Python Implementation



```
$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/pyspark
```

OR

```
$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/spark-submit wordcount.py
```

- Application : Programme utilisateur construit sur Spark (driver program + executors)
- Application jar : Le jar contenant l'application utilisateur
- Driver program : Processus qui lance la fonction main et qui crée le spark context
- Cluster manager : Service externe permettant d'acquérir les ressources du cluster
- Deploy mode : Indique où le driver program se lance (cluster, client)
- Worker node : Tout nœud qui peut exécuter du code de l'application dans le cluster
- Executor : Processus lancé pour une application sur un worker node, lance les tasks et garde les données en mémoire ou sur disque entre elles
- Task : Morceau de travail qui sera envoyé à un executor
- Job : Calcul parallèle constitué de plusieurs tâches dont la création est basé sur une Action Spark
- Stage : Chaque job se divise en petit ensemble de tâches appelées "stages" qui dépendent les unes des autres

Le système supporte 4 clusters managers :

- Standalone : Un simple cluster manager inclus avec Spark qui rend facile la mise en place d'un cluster
- Hadoop Yarn : le ressource manager d'Hadoop 2
- Apache Mesos : Un cluster manager général qui peut aussi fonctionner avec Hadoop Map Reduce
- Kubernetes: Un cluster manager basé sur les containers docker

Il est possible de soumettre un job spark à un cluster à l'aide de la commande spark-submit.

- Il existe deux modes d'exécution:
 - Client: Le driver est exécuté depuis la machine sur laquelle le spark-submit a été lancé.
 - Cluster: Le cluster gère lui-même le noeud sur lequel le driver sera exécuté
- Il est possible de configurer les ressources qui seront utilisées par les executor lors de la soumission du job:

```
spark-submit --class org.apache.spark.examples.SparkPi \  
  --master spark://207.184.161.138:7077 --deploy-mode cluster \  
  --supervise --executor-memory 20G --total-executor-cores 100 \  
  /path/to/examples.jar 1000
```

- Les adresse valides pour un master spark sont les suivantes:

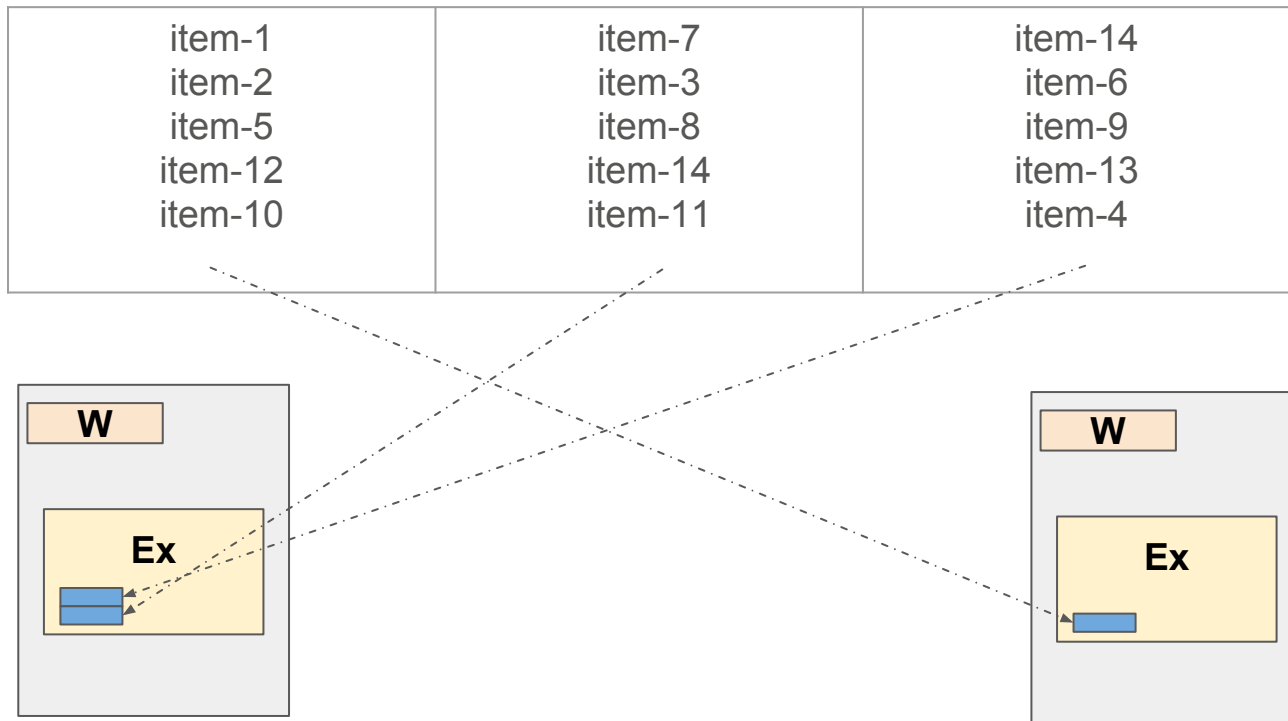
Master	Description
local	Exécute Spark localement avec un worker
local[n]	Exécute Spark localement avec n workers
local[*]	Exécute Spark localement avec un worker par coeur logique de la machine
spark://HOST:PORT	Se connecte au cluster Spark standalone
mesos://HOST:PORT	Se connecte au cluster mesos indiqué
yarn	Se connecte à un cluster Yarn
k8s://https://HOST:PORT	k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port>

- Il est également possible d'exécuter du code spark à l'intérieur d'un spark-shell
- Des notebooks sont également à disposition pour développer des applications spark (*Spark Notebook, Jupyter, Zeppelin...*)

Data = Resilient Distributed Datasets (RDD):

- Collections immutables d'objets
- Propagées sur les noeuds du cluster
- Construites par des opérations parallélisées (map, filter ...)
- Tolérantes aux erreurs
- Dont la persistance peut être contrôlée (cache mémoire et/ou disque)

Un exemple de RDD avec 3 partitions



Quelques types de RDDs

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- Double RDD
- JdbcRDD
- JsonRDD
- VertexRDD
- EdgeRDD

- CassandraRDD
- GeoRDD
- EsSpark

2 types d'opérations : transformations et actions

- Les transformations créent un nouveau jeu de données à partir d'un existant :

map : transforme chaque élément d'un jeu de données à travers une fonction et retourne un nouveau RDD

- Les actions qui retournent une valeur au driver après l'exécution d'un calcul sur le jeu de données

reduce : agrège les éléments d'un RDD en utilisant une fonction et retourne le résultat final au driver program

- Toutes les transformations sont lazy, calculés seulement quand une action nécessite un résultat
- Par défaut, chaque RDD transformé est recalculé chaque fois qu'une action est effectuée dessus \Rightarrow cache/persist méthode pour un accès plus rapide



= easy



= medium

Essential Core & Intermediate Spark Operations



TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe



ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- foreach
- top
- treeAggregate
- treeReduce
- foreachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile



= easy



= medium

Essential Core & Intermediate PairRDD Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

- partitionBy

ACTIONS



- keys
- values

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

Transformation	Description
map (<i>func</i>)	Retourne un nouvel RDD en passant chaque élément source à travers une fonction <i>func</i> .
filter (<i>func</i>)	Retourne un nouvel RDD formé par la sélection des éléments de la source sur laquelle <i>func</i> retourne true.
flatMap (<i>func</i>)	Similaire à <i>map</i> , mais chaque élément d'entrée peut être mappé à 0 ou plusieurs éléments en sortie (<i>func</i> devrait retourner une Seq plutôt qu'un seul élément).
mapPartitions (<i>func</i>)	Similaire à <i>map</i> , mais lancé séparément sur chaque partition du RDD, donc la fonction doit être de type <code>Iterator<T> => Iterator<U></code> lorsqu'on la lance sur un RDD de type T.

Transformation	Description
mapPartitionsWithIndex (<i>func</i>)	Similaire à mapPartitions, mais fournit avec un entier représentant l'index de la partition, donc la fonction doit être de type <code>(Int, Iterator<T>) => Iterator<U></code> lorsqu'on la lance sur un RDD de type T.
distinct (<i>func</i>)	Retourne un nouvel RDD contenant les éléments distincts du jeu de données source.
groupByKey (<i>func</i>)	Lorsqu'elle est appelée sur un jeu de données de paires (K , V), renvoie un jeu de données de paire (K , Iterable < V >).
reduceByKey (<i>func</i>)	Lorsqu'elle est appelée sur un jeu de données de paires (K , V), renvoie un jeu de données de paires (K, V) où les valeurs pour chaque clé sont agrégées en utilisant la fonction reduce func de type (V, V) => V.

Action	Description
reduce (<i>func</i>)	Agrège les éléments du dataset en utilisant une fonction <i>func</i> (qui prend 2 arguments et en retourne 1). La fonction doit être associative et commutative.
collect ()	Retourne tous les éléments du jeu de données comme un tableau au driver program.
first ()	Retourne le premier élément du dataset.
take (<i>n</i>)	Retourne un tableau avec les <i>n</i> premiers éléments du jeu de données.

Action	Description
takeOrdered (<i>n</i> , [<i>ordering</i>])	Retourne les n premiers éléments du RDD utilisant leur ordre naturel ou un comparateur personnalisé.
saveAsTextFile (<i>path</i>)	Écrit les éléments du jeu de données comme un fichier texte (ou un ensemble de jeu de données) dans un répertoire du fichier de système local, HDFS ou n'importe autre fichier système Hadoop supporté. Spark appellera toString sur chaque élément pour le convertir en une ligne d'un fichier texte.
countByKey ()	Dispo seulement sur les RDD de type (K,V). Retourne une HashMap de paires (K, Int) avec le nombre de chaque clé.
foreach (<i>func</i>)	Lance une fonction func sur chaque élément du dataset. Utilisé pour mettre à jour un Accumulator ou pour interagir avec un système de stockage externe. Modifier des variables autre que des Accumulators à l'extérieur d'un foreach() peut donner un résultat indéfini.

SparkContext = point d'entrée d'un programme, configure l'exécution (Nb de noeuds, de thread, mémoire, master ...)

```
val conf = new SparkConf().setAppName("App").setMaster("local")  
val sc = new SparkContext(conf)
```

- Permet de créer les RDDs :

```
val data = Array(1, 2, 3, 4, 5)  
val distData = sc.parallelize(data)
```

- broadcast variables : variables qui peuvent être utilisées pour mettre en cache une valeur en mémoire sur tous les noeuds
- accumulators : variables qui ne peuvent être que incrémentées comme les compteurs ou les sommes

- Les variables de broadcast permettent aux développeurs de garder une variable en lecture seule en cache sur chaque machine
- Spark broadcast automatiquement les données communes nécessaires par les tasks à l'intérieur d'une stage
- Donc à utiliser seulement explicitement seulement quand des tâches à travers de multiples stages ont besoin d'une même donnée

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

- Les accumulators sont des variables qui sont seulement en ajout à travers d'une opération associative et peut être lancé efficacement en parallèle
- La valeur peut être lue seulement par le driver program

```
val accum = sc.accumulator(0, "My Accumulator")
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
accum.value
```

- Spark supporte de base les accumulators de type int. Possibilité de développer ses propres types en héritant de la classe AccumulatorParam

- Fichier de système local, HDFS, Cassandra, Hbase, Amazon S3
- Spark support les fichiers textes, SequenceFiles et autres format d'entrée d'Hadoop
- Création d'un RDD de fichier texte : `val distFile = sc.textFile("data.txt")` \Rightarrow 1 enregistrement par ligne
- Si vous utilisez un chemin sur le système de fichier local, le fichier doit être présent sur tous worker nodes.
- Toutes les méthodes de Spark basées sur un fichier d'entrée, prennent en charge l'exécution sur les répertoires et les, fichiers compressés :

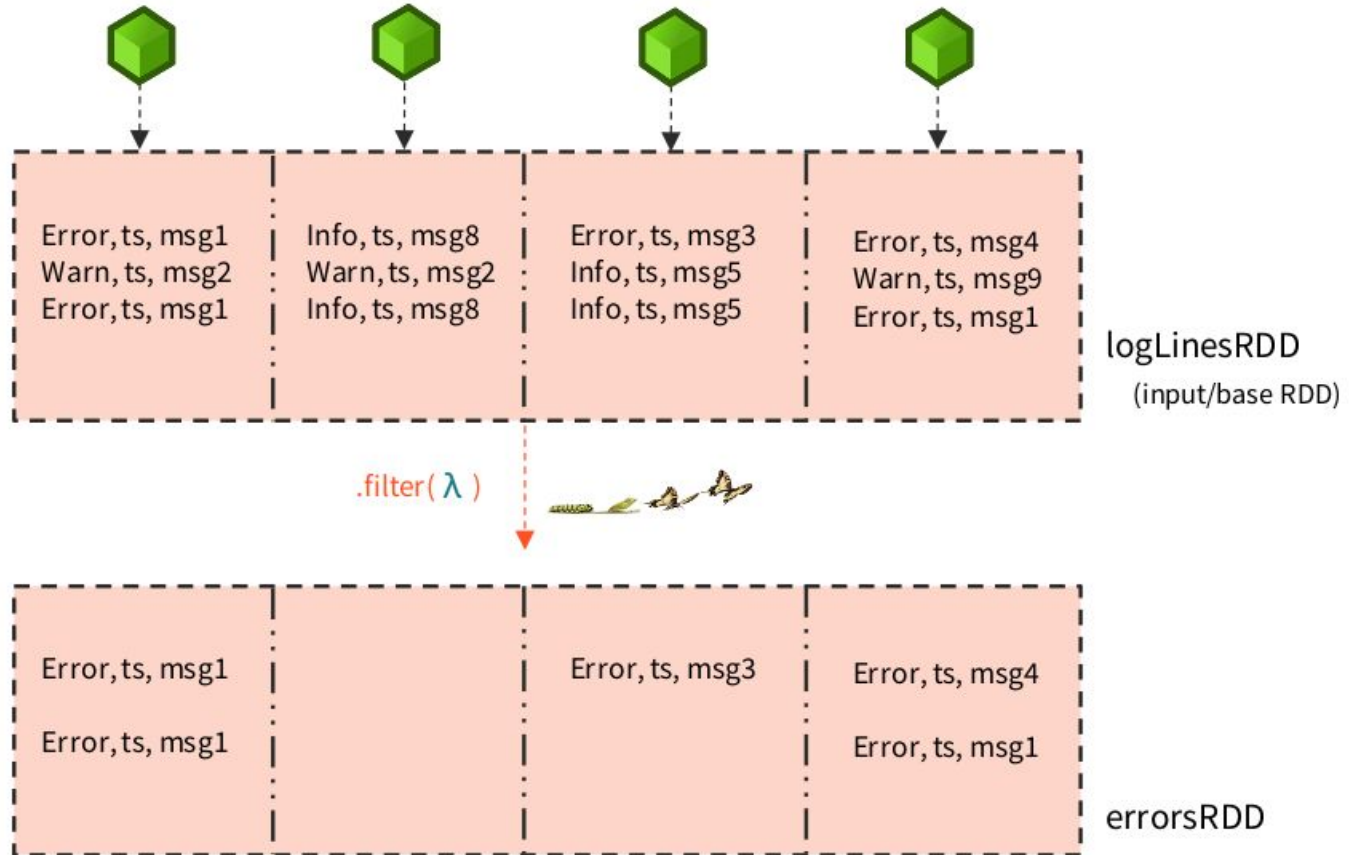
```
textFile("/my/directory"), textFile("/my/directory/*.txt"),  
textFile("/my/directory/*.gz")
```

- La méthode `textFile` prend un argument optionnel pour augmenter le nombre de partitions créer à partir d'un fichier (par défaut Spark créé une partition par bloc de fichier)
- `textFile` = 1 enregistrement par ligne
- `wholeTextFiles` : retourne une paire (filename, content) par fichier lu dans un répertoire
- `sequenceFile[K, V]` pour lire des sequences file où K et V sont les type de clés et de valeurs du fichier

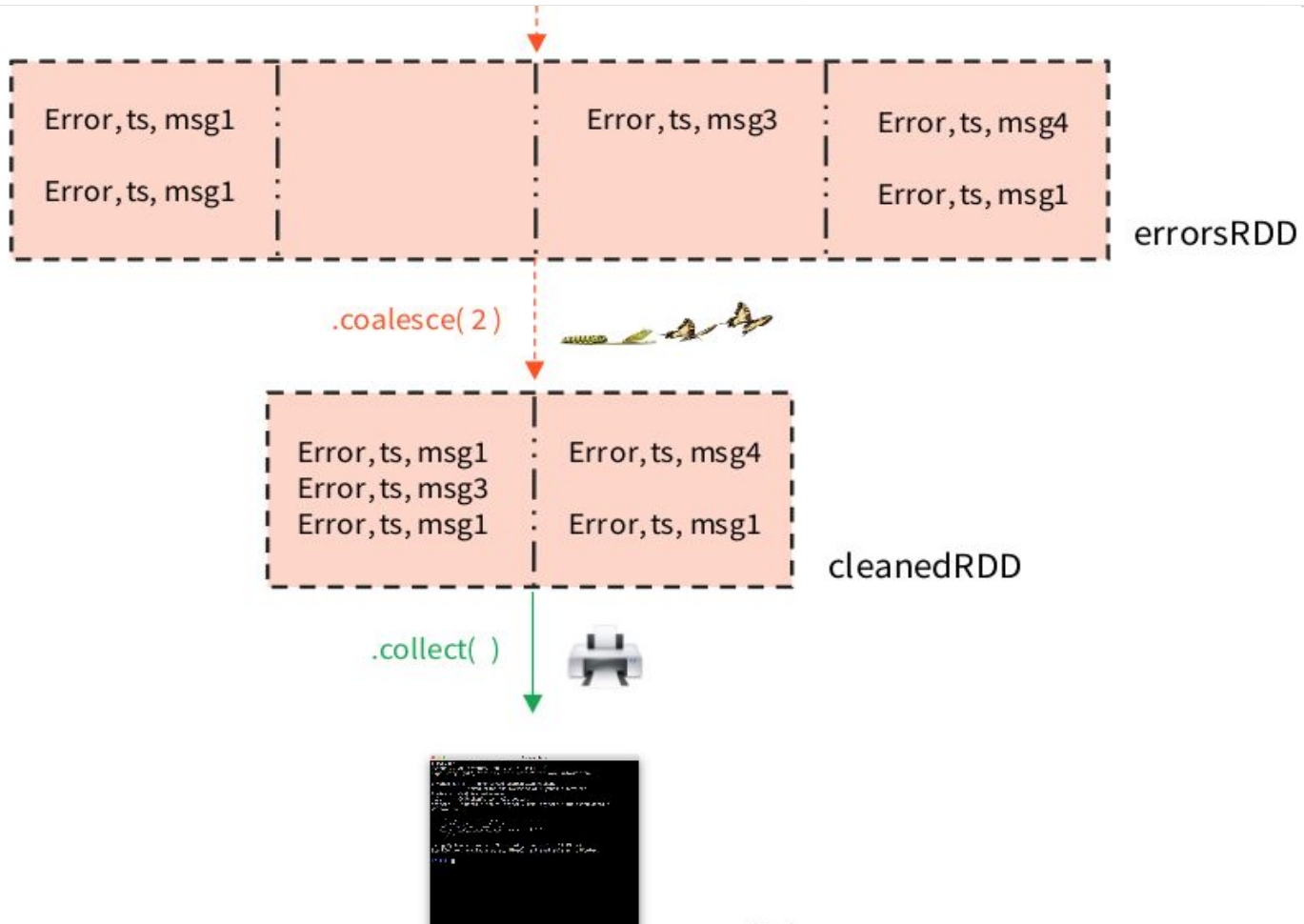

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

- La première ligne définit un RDD à partir d'un fichier texte
- La deuxième ligne définit lineLengths comme étant le résultat d'une transformation qui calcule la longueur d'une ligne
- Finalement on lance une action reduce, Spark découpe le calcul en tâche à exécuter sur des machines différentes, et chaque machine lance la transformation map et un reduce local dont le résultat est retourné au driver program

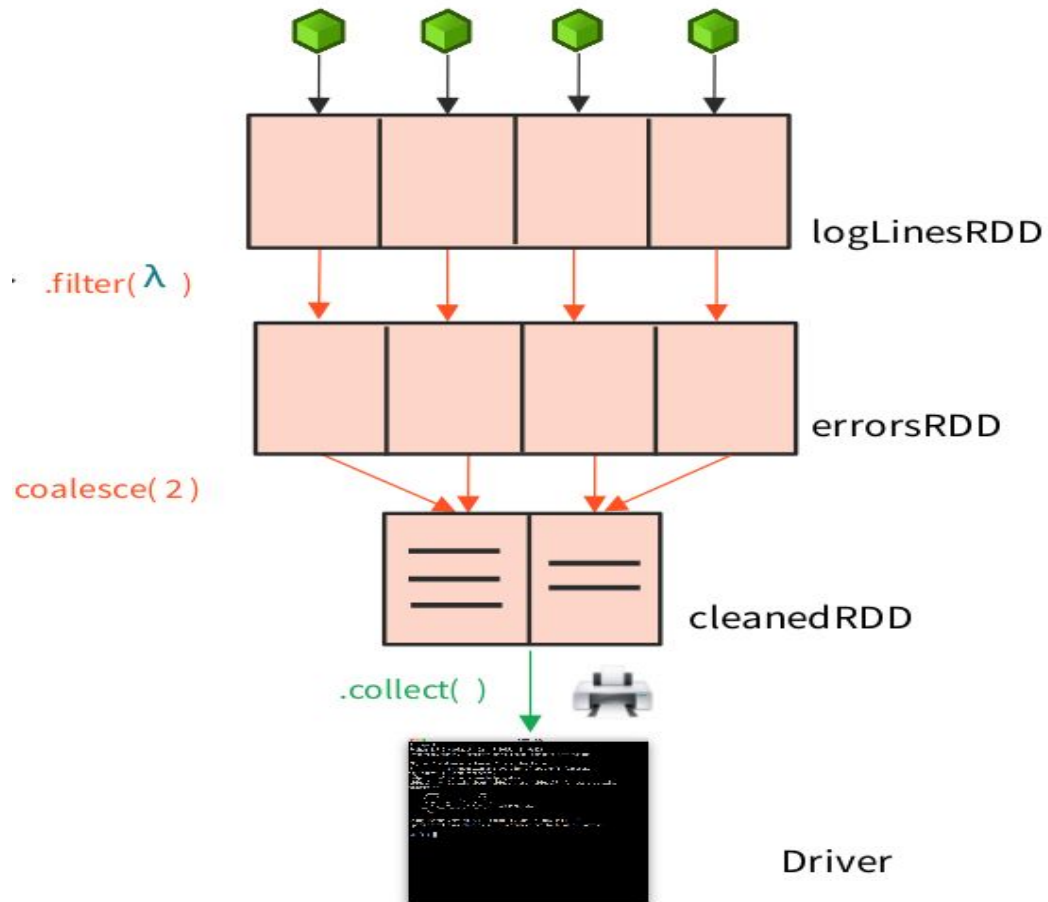
RDD Exemple (1/4)



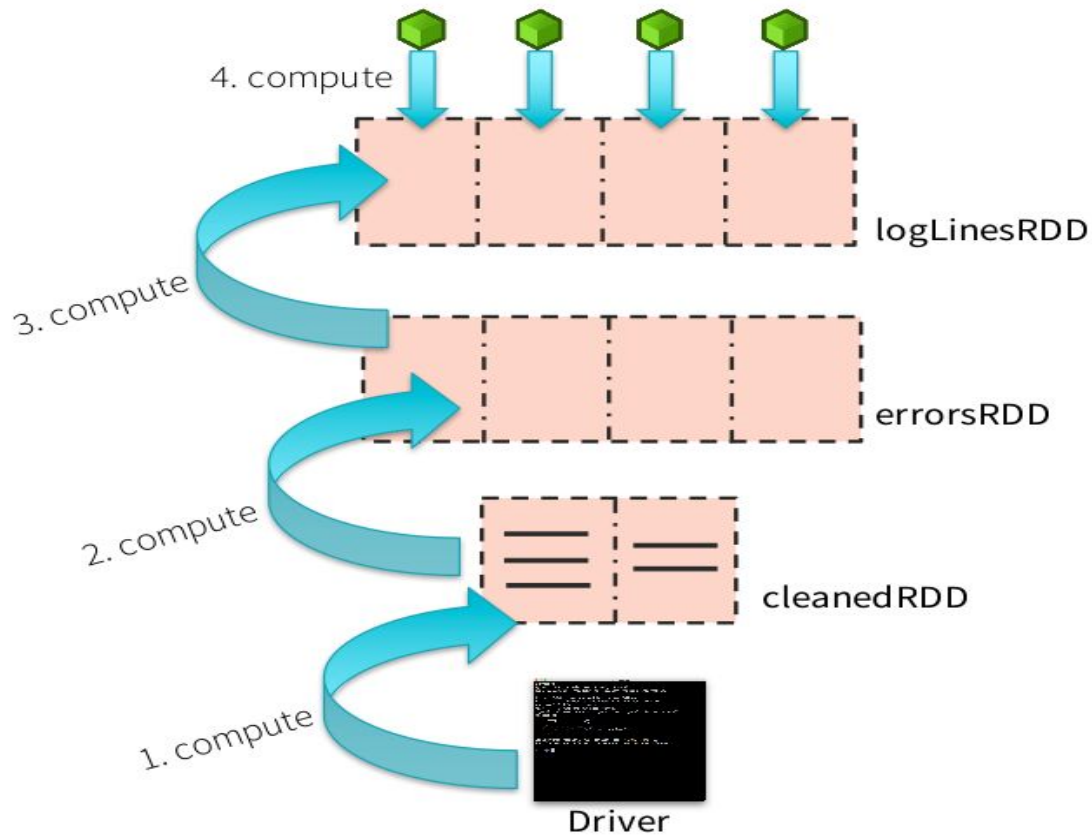
RDD Exemple (2/4)

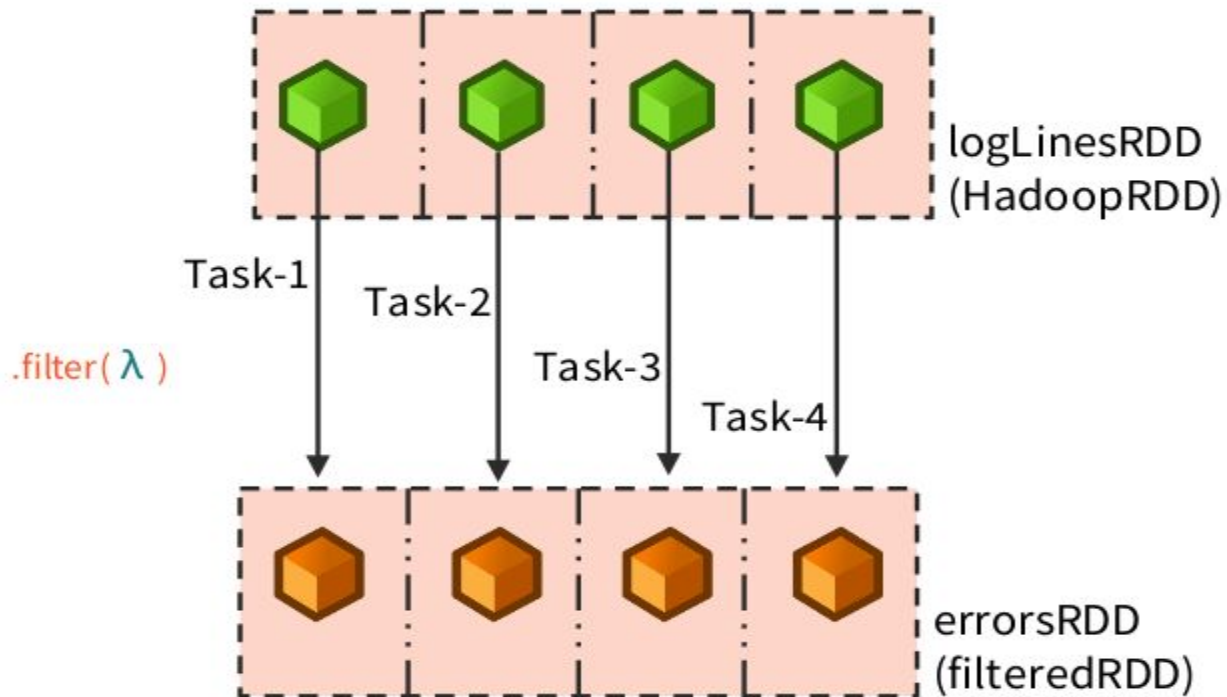


RDD Exemple (3/4)



RDD Exemple (4/4)





1. Créer des RDD d'entrée à partir de données externes ou paralléliser une collection dans le driver
2. Transformer les de manière lazy pour en définir des nouveaux ne utilisant des transformations (filter, map, ...)
3. Demander à Spark de persister (ou mettre en cache) les RDD intermédiaires qui devront être réutilisés
4. Lancer des actions telles que count() et collect() pour lancer un calcul distribué, qui est ensuite optimisé et exécuté par Spark.

- Possibilité de persister un RDD en mémoire sur tous les noeuds pour être réutiliser entre plusieurs actions
 - *Outil clé pour les algorithmes itératifs*
- Le cache de Spark est tolérant au panne
- Chaque RDD peut être persisté en utilisant différents niveaux de stockage :
 - *par défaut en mémoire*
 - *en mémoire et sur disque*
 - *en mémoire avec sérialisation*
 - *en mémoire et sur disque avec sur sérialisation*
 - *sur disque uniquement*
 - *et toutes les possibilités ci-dessus avec facteur de réplication = 2*

Niveau de stockage	Description
MEMORY_ONLY	Stocke les RDD comme des objets JAVA désérialisées dans la JVM. Si le RDD ne tient pas en mémoire, des partitions ne seront pas mis en cache et seront calculés à la volée chaque fois qu'elles sont nécessaires
MEMORY_AND_DISK	Similaire sauf que si IRDD ne tient pas en mémoire, les partitions qui ne tiennent pas seront stockées sur disque
MEMORY_ONLY_SER	Stocke les RDD comme des objets Java sérialisées
MEMORY_AND_DISK_SER	Similaire que <i>MEMORY_ONLY_SER</i> mais déverse les partitions qui ne tiennent pas en mémoire sur disque
DISK_ONLY	Stocke les partitions sur disque
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Pareil que les niveaux au dessus mais réplique chaque partition sur deux noeuds

- Fonctions anonymes pour de courts morceaux de code
- Méthodes statique dans un objet singleton :

```
object MyFunctions {  
  def func1(s: String): String = { ... }  
}  
myRdd.map(MyFunctions.func1)
```

- Possible aussi de passer une référence à une méthode d'une instance de classe, cela requiert d'envoyer l'objet qui contient cette classe avec la méthode :

```
class MyClass {  
  def func1(s: String): String = { ... }  
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }  
}
```

```
var counter = 0
var rdd = sc.parallelize(data)
// Wrong: Don't do this!!
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

- Opération découpés en tâches et envoyées aux executors
- Avant exécution, Spark calcule la closure : variable ou méthode visible par l'executor pour effectuer ses calculs sur le RDD.
- Closure est sérialisée et envoyée sur chaque executor

=> Résultat inattendu en mode cluster !!! Mais fonctionne en mode local

- Certaines opérations sont disponibles seulement pour des RDD de paires <clé, valeur>. Les plus courantes sont des opérations distribuées de “shuffle” (regroupement ou agrégation par clé)
- En scala, ces opérations sont disponibles automatiquement sur les RDD contenant des objets Tuples2

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Pour éviter les problèmes de sérialisation Java, utilisez Kryo pour réduire l'empreinte mémoire de vos objets.

Mettre **spark.serializers** à **org.apache.spark.serializer.KryoSerializer**

```
import _root_.com.esotericsoftware.kryo.Kryo
import org.apache.spark.SparkConf

class MyKryoRegistrar extends KryoSerializer {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[LogLine])
  }
}

object MyKryoRegistrar {
  def register(conf: SparkConf) {
    conf.set("spark.serializer", classOf[KryoSerializer].getName)
    conf.set("spark.kryo.registrator", classOf[MyKryoRegistrar].getName)
  }
}
```

- Réaliser un word count sur le fichier sample.txt (/user/formation-spark/sample.txt)
- <https://github.com/erwan-koffi/cours-insa/blob/master/sample.txt>
- Sauvegarder le résultat dans un fichier CSV
- Réaliser une fonction qui transforme le mot “sed” en “awk”
- Afficher les 10 mots les plus présents avec le pourcentage de densité

Spark SQL



- Traitement sur des données structurées et semi-structurées en SQL.
- Prise en charge d'un large éventail de formats de données et de systèmes de stockage
- JDBC Server intégré pour réaliser des analyses big data depuis des outils BI traditionnels.
- Plus accessible que les RDD mais moins d'optimisations possibles.
- Permettre à un plus large public d'écrire des applications Spark, au-delà des data ingénieurs.
- Meilleure performance grâce à l'optimiseur Catalyst.
- Une API unifiée quelque soit le langage (Python, Java, Scala et R)

RDD	Dataframe
<ul style="list-style-type: none">• Typage fort vérifié à la compilation• Nécessaire à certains type de logique• Beaucoup de code existant• Contrôle bas niveau de spark• Impératif	<ul style="list-style-type: none">• Utilisation de la mémoire plus faible• Prise en compte de la mémoire disponible (évite les OOM)• Tri, hash et sérialisation plus rapide• Optimisations “automatiques”• Déclaratif

Interface unifiée pour lire et écrire des données dans de nombreux formats

```
val df = sqlContext.  
  read.  
    format("json").  
    option("samplingRatio", "0.1").  
    load("/Users/spark/data/stuff.json")  
  
df.write.  
  format("parquet").  
  mode("append").  
  partitionBy("year").  
  saveAsTable("faster-stuff")
```



Interface unifiée pour lire et écrire des données dans de nombreux formats

```
val df = sqlContext.  
  read.  
    format("json").  
    option("samplingRatio", "0.1").  
    load("/Users/spark/data/stuff.json")  
  
df.write.  
  format("parquet").  
  mode("append").  
  partitionBy("year").  
  saveAsTable("faster-stuff")
```

read and **write**
functions create
new builders for
doing I/O

Interface unifiée pour lire et écrire des données dans de nombreux formats

```
val df = sqlContext.  
  read.  
    format("json").  
    option("samplingRatio", "0.1") }  
  load("/Users/spark/data/stuff.json")  
  
df.write.  
  format("parquet").  
  mode("append").  
  partitionBy("year").  
  saveAsTable("faster-stuff") }
```




Builder methods
specify:

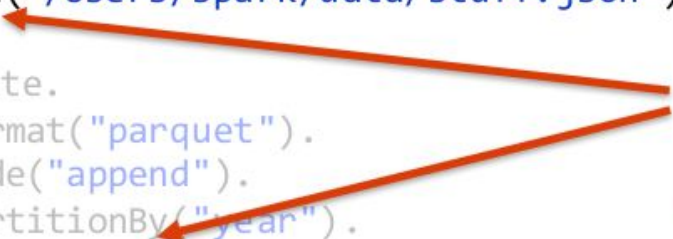
- format
- partitioning
- handling of existing data

Interface unifiée pour lire et écrire des données dans de nombreux formats

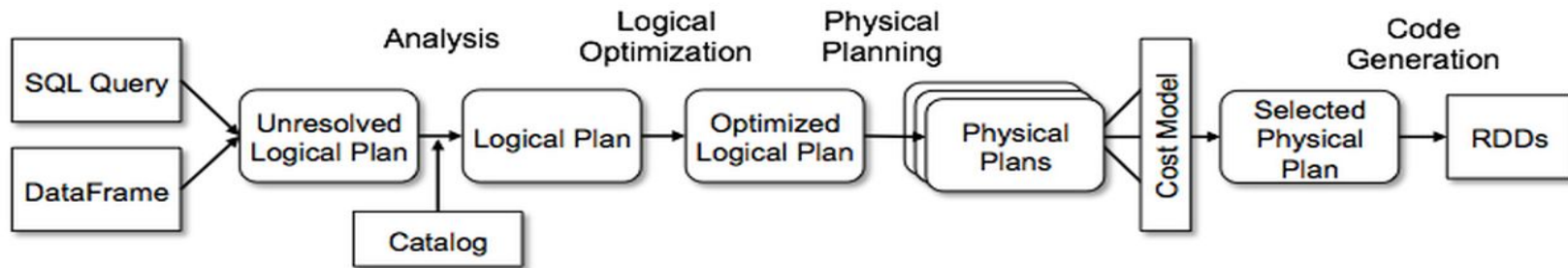
```
val df = sqlContext.  
  read.  
    format("json").  
    option("samplingRatio", "0.1").  
    load("/Users/spark/data/stuff.json")  
  
df.write.  
  format("parquet").  
  mode("append").  
  partitionBy("year").  
  saveAsTable("faster-stuff")
```



**load(...), save(...),
or saveAsTable(...)**
finish the I/O
specification



- Catalyst construit un “Unresolved Logical Plan” et le résout grâce au Catalog
- La phase “Logical Optimization” applique des optimisations à base de règle standard
- La phase “Physical Planning” génère de multiple plans et les compare en fonction de leur coût
- La phase de génération de code génère le bytecode qui sera lancé



Plan d'exécution 1 / 2

▼Details

```
== Parsed Logical Plan ==
Aggregate [count(1) AS count#25L]
  Filter (site#11 = mobile)
    Subquery pageviews_by_second
      Relation[timestamp#10,site#11,requests#12] CsvRelation(/FileStore/tables/rv3hb79r1449478143619,true,,"",null,#,PERMISSIVE,COMMONS,false,false,StructType(StructField(timestamp,StringType,true), StructField(site,StringType,true), StructField(requests,IntegerType,true)),UTF-8,false)

== Analyzed Logical Plan ==
count: bigint
Aggregate [count(1) AS count#25L]
  Filter (site#11 = mobile)
    Subquery pageviews_by_second
      Relation[timestamp#10,site#11,requests#12] CsvRelation(/FileStore/tables/rv3hb79r1449478143619,true,,"",null,#,PERMISSIVE,COMMONS,false,false,StructType(StructField(timestamp,StringType,true), StructField(site,StringType,true), StructField(requests,IntegerType,true)),UTF-8,false)

== Optimized Logical Plan ==
Aggregate [count(1) AS count#25L]
  Project
    Filter (site#11 = mobile)
      Relation[timestamp#10,site#11,requests#12] CsvRelation(/FileStore/tables/rv3hb79r1449478143619,true,,"",null,#,PERMISSIVE,COMMONS,false,false,StructType(StructField(timestamp,StringType,true), StructField(site,StringType,true), StructField(requests,IntegerType,true)),UTF-8,false)

== Physical Plan ==
TungstenAggregate(key=[], functions=[(count(1),mode=Final,isDistinct=false)], output=[count#25L])
  TungstenExchange SinglePartition
    TungstenAggregate(key=[], functions=[(count(1),mode=Partial,isDistinct=false)], output=[currentCount#28L])
      TungstenProject
        Filter (site#11 = mobile)
          Scan CsvRelation(/FileStore/tables/rv3hb79r1449478143619,true,,"",null,#,PERMISSIVE,COMMONS,false,false,StructType(StructField(timestamp,StringType,true), StructField(site,StringType,true), StructField(requests,IntegerType,true)),UTF-8,false)[timestamp#10,site#11,requests#12]

Code Generation: true
```


- Analysis: Analyse d'un logical plan pour résoudre les références
- Logical Optimization: Optimisation du plan logique
- Physical Planning: Plan physique
- Code Generation: Compilation des requêtes en bytecode

- Point d'entrée des fonctionnalités SQL

```
val spark = SparkSession.builder().appName("Application").getOrCreate()
```

- Les fonctions de conversion ainsi que la DSL peuvent être importés

```
import spark.implicits._
```

- Il est possible d'ajouter également le support de Hive et d'autres configurations.

```
SparkSession.builder().enableHiveSupport()  
    .config("spark.eventLog.enabled", "true")  
    .appName("Application").getOrCreate()
```

- Cela permet de lire et écrire les données dans Hive
- Nécessité d'indiquer le metastore par configuration dans le fichier hive-site.xml
- L'utilisateur Spark doit avoir les droit de lecture / écriture sur les fichiers du warehouse Hive.
- Utilisation des SerDes et des UDF présents dans Hive.

DataFrames = Collections distribuées de données organisée en colonne nommées

- SchemaRDD avant Spark 1.3
- Plusieurs possibilités pour les créer : RDD, Table Hive, JSON, CSV, Parquet, source de données JDBC...
- 2 possibilités pour les manipuler :
 - *Dataframes API (DSL) en Scala, Python et R*
 - *En Java il faut utiliser l'API Dataset en précisant Dataset<Row>*
 - *SQL*

```
val spark = SparkSession.builder().appName("Application").getOrCreate()

val df = spark.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()

// Select only the "name" column
df.select("name").show()

// Select people older than 21
df.filter(df("age") > 21).show()

// Count people by age
df.groupBy("age").count().show()
```

```
val spark = SparkSession.builder().appName("Application").getOrCreate()
import spark.implicits._

// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create a dataframe of Person objects
val people =
  session.sparkContext.textFile("examples/src/main/resources/people.txt").map(_.split(",
")).map(p => Person(p(0), p(1).trim.toInt)).toDF()
```

```
// Create an RDD
val people = session.sparkContext.textFile("examples/src/main/resources/people.txt")
// The schema is encoded in a string
val schemaString = "name age"
// Import Row.
import org.apache.spark.sql.Row;
// Import Spark SQL data types
import org.apache.spark.sql.types.{StructType, StructField, StringType};
// Generate the schema based on the string of schema
val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName,
  StringType, true)))
// Convert records of the RDD (people) to Rows.
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
// Apply the schema to the RDD.
val peopleDataFrame = session.createDataFrame(rowRDD, schema)
```

- Générique Load/Save Function, par défaut le parquet :

```
val df = session.read
    .load("examples/src/main/resources/users.parquet")

df.select("name", "favorite_color")
    .write
    .save("namesAndFavColors.parquet")
```

- En spécifiant la data source qui sera utilisé :

```
val df = session
    .read
    .format("json")
    .load("examples/src/main/resources/people.json")

df.select("name", "age").write
    .format("parquet").save("namesAndAges.parquet")
```

- Les opérations de sauvegarde peuvent prendre un paramètre optionnel SaveMode
- Ces SaveMode n'utilisent pas de lock et ne sont pas atomiques.

Mode	Description
SaveMode.ErrorIfExists (default)	Lors de l'enregistrement d'un dataframe à une source de données, si les données existent déjà une exception est lancée
SaveMode.Append	Lors de l'enregistrement d'un dataframe à une source de données, si les données/tables existent, le contenu du dataframe devraient être ajoutées aux données existantes
SaveMode.Overwrite	Lors de l'enregistrement d'un dataframe à une source de données, si les données/tables existent, les données existantes devraient être écrasées par le contenu du dataframe
SaveMode.Ignore	Similaire à un CREATE TABLE IF NOT EXISTS

- Il est possible de définir des fonctions pour les utiliser ensuite sur les datasets
- Les UDF de Hive sont également utilisables lorsque Spark SQL est utilisé sur une base Hive

Exemple pour le SQL:

```
session.udf.register("strLen", (s: String) => s.length())  
session.sql("SELECT strLen(name) FROM users").show()
```

Exemple avec la DSL Scala:

```
import org.apache.spark.sql.functions.udf  
  
val strLen = udf((s: String) => s.length)  
df.select(strLen($"name")).show()
```

- De la même manière, il est possible de créer des UDAF, afin de définir des agrégations custom.

- Les UDF peuvent avoir un coût.
- Essayer d'éviter d'écrire une udf quand il est possible d'utiliser d'autres moyens.

- Exemple

```
val monUDF = udf((a: String, b: String) => a == b)  
df1.join(df2, monUDF($"x", $"y"))
```



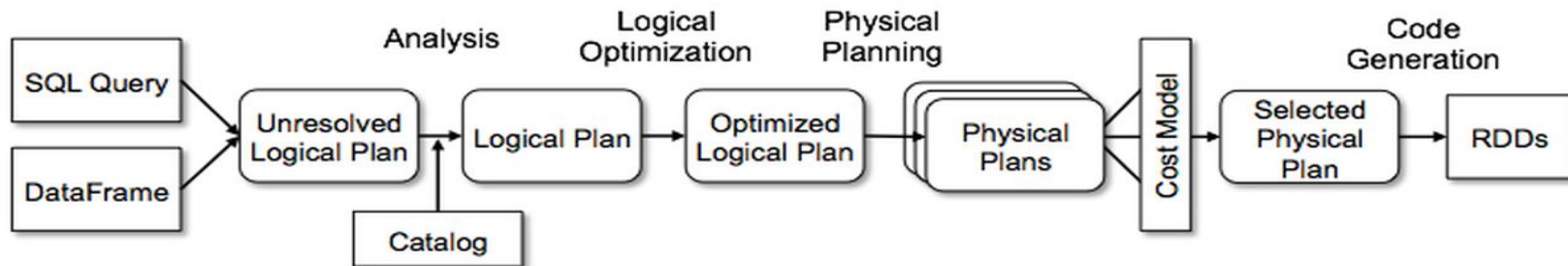
Produit cartésien:
 n^2

```
df1.join(df2, $"x" == $"y")
```

Les valeurs seront triées avant d'être comparées



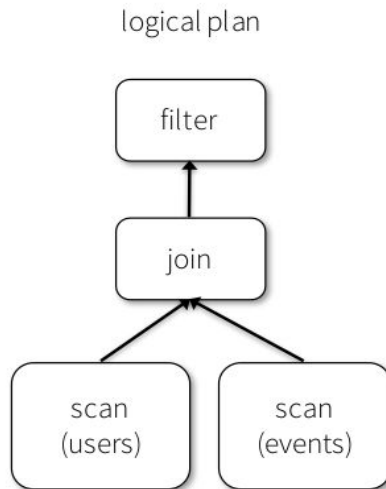
SortMergeJoin:
 $n \log(n)$



Optimisations Logiques	Plan physiques et génération du code JVM
<ul style="list-style-type: none">• Les prédicats de filtres sont gérés par la source de données. Les données invalides peuvent être ignorées.• Parquet: Skip de blocs entiers, transforme la comparaison des strings en comparaison sur les entiers (encodage dictionnaire).• RDBMS: Réduit le trafic en déléguant les prédicats.	<ul style="list-style-type: none">• Catalyst compile les opérations en un plan physique pour l'exécution et génère du bytecode JVM.• Choisis intelligemment entre les jointures en broadcast et en shuffle pour réduire l'utilisation de bande passante.• Optimisations bas niveau

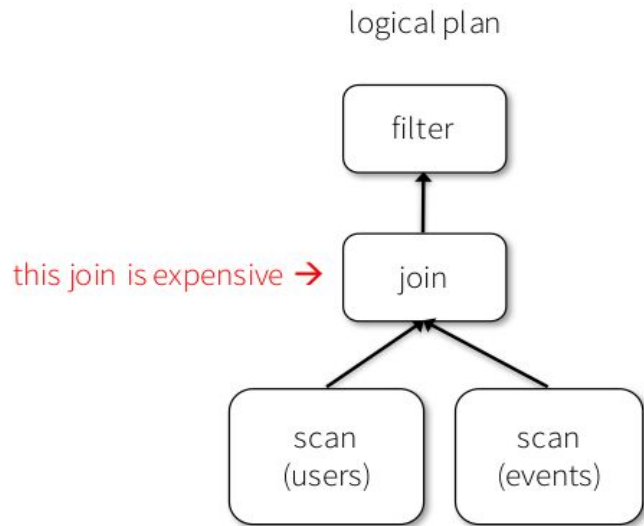
Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)  
filtered = joined.filter(events.date >= "2015-01-01")
```



Plan Optimization & Execution

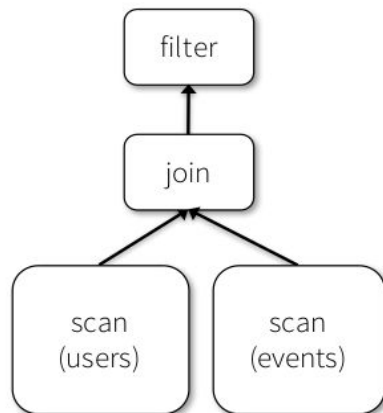
```
joined = users.join(events, users.id == events.uid)  
filtered = joined.filter(events.date >= "2015-01-01")
```



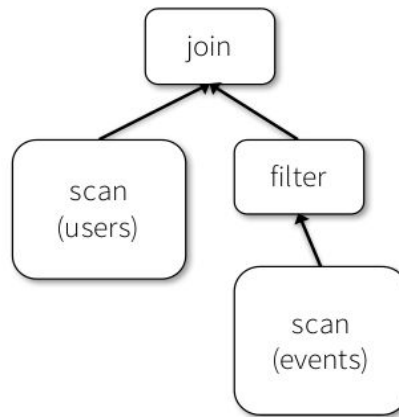
Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)  
filtered = joined.filter(events.date >= "2015-01-01")
```

logical plan

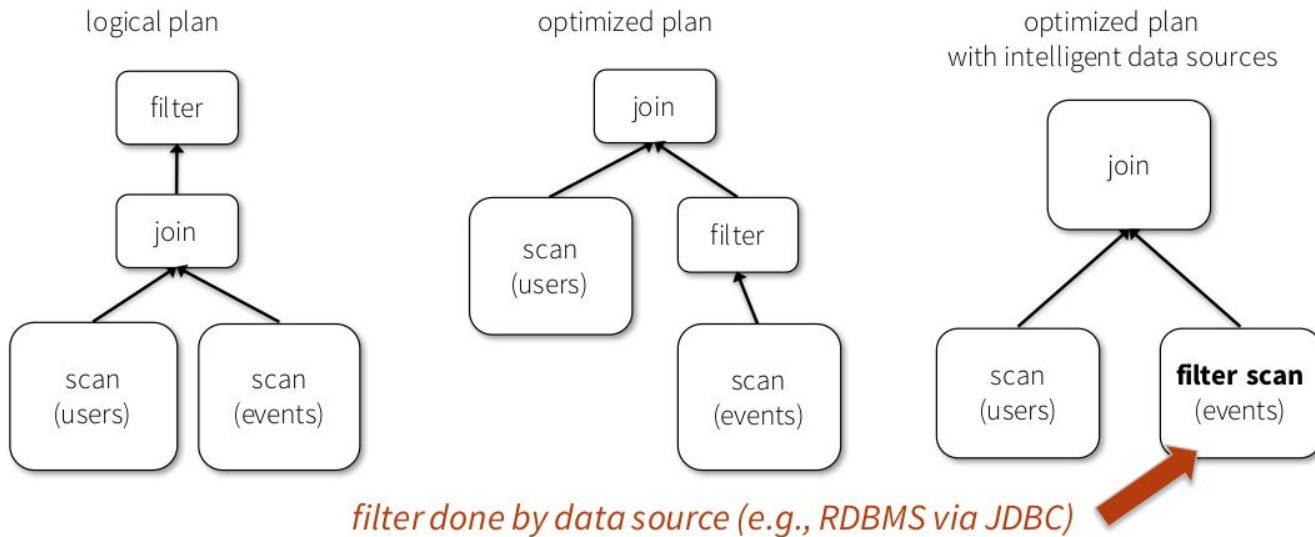


optimized plan



Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)  
filtered = joined.filter(events.date >= "2015-01-01")
```

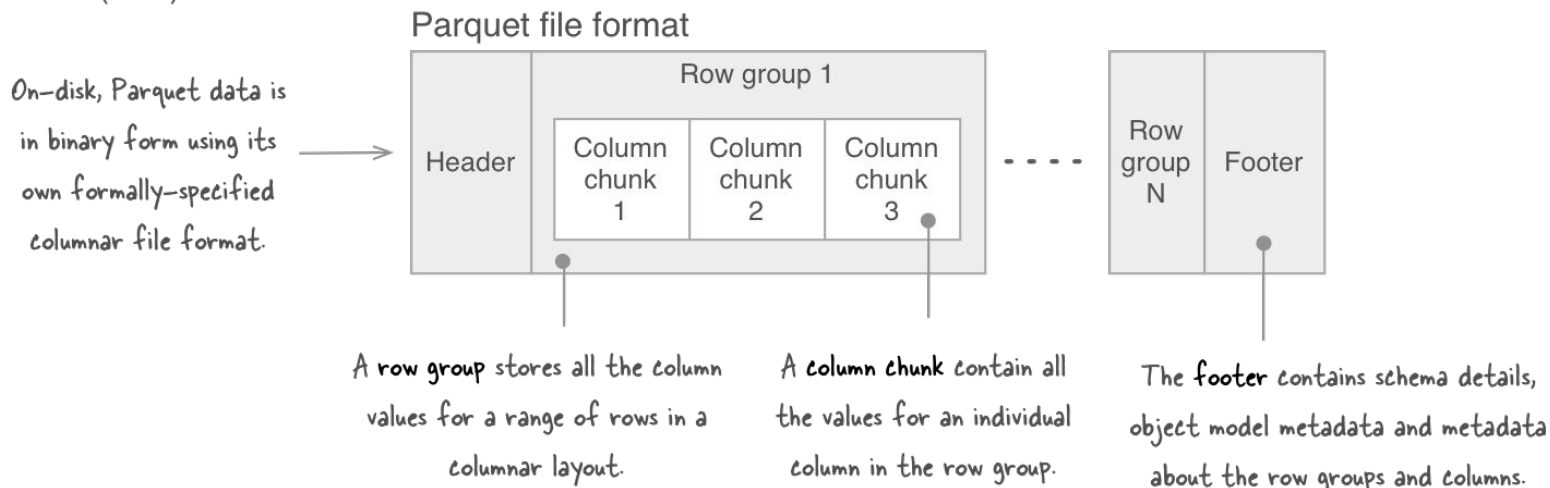




```
scala> val df3 = df.join(df2,  
  (df("first_name") === df2("firstName")) & (df("last_name") === df2("lastName"))  
scala> df3.explain()
```

```
ShuffledHashJoin [last_name#18], [lastName#36], BuildRight  
  Exchange (HashPartitioning 200)  
    PhysicalRDD [first_name#17,last_name#18,gender#19,age#20L], MapPartitionsRDD[41]  
at applySchemaToPythonRDD at NativeMethodAccessorImpl.java:-2  
  Exchange (HashPartitioning 200)  
    PhysicalRDD [firstName#35,lastName#36,medium#37], MapPartitionsRDD[118] at  
executedPlan at NativeMethodAccessorImpl.java:-2
```


Storage format (disk)



Optimisation au format parquet

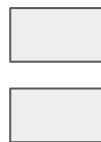
a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Column projection



a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Predicate push-down

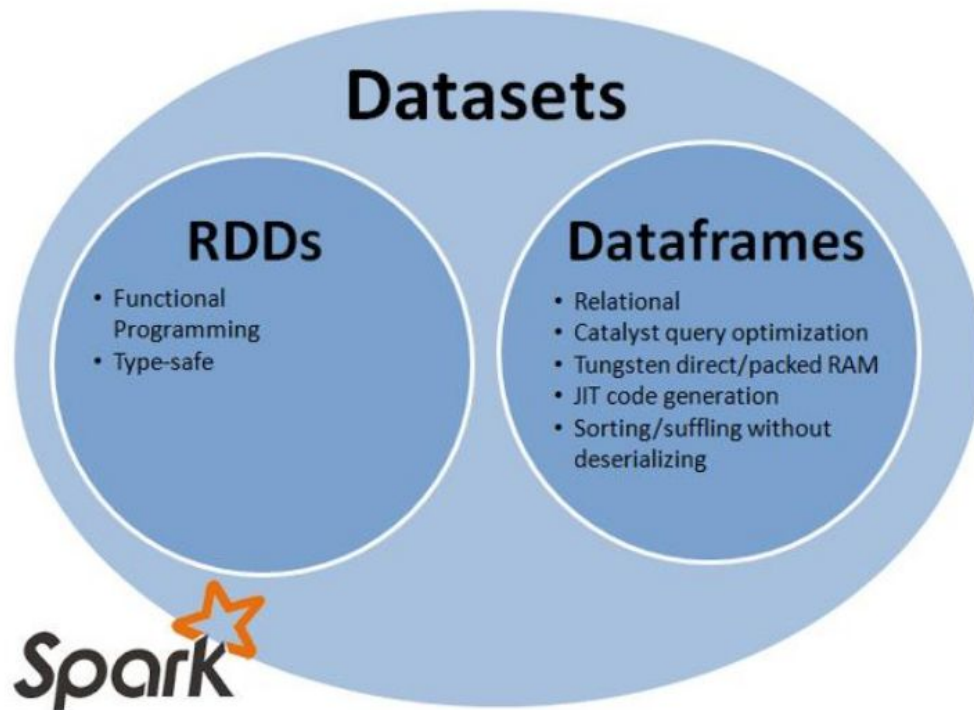


a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Only the data you need

- Converti un plan logique en plan physique
- Manipule et optimise l'arbre de transformations:
 - élimination des sous-requêtes
 - remplacement des expressions par des constantes
 - suppression des filtres inutiles
 - prédicats pushdowns

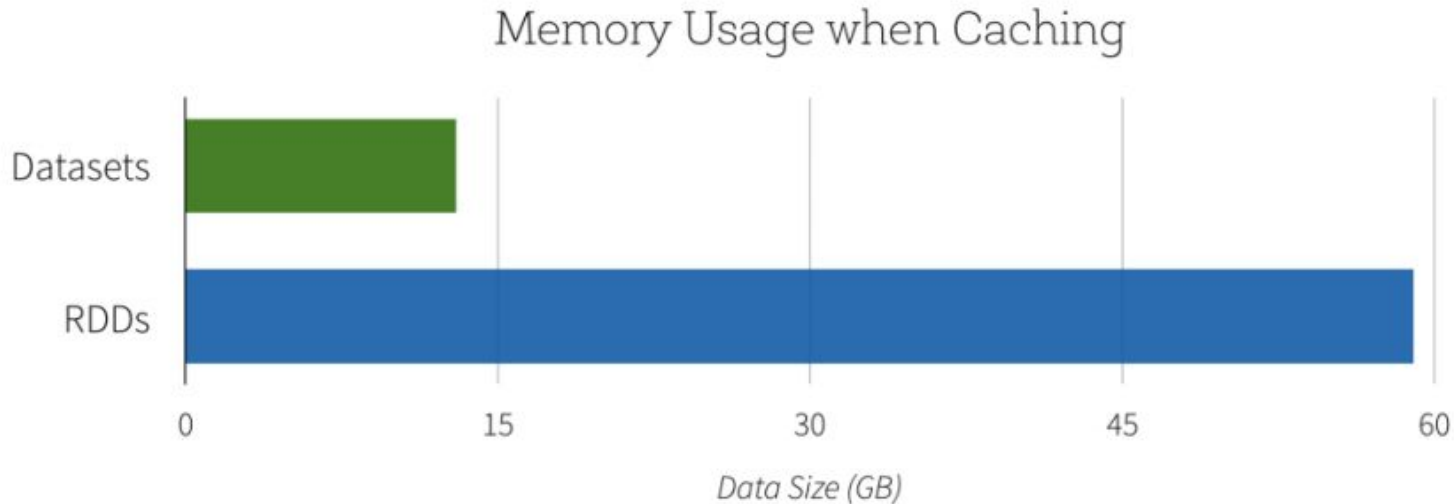
- Les Dataframes ne sont pas automatiquement repartitionnés de manière optimale
- Durant un shuffle sur les Dataframes, Spark SQL n'utilisera que la propriété `spark.sql.shuffle.partitions` pour choisir le nombre de partitions dans les RDD sous-jacents.
- La structure des objets lors du shuffle des Dataframes change lorsque le nombre de partitions est supérieur à 2000.
- Typage dynamique



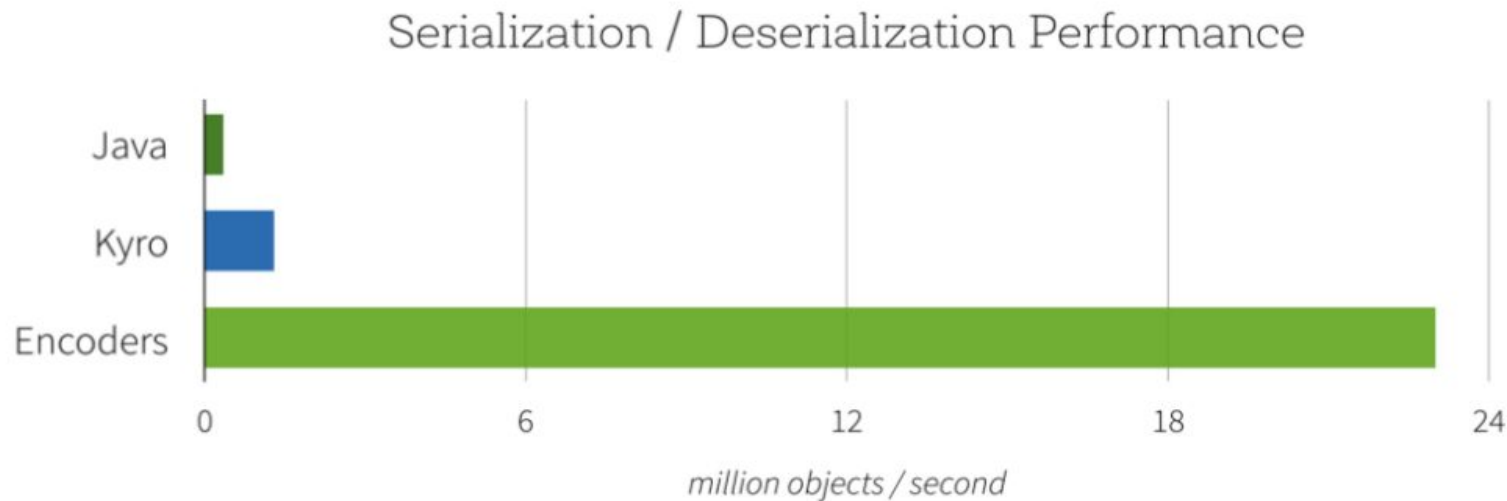
- Interface typée pour les DataFrames / Tungsten
- Utilisation d'un encoder pour passer de dataset à dataframe.
- Permet d'utiliser des lambdas
- N'est accessible qu'en Scala, Java

- Typed interface over DataFrames / Tungsten
- `case class` Person(name: String, age: Int)
- `val` dataframe = read.json("people.json")
- `val` ds: Dataset[Person] = dataframe.as[Person]
- ds.filter(p => p.name.startsWith("M"))
 .groupBy("name")
 .avg("age")

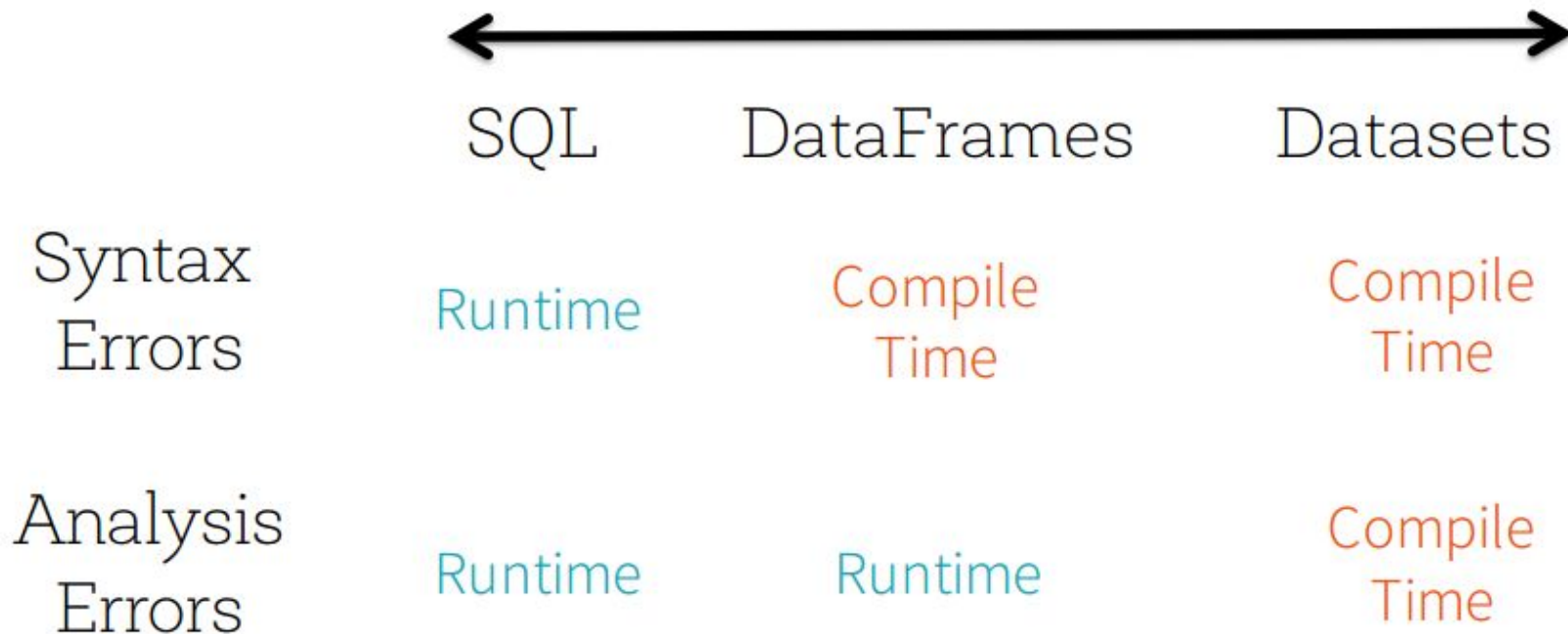
Space Efficiency



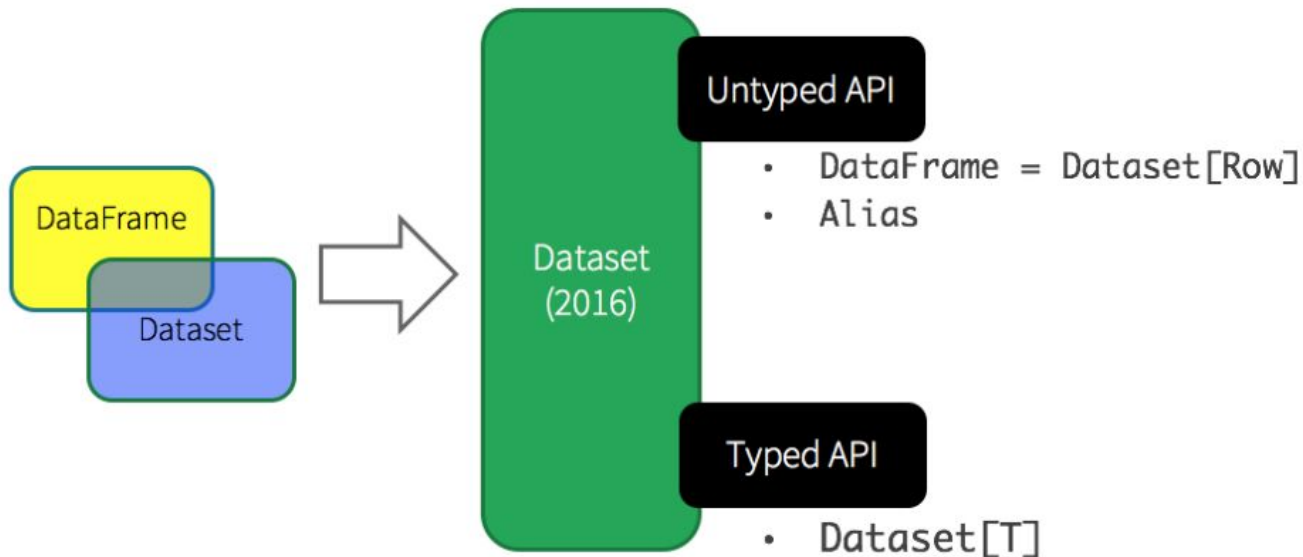
Datasets: Lightning-fast Serialization with Encoders



Structured APIs In Spark



Unified Apache Spark 2.0 API



Type-safe: operate
on domain objects
with compiled
lambda functions

```
val df = spark.read.json("people.json")

// Convert data to domain objects.
case class Person(name: String, age: Int)
val ds: Dataset[Person] = df.as[Person]
ds.filter(_.age > 30)

// Compute histogram of age by name.
val hist = ds.groupBy(_.name).mapGroups {
  case (name, people: Iter[Person]) =>
    val buckets = new Array[Int](10)
    people.map(_.age).foreach { a =>
      buckets(a / 10) += 1
    }
    (name, buckets)
```

- Depuis Spark 2.0 un Dataframe est un Dataset[Row]
- Pour transformer un dataframe en dataset, il suffit d'utiliser la syntaxe suivante:

```
val ds = df.as[MaClasse]
```

```
events =  
  sc.read.json("/logs")  
  
stats =  
  events.join(users)  
    .groupBy("loc", "status")  
    .avg("duration")  
  
errors = stats.where(  
  stats.status == "ERR")
```

DataFrame API

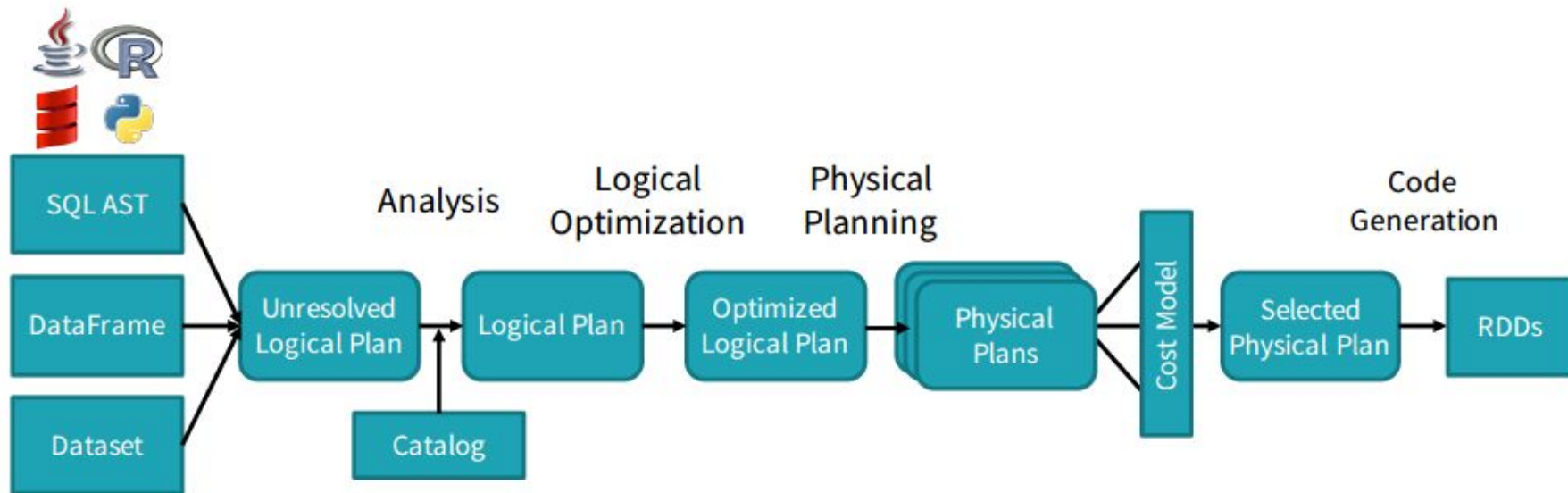


Optimized Plan



```
while(logs.hasNext) {  
  e = logs.next  
  if(e.status == "ERR") {  
    u = users.get(e.uid)  
    key = (u.loc, e.status)  
    sum(key) += e.duration  
    count(key) += 1  
  }  
}  
...
```

Specialized Code



DataFrames, Datasets and SQL
share the same optimization/execution pipeline

- **Primitifs:**
 - Byte, Short, Integer, Long, Float, Double, Decimal, String, Binary, Boolean, Timestamp, Date
- **Array[Type]:**
 - Collections à taille variable
- **Struct:**
 - Nombre fixe de colonnes avec des types stricts
- **Map[Type, Type]:**
 - Association à taille variable

Encoders map columns
to fields by name

{ JSON }



JDBC



elasticsearch.



```
{  
  "name": "Michael",  
  "zip": "94709"  
  "languages": ["scala"]  
}
```



```
case class Person(  
  name: String,  
  languages: Seq[String],  
  zip: Int)
```

- Réaliser un word count sur le fichier sample.txt
- Sauvegarder le résultat dans un fichier CSV
- Réaliser une udf qui transforme le mot “sed” en “awk”
- Afficher les 10 mots les plus présents avec le pourcentage de densité
- Utiliser des Datasets à la place des Dataframes

Spark ML

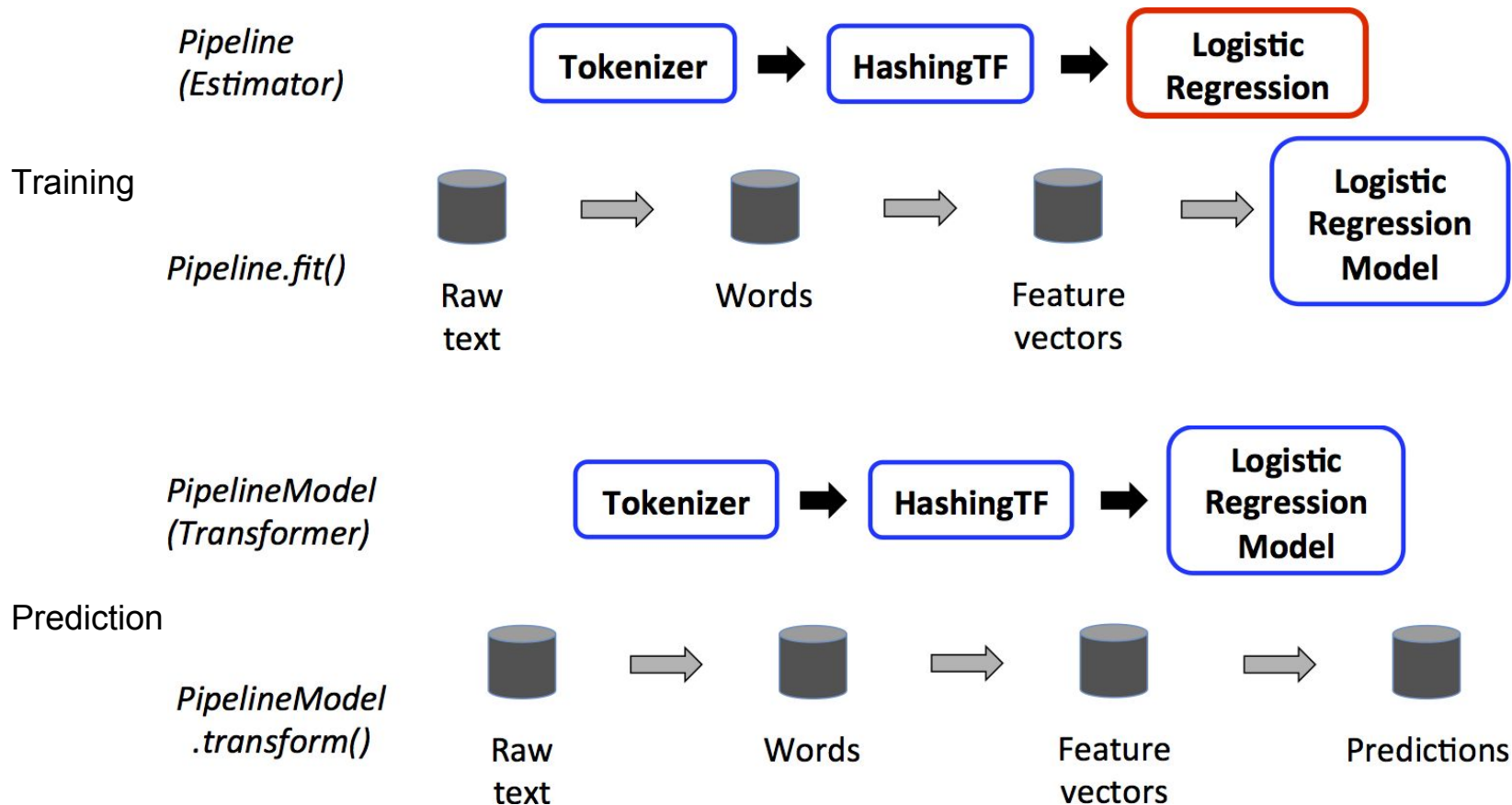


- Fournit plusieurs outils pour Spark
 - Algorithmes de machine learning (classification, regression, clustering...)
 - Traitements des Features
 - Constructions de pipelines de machine learning
 - Persistance des algorithmes, modèles et pipelines
 - Outils d'algèbre linéaire, statistiques...

- **spark.ml / SparkML**
 - API haut niveau fonctionnant sur les DataFrames.
 - Base pour tous les futurs développements.
 - Sera complète aux alentours de Spark 2.3
- **spark.mllib / SparkMLlib**
 - API bas niveau utilisant les RDD.
 - Pas de nouvelles features dans les nouvelles versions.
 - Sera supprimé en Spark 3.0

- Inspiré de scikit-learn.
- Les datasets utilisent l'API DataFrame de Spark SQL.
- Les Transformer transforment un DataFrame (i.e. un modèle qui transforme un DataFrame de features en DataFrame de prédictions).
- Les Estimator produisent des Transformers (i.e. un algorithme d'apprentissage qui produit un modèle).
- Un Pipeline qui permet de chaîner plusieurs Transformers et Estimators.
- Les Parameters qui est une API commune pour configurer les Transformers et les Estimators.

Pipelines



- Il est possible de sauvegarder un pipeline sur le disque.

```
pipeline.write.overwrite().save("/tmp/mon-pipeline")
```

Utilise la même API que pour les DataFrames.

- Pipeline quelque soit son type peut être rechargé.

```
PipelineModel.load("/tmp/spark-logistic-regression-model")
```


Charger le fichier `/user/formation-spark/dataset.csv`

- Effectuer une régression sur le prix
- Appliquer une classification sur ce dataset pour déterminer le browser
 - Mettre en place un pipeline pour sélectionner des features et l'application du modèle:
- Sauvegarder ce modèle sur le did
- Appliquer un autre modèle de classification que celui choisi précédemment et le sauvegarder sur le Hdfs

Documentation MLlib: <https://spark.apache.org/docs/2.1.2/ml-guide.html>

- La sélection de modèle se fait en utilisant deux outils:
 - CrossValidator
 - TrainValidationSplit
- Ces outils ont besoin:
 - Un Estimator qui sera “Tuné”
 - D’une Map de Parameter
 - Un Evaluator, la métrique pour vérifier la qualité du modèle.
- La liste des Evaluators est la suivante:
 - RegressionEvaluator
 - BinaryClassificationEvaluator (pour les classifications binaires)
 - MulticlassClassificationEvaluator

- À l'aide d'une ParamGrid améliorer les résultats des deux pipelines précédents.
- Effectuer sur une régression sur le prix
- Reprendre le fichier `sample.txt` Et appliquer une régression pour prédire l'enchaînement des mots

Documentation param: <https://spark.apache.org/docs/2.1.2/ml-tuning.html>

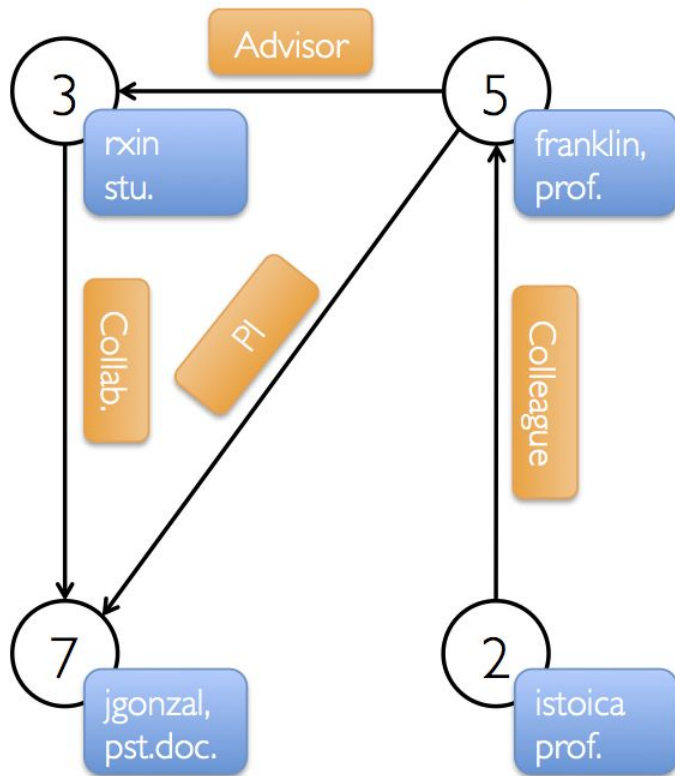
- La profondeur maximale d'un arbre de décision est de 30
- Il n'est possible d'utiliser le GBT que pour des classifications binaires

GraphX



- GraphX permet d'effectuer des calculs distribués sur les graphes à l'aide de Spark.
- Il s'agit d'une abstraction au dessus des RDD.
- GraphX embarque également un ensemble d'algorithmes sur les graphes.
- GraphX dispose également d'une implémentation de Pregel. *(Framework de calcul sur les Graphe de Google)*

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrclId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

// Assume the SparkContext has already been constructed

val sc: **SparkContext**

// Create an RDD for the vertices

val users: **RDD**[(**VertexId**, (**String**, **String**))] =
sc.parallelize(**Array**((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
 (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges

val relationships: **RDD**[**Edge**[**String**]] =
sc.parallelize(**Array**(**Edge**(3L, 7L, "collab"), **Edge**(5L, 3L, "advisor"),
 Edge(2L, 5L, "colleague"), **Edge**(5L, 7L, "pi")))

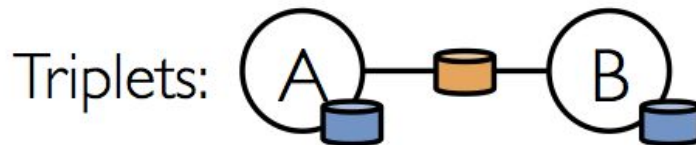
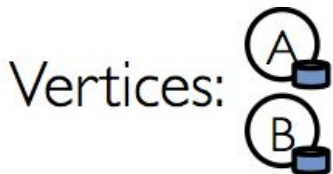
// Define a default user in case there are relationship with missing user

val defaultUser = ("John Doe", "Missing")

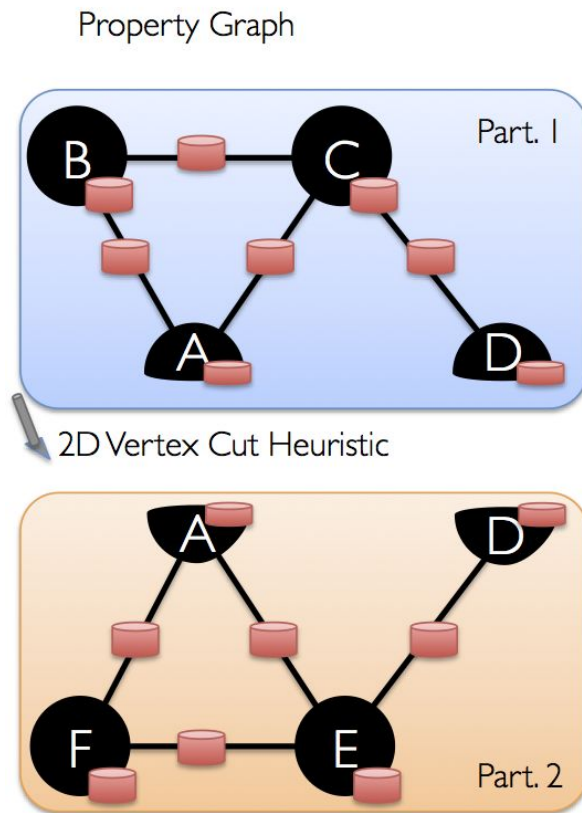
// Build the initial Graph

val graph = **Graph**(users, relationships, defaultUser)

- Un graphe est composé de deux rdd:
 - EdgeRDD[T] représentant les arêtes du graphe.
 - VertexRDD[E] représentant les noeuds du graphe.
- Les noeuds doivent tous être du même type.
- Le graphe peut être représenté en triplet pour décrire les relations entre les noeuds:



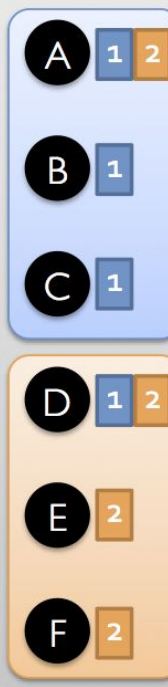
Découpage des graphes



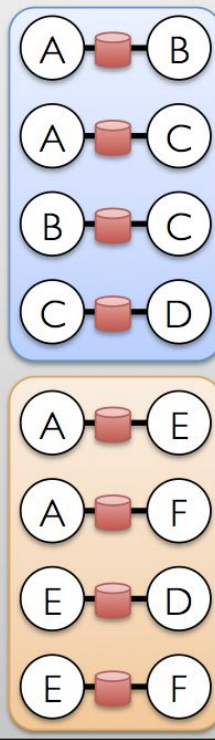
Vertex Table
(RDD)



Routing
Table
(RDD)



Edge Table
(RDD)



Les algorithmes suivant sont embarqués par GraphX

- **PageRank**: Un algorithme de page rank pour déterminer l'importance de chaque noeud du graphe
- **Connected Components**: Associe l'id du noeud avec le moins de connexion au groupe de composants
- **Triangle Counting**: Détermine le nombre de triangles de chaque noeud