

Introduction à Map Reduce et à Scala

Master DAC – Bases de Données Large Echelle
Mohamed-Amine Baazizi
baazizi@ia.lip6.fr
Octobre 2017

Organisation de BDLE

- Partie 1 : BD multidimensionnelles
- *Partie 2 : MR et traitements sur Spark*
 - *Introduction MapReduce, Spark et Scala*
- Partie 3 : Systèmes large échelle
- Partie 4 : Analyse de données graphes

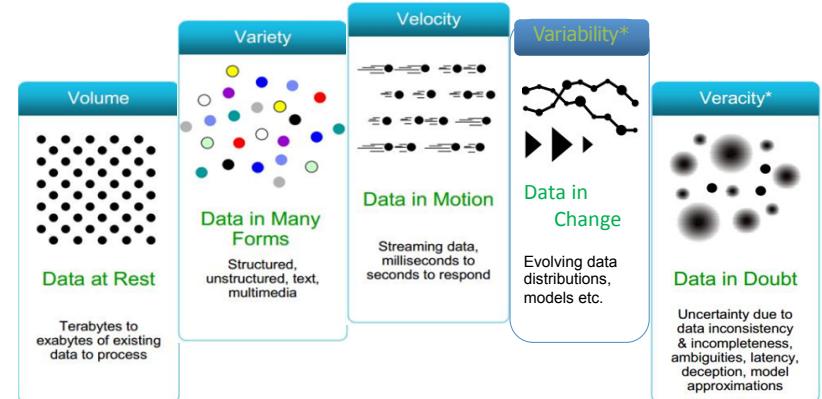
2

Contexte Big Data

- Nouvelles applications
 - Essor du web, indexation large volume de données
 - Réseaux sociaux, capteurs, GPS
 - Données scientifiques, séquençage ADN
- Analyses de plus en plus complexes
 - Moteurs de recherche
 - Comportement des utilisateurs
 - Analyse données médicales

3

Caractéristiques du big data



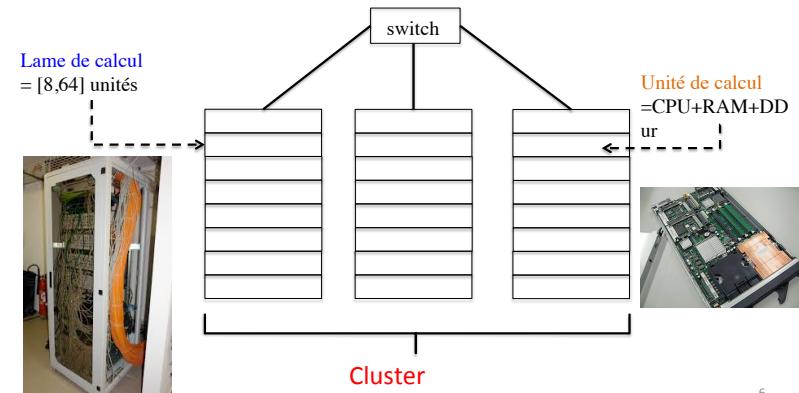
4

Relever le défi big data

- Systèmes distribués type cluster
 - à base de machines standard (*commodity machines*)
 - extensibles à volonté (architecture RAIN)
 - faciles à administrer et tolérants aux pannes
- Modèle de calcul distribué Map Reduce
 - calcul massivement parallèle
 - abstraction de la parallélisation (pas besoin de se soucier des détails sous-jacents)
 - plusieurs implantations (Hadoop, Spark, Flink...)

5

Architecture d'un cluster



6

Origine du modèle Map Reduce

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection C
- *Map* ($f: T \Rightarrow U$), f unaire : appliquer f à chaque élément de C

$$\begin{array}{|c|c|c|c|} \hline 12 & 4 & 12 & 18 \\ \hline \end{array} \xrightarrow{f(x)=x/2} \begin{array}{|c|c|c|c|} \hline 6 & 2 & 6 & 9 \\ \hline \end{array}$$

Propriétés : la dimension de C est préservée
le type en entrée peut changer

7

Origine du modèle Map Reduce

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection C
- *Reduce* ($g: (T,T) \Rightarrow T$), g binaire
 - agrégater les éléments de C deux à deux

$$\begin{array}{|c|c|c|c|} \hline 6 & 2 & 6 & 9 \\ \hline \end{array} \xrightarrow{g = '+'} \begin{array}{|c|} \hline 23 \\ \hline \end{array}$$

Propriétés : réduit la dimension de n à 1
le type en sortie identique à celui en entrée

8

Adaptation pour le big data

- Type de données
 - logs de connections, transactions, interactions utilisateurs, texte, images
 - structure homogène (schéma implicite)
- Type de traitements
 - Aggrégations (count, min, max, avg) → group by
 - Autres traitements (indexation, parcours graphes, Machine Learning)

9

Map Reduce pour le big data

- Les données en entrée sont des n-uplets : identifier attribut de regroupement (appelé clé, à ne pas confondre avec notion de clé)
 - *Map (f: T=>(k,U)).f uneire*
 - produire une paire (clé, val) pour chaque val de C
-
- $f(x)=\text{Proj}_{2,4}(x)$
- | | | |
|-------------------|--------------------|--------------------|
| <7,2010,04,27,75> | <12,2009,01,31,78> | <41,2009,03,25,95> |
| (2010, 27) | (2009, 31) | (2009, 25) |
- Regrouper les paires ayant la même clé pour obtenir (clé, [list-val])
- | | |
|------------------|--------------|
| (2009, [25, 31]) | (2010, [27]) |
|------------------|--------------|

10

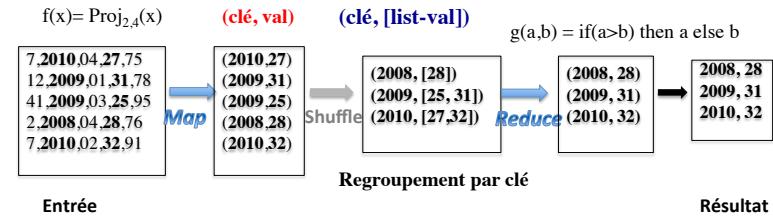
Map Reduce pour le big data

- *ReduceByKey (g: (T,T)=>T), g binaire*
 - pour chaque (clé, [list-val]) produit (clé, g([list-val]))
-
- $g(a,b) = \text{if}(a>b) \text{ then } a \text{ else } b$
- | | |
|------------------|--------------|
| (2009, [25, 31]) | (2010, [27]) |
| (2009, 31) | (2010, 27) |
- **Important :** g doit être **commutatif** et **associatif** car ordre de traitement des éléments de C non prescrit

11

Map Reduce : Exemple

- Entrée : n-uplets (station, année, mois, temp, dept)
- Résultat : select année, Max(temp) group by année



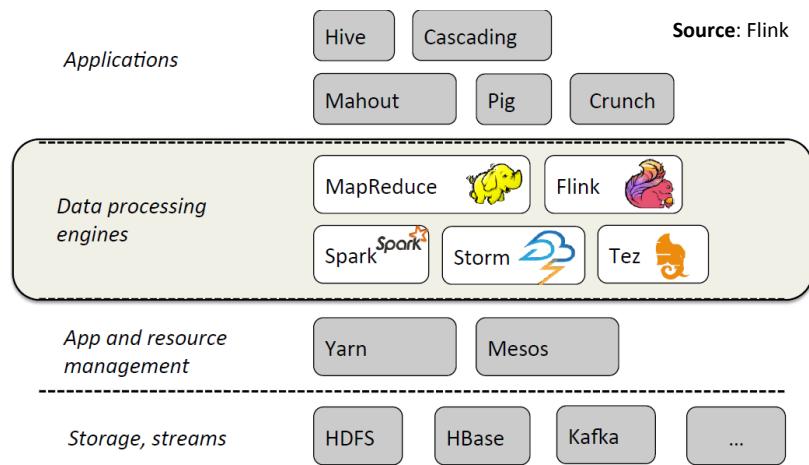
12

Plateformes Map Reduce

- Traitement distribué
 - Hadoop MapReduce (Google , 2004)
 - Spark (projet MPLab, U. Stanford , 2012)
 - Flink (projet Stratosphere, TU Berlin, 2009)
- Stockage
 - Hadoop FS, Hbase, Kafka
- Scheduler
 - Yarn, Mesos
- Systèmes haut niveau
 - Pig, Hive, Spark SQL

13

Open Source Big Data Landscape



Hadoop Map Reduce

- Introduit par Google en 2004
- Répondre aux trois principales exigences
 - Utiliser cluster de machines standards
 - extensibles à volonté (architecture RAIN)
 - faciles à administrer et tolérants aux pannes
- Ecrit en Java. Utilisation autre langages possible
- Plusieurs extensions
 - Pig et Hive pour langage de haut niveau
 - HaLoop (traitement itératif), MRShare (optimisation)

15

Critique de Map Reduce

- Traitement complexes = performances dégradées
 - Traitement complexe = plusieurs étapes
 - Solution naïve : matérialiser résultat de chaque étape
 - avantages : reprise sur panne performante
 - inconvénients : coût élevé d'accès au disque
 - Solution optimisée : pipelining et partage
- Quid des traitements itératifs - interaction avec l'utilisateur

16

Spark

- Résoudre les limitations de Map Reduce
 - Matérialisation vs maintien en mémoire centrale
 - Traitement en lot vs interactivité (Read Execute Print Loop)
- *Resilient Distributed Dataset (RDD)*
 - collection logique de données distribuées sur plusieurs nœuds
 - 1 RDD référence données sur disque (HDFS), donnée en mémoire centrale ou autre RDD dont elle dépendent
 - traitement gros granule (pas de modification partielle)
 - tolérance aux pannes par réexécution d'une chaîne de traitement
- Réutilisation de certains mécanismes de Map Reduce
 - HDFS pour le stockage des données et résultat
 - Similitude des modèles d'exécution

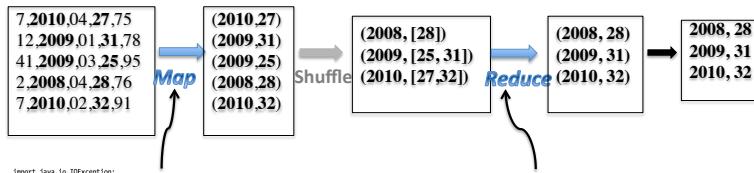
17

Pour ce cours

- Choix du système : Spark
 - Un *framework* assez complet, en continue évolution
 - Système interactif et de production à la fois
- Choix du langage hôte : Scala
 - langage natif de Spark, élégant (car fonctionnel), en pleine expansion, concis

18

Programmer en Map Reduce



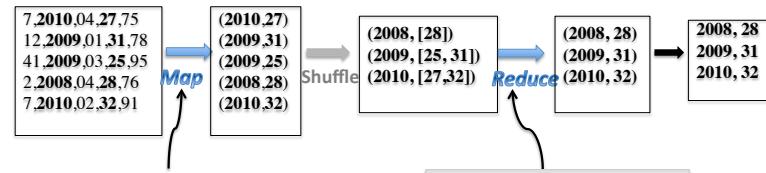
```

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, IntWritable> {
  private static final int MISSING = 9999;
  @Override
  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    String[] lineArr = line.split(",");
    int airTemperature;
    if (line.charAt(0) == '+') { // parent doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(1, 9));
    } else {
      airTemperature = Integer.parseInt(line.substring(0, 9));
    }
    String quality = line.substring(9, 10);
    if (airTemperature > MISSING && quality.matches("(0|1|2|3)")) {
      context.write(key, new IntWritable(airTemperature));
    }
  }
}
  
```

Java

19

Programmer en Spark



reduceByKey((a,b)=>if (a>b)a else b)

Scala

20

INTRODUCTION À SCALA

21

Avantages de Scala

- Compatibilité avec Java
 - compilation pour JVM, types de base de Java (Int, float, ..)
- Syntaxe concise
 - Un prog. Scala = 1/2 lignes d'un prog. Java
- Haut niveau d'abstraction
 - possibilité de cacher des détails à l'aide d'interface
- Typage statique
 - mécanisme d'inférence de types

23

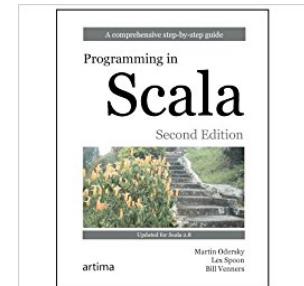
Scala : rapide tour d'horizon

- Langage orienté-objet et fonctionnel à la fois
 - Orienté objet : valeur → objet, opération → méthode
Ex: l'expression 1+2 signifie l'invocation de la méthode '+' sur des objets de la classe Int
 - Fonctionnel
 - Les fonctions se comportent comme des valeurs : peuvent être retournées ou passées comme arguments
Ex: Map(x=>f(x)) avec f(x) = x/2
 - Les structures de données sont immuables (*immutable*) : les méthodes n'ont pas d'effet de bord, elles associent des valeurs résultats à des valeurs en entrée
Ex: c=[2, 4, 6] c.Map(x=>f(x)) produit une nouvelle liste [1, 2, 3]

22

Scala : en quelques diapos

- Prise en main
- Types et opérations de base
- Structures de contrôle
- Types complexes
- Fonctions d'ordre supérieur



Référence bibliographique

M. Odersky, L. Spoon, B. Venners. *Programming in Scala*. 2nd Edition. 2012

https://booksites.artima.com/programming_in_scala_2ed

24

Prise en main

- Mode interactif

```
$ spark-shell
...
scala>
```

Manipulations de base

```
scala> 1+2
res0: Int = 3

scala> res0+3
res1: Int = 6
```

res0	la valeur calculée
:Int	le type inféré
=3	la valeur calculée

25

Prise en main

- Valeurs vs variables

- les valeurs sont immuables contrairement aux variables

```
scala> val n=1+10
n: Int = 11
```

```
scala> n=n+1
<console>:12: error: reassignment
          to val
                  n=n+1
                  ^
scala> val n=12
n: Int = 12
```

```
scala> m=m+1
m: Int = 11
```

```
scala> var m=10
m: Int = 10
```

possibilité de redéfinir une valeur

26

Prise en main

- Définition des fonctions

```
scala> def max(x: Int, y: Int): Int = if (x > y) x else y
max2: (x: Int, y: Int)Int

scala> max(1,3)
res3: Int = 3

scala> max(max(1,2),3)
res6: Int = 3
```

Le type retour inféré automatiquement sauf pour les fonctions récursives
Type par défaut Unit : correspond à void en Java

```
scala> def bonjour() = println ("bonjour")
bonjour: ()Unit
```

27

Types et opérations de base

Table 5.1 · Some basic types

Value type	Range
Byte	8-bit signed two's complement integer (-2 ⁷ to 2 ⁷ - 1, inclusive)
Short	16-bit signed two's complement integer (-2 ¹⁵ to 2 ¹⁵ - 1, inclusive)
Int	32-bit signed two's complement integer (-2 ³¹ to 2 ³¹ - 1, inclusive)
Long	64-bit signed two's complement integer (-2 ⁶³ to 2 ⁶³ - 1, inclusive)
Char	16-bit unsigned Unicode character (0 to 2 ¹⁶ - 1, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

- Opérateurs arithmétiques : + - * / %
- Opérateurs logiques : && || !
- Opérateurs binaires ...
- Conversions : toInt toDouble toLowerCase toUpperCase
 - à explorer en mode interactif

28

Types et opérations de base

- Conversions (exploration en mode interactif)

```
scala> val v = 124
v: Int = 124
scala> v.to
Taper TAB
  to          toChar     toFloat    toLong      toShort
  toBinaryString toDegrees  toHexString  toInt      toString
  toByte        toDouble   toRadians
scala> v.toInt
Taper TAB
  def toInt: Int
```

- Utiliser ce mode pour accéder aux méthodes associées à une valeur/variable

29

Structures de contrôle

- Cinq structures : *if*, *while*, *for*, *try* et *match*
- Retournent une valeur (paradigme fonctionnel)
- Similitude avec la syntaxe de Java pour if, while
- Aspects avancés (générateurs, pattern matching)

30

Structures de contrôle : exemples

- if*

```
scala> val chaine = "abcde"
scala> val longueur =
  if (chaine.length %2 ==0) "pair"
  else "impair"
longueur: String = impair
```

- While*

– à éviter car style impératif

- for*

– permet d'itérer sur des collections sans se soucier de l'index
– **for clauses yield body** pour retourner une valeur
– imbrication de plusieurs *for* possible

31

Structures de contrôle : exemples

- Exemple *for*

```
scala> val wdays = List("Mon", "Tue", "Wed", "Thu", "Fri")
wdays: List[String] = List(Mon, Tue, Wed, Thu, Fri)

scala> for (d <- wdays) print(d +"\t")
MonTue WedThu Fri

scala> for (d <- wdays) yield (d +"\t")
res2: List[String] = List("Mon ", "Tue ", "Wed ", "Thu ", "Fri ")

scala> for { d <- wdays;  md <- 1 to 4 } yield d + md
res5: List[String] = List(Mon1, Mon2, Mon3, Mon4, Tue1, Tue2, ...)
```

A éviter car programmation impérative!

32

Structures de contrôle : exemples

- *match*

- branchement conditionnel à n alternatives (*switch* en Java)
- à la base du *pattern matching*

Syntaxe :

```
var match {
  case val0 => res0
  case val1 => res1
  ...
  case _ => res_default
}
```

```
scala> def verif(age : Int) = age match {
    case 25 => "argent"
    case 50 => "or"
    case 60 => "diamond"
    case _ => "inconnu"
}
verif: (age: Int)String

scala> verif(10)
res7: String = inconnu
```

33

Structures de contrôle : fonctions

- Fonctions comme *littéraux* ou *variables*

```
scala> (x: Int) => x+1
res8: Int => Int = $$Lambda$1296/1094867051@4966454a

scala> var inc = (x: Int) => x+1
inc: Int => Int = $$Lambda$1285/2112826959@7da4486

scala> inc(10)
res9: Int = 11

scala> inc = (x: Int) => x + 100
inc: Int => Int = $$Lambda$1294/1404292507@4a218cc6

scala> inc(10)
res10: Int = 110
```

34

Types complexes

- le type tableau (Array)

- séquence d'éléments (souvent du même type)
- construction en utilisant le nom de la classe
- accès indexé pour lecture ou écriture, indice 1^{er} élément = 0

```
scala> val weekend = Array("sam", "dim")
weekend: Array[String] = Array(sam, dim)

scala> weekend.update(0, "ven")

scala> weekend.update(1, "sam")

scala> weekend
res3: Array[String] = Array(ven, sam)
```

0	1
"sam"	"dim"

35

Types complexes

- le type tableau (Array)

```
scala> weekend
res3: Array[String] = Array(ven, sam)

scala> weekend(0)
res6: String = ven

scala> weekend(1)
res7: String = sam

scala> weekend(0) = "jeu"
```

"ven"	"sam"
-------	-------

"jeu"	"sam"
-------	-------

36

Types complexes

- le type liste (List)
 - collection d'éléments (souvent du même type)
 - construction suivant différentes manières :
 - conversion d'un tableau
 - instantiation d'un objet List avec valeurs fournies
 - de manière récursive avec l'opérateur *cons* noté *elem:::liste*

```
scala> val lweekend = weekend.toList  
lweekend: List[String] = List(ven, sam)
```

```
scala> val fruits = List("pomme", "orange", "poire")  
fruits: List[String] = List(pomme, orange, poire)
```

```
scala> val unAtrois = 1 :: 2 :: 3 :: Nil  
unAtrois: List[Int] = List(1, 2, 3)
```

ordre
d'invocation?

37

Types complexes

- Manipulation de listes
 - ajout en tête seulement (immuabilité)

```
scala> 4 :: unAtrois  
res13: List[Int] = List(4, 1, 2, 3)
```

```
scala> val quatreAun = 4 :: unAtrois.reverse  
quatreAun: List[Int] = List(4, 3, 2, 1)
```

- concaténation à l'aide de :: - l'ordre interne est préservé

```
scala> val quatreAcinq = 4 :: 5 :: Nil  
quatreAcinq: List[Int] = List(4, 5)
```

```
scala> val unAcinq = unAtrois ::: quatreAcinq  
unAcinq: List[Int] = List(1, 2, 3, 4, 5)
```

38

Types complexes

- Listes et pattern matching
 - récursion sur une liste : cas de base et cas induction

cas de
base :
List()
vide

```
scala> def affiche (fruits : List[String]) : String = {  
    fruits match {  
        case List() => ""  
        case elem :: suite => elem + ", " + affiche (suite)  
    }  
}  
affiche: (fruits: List[String])String  
  
scala> affiche(fruits)  
res16: String = pomme, orange, poire, .  
  
//A Faire : comment éliminer le dernier symbole ',' ?
```

cas induction :
elem et suite sont
des variables

39

Types complexes

- dés-imbrication de listes : la méthode *flatten*

```
scala> val nestd_unAcinq = List(unAtrois, quatreAcinq)  
nestd_unAcinq: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> nestd_unAcinq.flatten  
res19: List[Int] = List(1, 2, 3, 4, 5)
```

Pourquoi
le type
Any ?

```
unAhuit: List[List[Any]] = List(List(List(1, 2, 3), List(4, 5)), List(6, 7))
```

```
scala> unAhuit.flatten  
res24: List[Any] = List(List(1, 2, 3), List(4, 5), 6, 7)
```

40

Types complexes

- Tuples
 - Collection d'attributs relatifs à un object (cf. modèle rel.)
 - Accès indexé avec `_index` où `index` commence par 1
 - structure immuable, construits souvent à partir de sources externes (ex. fichier csv)

```
scala> val tuple = (12, "text", List(1,2,3))
tuple: (Int, String, List[Int]) = (12, text, List(1, 2, 3))

scala> tuple._1 = 13
<console>:12: error: reassignment to val
      tuple._1 = 13
           ^
```

41

Types complexes

- Tuples et pattern matching
 - possibilité de reconnaître la forme des tuples et d'enclencher un traitement spécifique en utilisant des variables

```
scala> val listeTemp = List((7,2010,4,27,75), (12,2009,1,31,78))
listeTemp: List[(Int, Int, Int, Int, Int)] = List((7, 2010, 4, 27, 75),
(12, 2009, 1, 31, 78))

scala>
listeTemp.map{case(sid,year,month,value,zip)=>(year,value)}
res0: List[(Int, Int)] = List((2010,27), (2009,31))
```

42

Types complexes

- Tableaux associatifs (map)
 - ensemble de paires (clé, valeur) - unicité de clé – clé et valeur de type quelconque mais fixés une fois pour toute
 - possibilité d'insertion et de mise à jour de nouvelles paires

```
scala> var capital = Map("US" -> "Washington", "France" -> "Paris")
capital...

scala> capital("US")
res2: String = Washington

scala> capital += ("US" -> "DC", "Japan" -> "Tokyo")
Map(US -> DC, France -> Paris, Japan -> Tokyo)
```

43

Types complexes

- Classes
 - Conteneurs pour objets ayant les mêmes attributs
 - `class MaClasse (nom: String, num: Int)`
`{ //attributs et méthodes – partie optionnelle }`

```
scala> class Mesure(sid:Int, year:Int, value:Float)
defined class Mesure

scala> listeTemp.map{case(sid,year,month,value,zip)=>new
Mesure(sid,year,value)}
res2: List[Mesure] = List(Mesure@364c93e6, Mesure@66589252)
```

44

Fonctions d'ordre supérieur

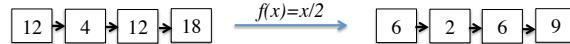
- But : manipulation des éléments d'une liste
 - Correspondance : *map*, *flatMap* et *foreach*
 - Filtres : *filter*, *partition*, *find*
 - Prédicats : *exists*, *forall*
 - Réduction : *fold*, *reduce*, ...

45

Fonctions d'ordre supérieur

- Transformations : *map*, *flatMap* et *foreach*

Map (f : $T \Rightarrow U$), f unaire : appliquer f à chaque élément de C



la dimension de C est préservée le type en entrée peut changer

```
scala> def divide(n:Int) = n/2
succ: (n: Int)Int
```

```
scala> val l= List(12,4,12,18)
```

```
scala> l.map(x=>divide(x))
res1: List[Int] = List(6, 2, 6, 9)
```

46

Fonctions d'ordre supérieur

- Transformations : *map*, *flatMap* et *foreach*

– $l.flatMap(f)$: équivalent de $l.map(f).flatten$

```
scala> def succ(n:Int) = n+1
succ: (n: Int)Int
```

```
scala> nestd_unAinq
```

```
res38: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> val deuxAsix = nestd_unAinq.flatMap(succ)
```

```
deuxAsix: List[Int] = List(2, 3, 4, 5, 6)
```

– $l.foreach(f)$: applique f à chaque élément sans retourner de valeur

```
scala> quatreAinq.foreach(println)
```

```
4
```

```
5
```

47

Fonctions d'ordre supérieur

- Filtres : *filter*, *partition*, *find*

– $l.filter(cond)$, cond retourne un booléen : sélection

– $l.partition(cond) \Leftrightarrow (l.filter(cond), l.filter(!cond))$

– $l.find(cond)$: retourne première occurrence vérifiant $cond$

```
scala> deuxAsix.filter(x=>x%2 ==0)
res46: List[Int] = List(2, 4, 6)
```

```
scala> deuxAsix.partition(x=>x%2 ==0)
res48: (List[Int], List[Int]) = (List(2, 4, 6),List(3, 5))
```

```
scala> deuxAsix.find(x=>x%2 ==0)
res50: Option[Int] = Some(2)
```

```
scala> deuxAsix.find(x=>x%7 ==0)
res51: Option[Int] = None
```

48

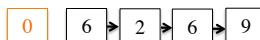
Fonctions d'ordre supérieur

- Réduction : *fold*

– $l.fold(zero)(g: (T,T) \Rightarrow T)$: applique op sur toute paire (x, y) de l , la première paire étant ($zero, l(0)$)

```
scala> def g(a:Int, b:Int) =  
{ println(a+"\t"+b)  
  a+b }
```

```
scala> val l=List(6,2,6,9)  
  
scala> l.fold(0)((a,b)=>g(a,b))  
0  6  
6  2  
8  6  
14 9  
res15: Int = 23
```



49

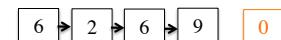
Fonctions d'ordre supérieur

- Réduction : *foldRight*

– $l.foldRight(zero)(g: (T,T) \Rightarrow T)$: applique g en commençant par la droite

```
scala> def g(a:Int, b:Int) =  
{ println(a+"\t"+b)  
  a+b }
```

```
scala> val l=List(6,2,6,9)  
  
scala> l.foldRight(0)((a,b)=>g(a,b))  
9  0  
6  9  
2  15  
6  17  
res16: Int = 23
```



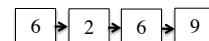
50

Fonctions d'ordre supérieur

- Réduction : *reduce*

– $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

```
scala> def g(a:Int, b:Int) =  
{ println(a+"\t"+b)  
  a+b }  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>g(a,b))  
6  2  
8  6  
14 9  
res17: Int = 23
```



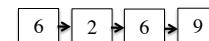
51

Fonctions d'ordre supérieur

- Réduction : *reduce* avec g non associatif!

– $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

```
scala> def moyAB(a:Int, b:Int) = {  
    |   println(a+"\t"+b)  
    |   (a+b)/2  
  }  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>moyAB(a,b))  
6  2  
4  6  
5  9  
res19: Int = 7  
  
scala> l.sum/l.length  
res21: Int = 5
```



Solution?

52

Fonctions d'ordre supérieur

- Réduction : *aggregate*

- *l.reduce(g: (T,T)⇒T)* : même type en entrée et sortie
- *l.aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)* :
 - possibilité d'avoir type en entrée et sortie différents
 - modification du type en entrée T vers U
 - application d'un opérateur sur U

53

Fonctions d'ordre supérieur

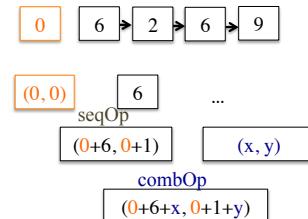
- Réduction : *aggregate*

- *l.aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)*

```
scala> val moy = l.aggregate((0,0))
      ( (acc,value) => (acc._1+value, acc._2+1),
        (lpair,rpair) =>
          (lpair._1+rpair._1,lpair._2+rpair._2))
```

moy: (Int, Int) = (23,4)

élément neutre pour Op
transformation de T vers U
combinaison des paires U



...

54

TME PRISE EN MAIN SCALA

55