Master d'Informatique spécialité DAC

BDLE (Bases de Données Large Echelle)
-Seconde Partie-

Cours 1 : Introduction de Map Reduce et Présentation du système Spark

Mohamed-Amine Baazizi – email: prénom.nom@lip6.fr
http://dac.lip6.fr/master/ues-2014-2015/bdle-2014-2015/

Organisation de la seconde partie du cours BDLE

- Objectifs :
  - Introduction du paradigme Map Reduce (MR)
  - Présentation des différents algorithmes en MR
  - Présentation des langages de haut niveau sur MR
  - Pratique sur un système MR (Spark)
- Organisation :
  - 4 séances, chaque séance = 2h cours + 2h TD/TME
  - Cours interactifs
  - TD/TME : illustration concepts cours + manipulation de différents langages (Scala, Hive, Pig)

2

Plan de la seconde partie BDLE

- Séance 1 (aujourd'hui)
  - Cours : introduction rapide big data – présentation MR, langage Scala et système Spark
  - TME : prise en main de Spark, algorithmes MR
- Séance 2 (ven 17-10)
  - Cours : traduction des requêtes SQL en MR
  - TME : algorithmes MR, requêtes SQL en MR

3

Plan de la seconde partie BDLE (suite)

- Séance 3 (ven 24-10)
  - Cours : présentation de Spark (et Hadoop)
  - TME : fonctionnalités Spark
- Séance 4 (ven 31-10)
  - Cours : présentation Pig et Hive
  - TME : utilisation de Shark

Note : plan peut être adapté si besoin (surtout les TME)

4

Bibliographie conseillée

- Livres

  [Raja] *Mining Massive Datasets*, A.Rajaraman, J. Leskovec and J.Ullman. 2011

  [Lin] *Data-Intensive Text Processing with MapReduce*, J.Lin and C.Dyer. 2010

  [Hadp] *Hadoop, the definitive guide*, T.White. 2012
- Documentation en ligne

  [Spark] http://spark.apache.org/

  … liste non exhaustive, à compléter au fil de l'eau

5

Introduction

- Phénomène Big data*
  - Problématique pas si récente
    - Données scientifiques
  - Emergence d'applications liées au web
    - Croissance du nombre d'utilisateurs (marchands en ligne, réseaux sociaux)
  - Besoin croissant d'analyse
    - Analyse climatographic (risques naturels)
    - Généralisation des capteurs

(*) le terme masses de données est souvent utilisé pour désigner Big data

6

Faire face au phénomène Big data

- Nouvelles solutions matérielles
  - Super-calculateurs, calcul sur GPU
  - Grappe de machine, cloud
- Solutions logicielles
  - Optimisation pour nouvelles architectures (HPC)
  - Parallélisation des algorithmes

7

Nouvelles architectures matérielles

- Grappes de calculs (cluster)
  - Baisse coût des machines
  - Virtualisation, debit réseaux
  - Essor applis. Web
- Gestion distribuée des fichiers
  (+) Capacité importante de stockage
  (-) Réplication et gestion de la cohérence

Architecture en grappe (cluster)

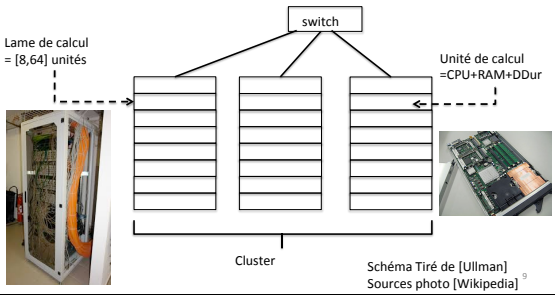


Schéma Tiré de [Ullman]
Sources photo [Wikipedia]

9

Nouvelles architectures matérielles

- Cluster = ensemble de racks (lames) connectés via un switch
- Rack = 8-64 unités (blade) connectées via gigabit Ethernet
- Unité de calcul = CPUs +RAM+DDur

10

Tirer profit des nouvelles architectures matérielles

- Niveau calcul :
  - Diviser en plusieurs petits calculs
  - → synchronisation, gestion des pannes
- Niveau données:
  - Répliquer les données sur plusieurs unités
  - → gestion de la cohérence et des pannes

11

Comment tirer profit des nouvelles architectures matérielles?

- Map Reduce = paradigme + éco-système
- Paradigme
  - Spécification de tâches de calculs
  - Deux primitives : Map et Reduce
- Eco-système
  - Gestion de la tolérance aux pannes
  - Gestion de la réplication

12

Paradigme Map-Reduce

- Inspiré du fonctionnel
- Rappels fonctions d'ordre supérieur
  - *Map*
    - Entrée = une fonction f, une liste L=[e_1,e_2,…,e_n]
    - Résultat = Map (f, L)=[f(e_1), f(e_2) ,…, f(e_n)]

    Exemple f(x)=x/2 L=[12,4,12,3] Map (f, L)=[6,2,6,1.5]
  - *Reduce* (appelé parfois *Fold* ou *Aggregate*)
    - Entrée = un opérateur binaire ϱ, une liste L=[e_1,e_2,…,e_n]
    - Résultat = Reduce (ϱ, L)=ϱ(e_1, ϱ(e_2 ,…, ϱ(e_{n-1},e_n)…)

    Exemple ϱ = '+' Reduce ('+', L)=+(12, +(4, +(12, 3)))=31

13

Paradigme Map-Reduce

Map-Reduce = Généralisation prog. fonctionnelle

Map (f, L)=[f(e_1), f(e_2) ,…, f(e_n)]

Reduce (ϱ, L)=ϱ(e_1, ϱ(e_2 ,…, ϱ(e_{n-1},e_n)…)

Désormais, f génère une liste de paires (clé, val)

f : D_{val} → (D_{cle}, D_{val}) où

- D_{val} valeurs de type quelconque (tuples, strings,…)
- D_{cle} clés (souvent numérique ou strings)

Exemple

f : N→(N,R^+) avec f(x)=(x,x^2-1)

pour L=[9,4,1] on obtient Map (f,L)=[(9,80),(4,15),(1,0)]

14

Paradigme Map-Reduce

Map-Reduce = Généralisation prog. fonctionnelle

Map(f, L)=[f(e_1), f(e_2) ,…, f(e_n)]

Reduce (ϱ, L)=ϱ(e_1, ϱ(e_2 ,…, ϱ(e_{n-1},e_n)…)

Désormais, entrée = liste (clé,[liste-val])

où clé est unique!

Sortie = liste (clé, ϱ([liste-val]))

Exemple

L=[(3,[0.5, 0.3, 0.2]) , (4,[2.0])]

Reduce ('+', L)=[(3,1), (4,2.0)]

15

Workflow Map-Reduce

- Map :
  - Entrée : $[e_1,..,e_n]$
  - Résultat : $[(k_1,v_1),..,(k_n,w_n)]$
- Reduce (une paire à la fois) :
  - Entrée : $(k_1, [v_{1,..,} v_n])$
  - Résultat = $(k, \varrho([v_1,..,v_n]))$

16

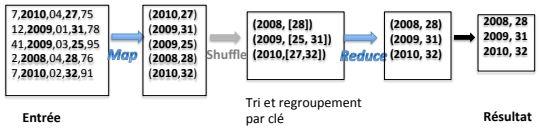
Workflow Map-Reduce

- Map :
  - Entrée : $[e_1,..,e_n]$
  - Résultat : $[(k_1,v_1),..,(k_n,w_n)]$
- Regroupement des valeurs par clé
  - Résultat : $(k_1, [v_{1,..,} v_n]), (k_2, [w_{1,..,} w_n]),\ldots$
- Reduce (une paire à la fois) :
  - Entrée : $(k_1, [v_{1,..,} v_n])$
  - Résultat = $(k, \varrho([v_1,..,v_n]))$

17

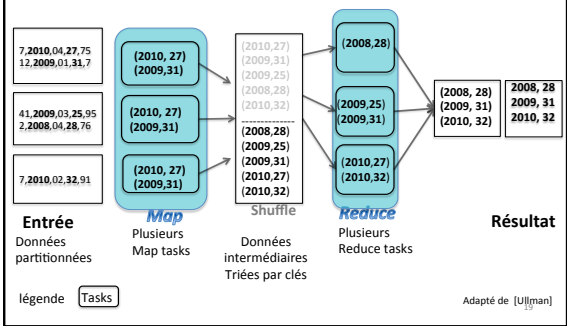
Exemple Map-Reduce

- Entrée : n-uplets (station, annee, mois, temp, dept)
- Résultat : annee, Max(temp)



18

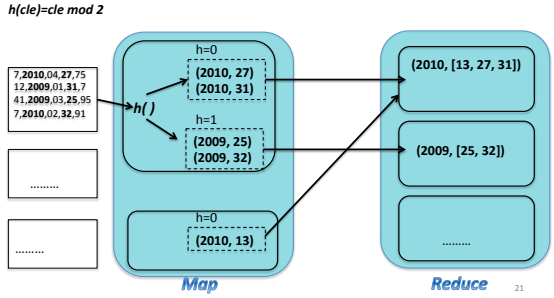
Workflow Map-Reduce



Exécution Map-Reduce

- Map
  - Génère les paires (cle,val) à partir de sa parition
  - Applique une fonction de hachage h sur cle
  - Stocke (cle,val) sur le bucket local h(cle)
- Nœud superviseur (*master*)
  - Fusionne les bucket h(cle) de chaque map
- Reduce
  - Applique ρ

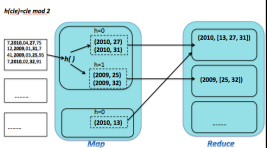
Exécution Map-Reduce



Exécution Map-Reduce : performances

1. Nombre de reduce task
   - 1 reduce task = 1 bucket
   - 1bucket = 1 fichier

   ➔ Limiter le nombre de reduce task

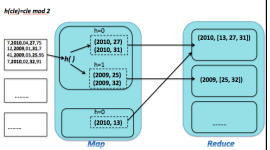


22

Exécution Map-Reduce : performances

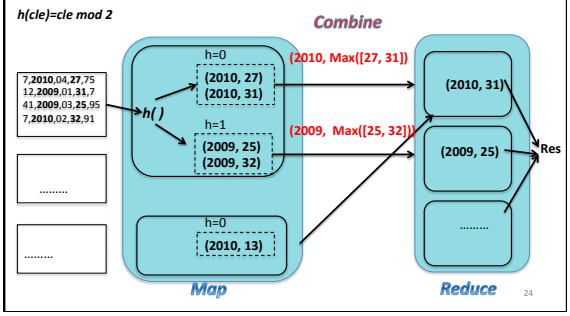
2. Combine
   - Les map et reduce s'exécutent sur des unités différentes

   ➔ transfert des résultats intermédiaires



23

Exécution Map-Combine-Reduce



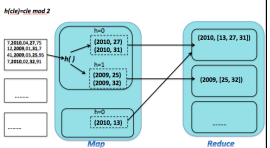
24

Exécution Map-Reduce : performances

2. Combine
   - Les map et reduce s'exécutent sur des unités différentes

   ➔ transfert des résultats intermédiaires

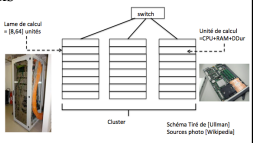
   ➔ **ρ doit être commutatif et associatif**



25

Exécution Map-Reduce

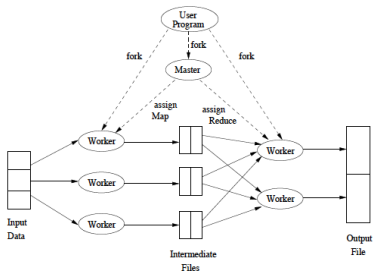
- Prog MR = 1 master + *n* workers
- Master :
  - Créer les map et les reduce tasks
  - Les affecter à des workers
  - Superviser l'exécution
    - Progression, stockage des résultats intermédiaires, relance des workers ayant échoué
- Worker
  - En charge d'un map ou d'un reduce **jamais les deux en même temps**



Rappel Archi. Cluster

26

Exécution Map-Reduce



Tiré de [Ullman]

27

Algorithmes en Map-Reduce

- Que peut-on exprimer ? Comment se traduisent les problèmes classiques ?
  - Pas couvert dans cette partie du cours
- Comment traduire les requêtes relationnelles?
  - Fortement inspirée de l'implantation des opérateur dans les SGBG classiques (jointure par hachage)

28

Opérateurs de l'algèbre relationnelle

- Opérateurs ensemblistes :
  - Union : $\cup$
  - Intersection : $\cap$
  - Différence : –
- Autres opérateurs :
  - Projection : π_X
  - Sélection : σ_C
  - Jointure naturelle : $\bowtie$
  - Renommage : $\varrho_{A->B}$
  - Produit cartésien : $\times$
  - Division : $\div$
- Schéma et instances des relations :
  Schéma : $R(att_1, \ldots, att_n)$ *Instance* : $R=\{r_0,r_1,\ldots\}$

29

Traduction Opérateurs unaires

1. Sélection $\sigma_C(R)$

Map Pour chaque r_i qui satisfait C constituer la paire (r_i,r_i)
Reduce Identité

2. Projection $\pi_X(R)$

Map Pour chaque r_i éliminer les attributs $\notin X$
Pour chaque p_i ainsi obtenu constituer la paire (p_i,p_i)
Reduce Transforme les paires $(p_i,[p_i, p_i,\ldots])$ en p_i
Astuce : envoyer les tuples identiques au même reducer
Analogue au partitionnement implanté dans les SGBD classiques!

30

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

1. Union R ∪ S

Map Pour chaque p_i constituer la paire (p_i, p_i)

Reduce Produit p_i à partir de $(p_i, [p_i,\ldots])$

Observation : Entrée reduce $= (p_i, [p_i])$ ou $(p_i, [p_i, p_i])$

2. Intersection R ∩ S

Map Idem

Reduce

31

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

1. Union R ∪ S

Map Pour chaque p_i constituer la paire (p_i, p_i)

Reduce Produit p_i à partir de $(p_i, [p_i,\ldots])$

Observation : Entrée reduce $= (p_i, [p_i])$ ou $(p_i, [p_i, p_i])$

2. Intersection R ∩ S

Map Idem

Reduce Produit p_i que si ou $(p_i, [p_i, p_i])$

32

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

1. Union R ∪ S

Map Pour chaque p_i constituer la paire (p_i, p_i)

Reduce Produit p_i à partir de $(p_i, [p_i,\ldots])$

Observation : Entrée reduce $= (p_i, [p_i])$ ou $(p_i, [p_i, p_i])$

2. Intersection R ∩ S

Map Idem

Reduce Produit p_i que si ou $(p_i, [p_i, p_i])$

Exemple : R={(1, 2), (3, 4), (1, 3)} et S={(1,2), (3, 3)}

33

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

1. Union R ∪ S

Map Pour chaque p_i constituer la paire (p_i, p_i)
Reduce Produit p_i à partir de (p_i, [p_i,…])
Observation : Entrée reduce = (p_i, [p_i]) ou (p_i, [p_i, p_i])

2. Intersection R ∩ S

Map Idem
Reduce Produit p_i que si ou (p_i, [p_i, p_i])
Exemple : R={(1, 2), (3, 4), (1, 3)} et S={(1,2), (3, 3)}
Réflexion : Généralisation?

34

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

3. Différence R-S

Map

Reduce

Exemple : R={(1, 2), (3, 4), (1, 3)} et S={(1,2), (3, 3)}

35

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

3. Différence R-S

Map Produit paires (r_i,R) et (s_i,S) à partir de R et de S resp.
Résultat interm. : (p_i, [R]), (p_i, [R, S]), (p_i, [S, R]) ou (p_i, [S])
Reduce Produit p_i à partir des (p_i, [R])

Exemple : R={(1, 2), (3, 4), (1, 3)} et S={(1,2), (3, 3)}

36

Traduction Opérateurs ensemblistes

Hypothèse : sch(R)=sch(S)

3. Différence R-S

Map Produit paires (r_i,R) et (s_i,S) à partir de R et de S resp.
Résultat interm. : (p_i, [R]), (p_i, [R, S]), (p_i, [S, R]) ou (p_i, [S])
Reduce Produit p_i à partir des (p_i, [R])

Exemple : R={(1, 2), (3, 4), (1, 3)} et S={(1,2), (3, 3)}

Réflexion : Implantation de la différence symétrique RΔS?
Rappel : RΔS= (R-S) ∪ (S-R) = (R∪S)-(R∩S)

37

Traduction de la jointure naturelle

Hypothèse : R(A, B) et S(B, C)
Intuition : partitionnement (tri ou hachage) utilisé par les SGBD
Map pour chaque (a, b) de R produire (b, (R, a))
pour chaque (b, c) de S produire (b, (S, c))
Formes résultat interm: p1 = (b, [(R,a_i)]) p2 = (b, [(S,c_j)])
p3= (b, [(R, a_1), …,(R, a_k), (S, c_1), ... (S, c_m)]) ← **tuples joignables**
Reduce Considérer les paires p3
Produire (b, {a_1,..,a_k} × {b}×{c_1,..,c_m})
Exemple : R={(2, 1), (2, 3), (3, 1), (4, 1)} et S={(1, 4), (1, 1), (2, 3)}
Réflexion : Généralisation à plusieurs attributs et à d'autres jointures

38

Traduction de la jointure naturelle

Hypothèse : R(**A**, **B**) et S(**B**, **C**) avec **A**, **B**, et **C** ensbles d'attributs
Map pour chaque ($\overbrace{a_0..a_n}^{\mathcal{A}}$,$\overbrace{b_0..b_m}^{\mathcal{B}}$) de R produire ($b_0..b_m$, (R, $a_0..a_n$))
pour chaque ($b_0..b_m$,$\overbrace{c_0..c_p}^{C}$) de S produire ($b_0..b_m$, (S, $c_0..c_p$))
Formes résultat interm: p1 = ($\mathcal{B}$, [(R,$\mathcal{A}$)]) p2 = ($\mathcal{B}$, [(S,C)])
p3= ($\mathcal{B}$, [(R, $\mathcal{A}_1$), …,(R, $\mathcal{A}_k$), (S, C_1), ... (S, C_p)])
Reduce Considérer les paires p3
Produire ($\mathcal{B}$, {$\mathcal{A}_1$,..,$\mathcal{A}_k$} × {$\mathcal{B}$}×{C_1,..,C_p})

39

Opérateurs de l'algèbre relationnelle

- Opérateurs ensemblistes :
  - ~~Union : $\cup$~~
  - ~~Intersection : $\cap$~~
  - ~~Différence : $-$~~
- Autres opérateurs :
  - ~~Projection : π_X~~
  - ~~Sélection : σ_C~~
  - ~~Jointure naturelle : $\bowtie$~~
  - Renommage : $\varrho_{A->B}$
  - Produit cartésien : $\times$
  - Division : $\div$
- Schéma et instances des relations :
  Schéma : $R(att_1, \ldots, att_n)$ *Instance* : $R=\{r_0,r_1,\ldots\}$

40

Traduction des fonctions d'agrégats

- Trivial pour Somme( ), Max( ) et Min( )
  - Max et Min et Addition commutatifs et associatifs
  - En fonctionnel : *list-homomorphism*
- Définition: (List-homomorphism)
  En se dotant d'un opérateur de concaténation de listes •
  Une fonction *h* est un *list-homomorphism* s'il existe un opérateur associatif ρ doté de l'élément neutre *e* tel que
  - $h([\,])= e$ où [] est la liste vide
  - $h(L_1 \bullet L_2)=h(L_1)\ \rho\ h(L_2)$

41

List-homomorphism : illustration

Question : Est-ce que Min( ) est LH?

42

List-homomorphism : illustration

Question : Est-ce que Min( ) est LH?

Réponse :

Considèrer que ρ retourne le plus grand entier parmi ses deux arguments a et b, i.e

a ρ b = b si a>b et a ρ b = a si a<=b

et $+\infty$ comme élément neutre.

$Min(L_1 \bullet L_2)=Min(L_1)\ \rho\ Min(L_2)$

43

L'homomorphisme pour caractériser le parallélisme

Lemme de l'homomorphisme

Une fonction *h* est un homomorphisme pour l'opérateur de concaténation • ssi

h=Reduce(ρ, Map(f, ..))

où *f* est une fonction et ρ un opérateur

- Comment l'utiliser?

F() admet une solution parallèle si on arrive à la formuler à partir de Reduce(ρ, Map(f, ..))

44

Illustration avec Min

Remarquer que

Min(L)=Reduce(ρ, Map(f, L))

Avec ρ retourne le plus grand entier (slide précédent) et *f* la fonction identité, i.e f(a b c …z)= (a b c …z)

Donc Min() admet un solution en parallèle!

Que se passe-t-il pour la Somme et la moyenne?

- Pour la Somme, il suffit de considérer que ρ est +
- Et pour la moyenne?

45

Solution parallèle pour la Moyenne

- **Tentative** :
  - *f* est l'identitié et a ρ b=(a+b)/2
  - →Moy() n'est pas un homomorphisme pour la concaténation d'entiers :

  Reduce(ρ,(a,b,c)) produit ((a+b)/2)+c)/2≠(a+b+c)/3
- **Solution** :
  - *f* doit retourner une paire (a,1)
  - (a,i) ρ (b,j) = (a+b, i+j)
  - Moy = FCT**(**Reduce(ρ, Map(*f*, ..))**)** où FCT((a,b))=a/b

46

Présentaion de Spark

Motivation de Spark

- Supporter des traitements itératifs efficacement
  - Applications émergentes tels que PageRank, clustering par nature itératives
  - Systèmes du style Hadoop matérialisent les résultats intermédiaires → performances dégradées
- Solution
  - Les données doivent résider en mémoire-centrale et être partagées → mémoire distribuée

48

Principe utilisé dans Spark

- *Resilient Distributed Datasets (RDDs)*
  - Structures accessibles en lecture seule
  - Stockage distribué en mémoire centrale
  - Restriction aux opérations sur gros granules
    - Transformations de la structure en entier vs MAJ valeurs atomiques qui nécessite propagation replicats
  - Journalisation pour assurer la tolérance aux fautes
    - Possibilité de rejouer les transformations vs checkpointing

49

Fonctionnement des RDD

1. Création
   - Chargement données depuis SGF distribué/local
   - Transformation d'une RDD existante

   Note : RDD est une séquence d'enregistrements
2. Transformations
   - *map* : applique une fonction à chaque élément
   - *filter* : restreint aux éléments selon condition
   - *join* : combine deux RDD sur la base des clés *

(*) Les RDD en entrée doivent être des séquences de paires (clé,valeur)

50

Fonctionnement des RDD

3. Actions
   - *collect* : retourne les éléments
   - *count* : comptes les éléments
   - *save* : écrit les données sur le SF
4. Paramétrage du stockage en mémoire
   - *persist* : force le maintien en mémoire
   - *unpersist* : force l'écriture sur disque
- Notes :
  - par défaut, les RDD sont persistantes en mémoire
  - Si manque d'espace alors écriture sur disque
  - Possibilité d'attribuer des priorités

51

Illustration d'une RDD

On considère une chaîne de traitements classique

1. Chargement depuis stockage (local ou hdfs)
2. Application d'un filtre simple
3. Cardinalité du résultat de 2
4. Paramétrage de la persistance

```
1 lines=spark.textFile("hfds://file.txt")
2 data=lines.filter(_.contains( "word"))
3 data.count
4 data.persist()
```

52

Illustration d'une RDD

On considère une chaîne de traitements classique

1. Chargement depuis stockage (local ou hdfs)
2. Application d'un filtre simple
3. Cardinalité du résultat de 2
4. Paramétrage de la persistance

```
1 lines=spark.textFile("hfds://file.txt")
2 data=lines.filter(_.contains( "word"))
3 data.count
4 data.persist()
```

Lazy evaluation

Construire les RDDs seulement si action (mode pipelined)

Exemple : lines n'est construit qu'à la ligne 3

➔ Chargement *sélectif* de file.txt

53

API Spark

- Documentation https://spark.apache.org/docs/latest/
- Plusieurs langages hôtes
  - Java
  - Scala (langage fonctionnel sur JVM)
  - Python
- Choix pour ce cours = Scala (Scalable Language)
  - Documentation http://www.scala-lang.org/api/current/#package
  - Tutoriel http://docs.scala-lang.org/tutorials/

54

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 '+ ' sur des objets de la classe Int
  - Fonctionnel :
    1. Les fonctions se comportent comme des valeurs : peuvent être retournées ou passées comme arguments
    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

55

Scala : rapide tour d'horizon

- Immuabilité des données

```
//1- déclarer une variable et lui associer une valeur
scala> var a=3
a: Int = 3
//2- vérifier la référence attribuée par Scala
scala> a
res0: Int = 3
//3- associer à a une nouvelle valeur, 5
scala> a=5
a: Int = 5
//même chose que 2
scala> a
res1: Int = 5
//manipuler l' "ancienne" valeur de a via res0
scala> res0*2
res2: Int = 6
```

56

Scala : rapide tour d'horizon

- Variables vs valeurs

```
//1- déclarons une valeur n
scala> val n=1+10
n: Int = 11
//2-essayons de la modifier
scala> n=n+1
<console>:12: error: reassignment to val
       n=n+1
         ^
//3- déclarons une variable
scala> var m=10
m: Int = 10
//4- idem que 2
scala> m=m+1
m: Int = 11
```

val n'autorise pas de lier une variable plus d'une fois

57

Scala : rapide tour d'horizon

- Inférence de types

```
scala> var a=1
a: Int = 1
scala> var a="abc"
a: String = abc
scala> var a=Set(1,2,3)
a: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
scala> a+=4
res1: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)
scala> a+="a"
<console>:9: error: type mismatch;
 found   : String
 required: scala.collection.immutable.Set[Int]
              a+="a"
               ^
```

58

Scala : rapide tour d'horizon

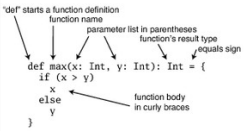
- Fonctions

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

scala> max2(1,3)
res3: Int = 3

scala> max2(res3,0)
res5: Int = 3

scala> max2(max2(1,2),3)
res6: Int = 3
```



59

Scala : rapide tour d'horizon

- Itérations avec *for-each*
  - style de programmation impérative
  - méthode associé à un tableau (ou liste, ou ensemble)
  - prend en entrée une fonction, souvent *print*

```
//déclarer une liste et l'initialiser
scala> var l=List(1,2,3)
l: List[Int] = List(1, 2, 3)
//imprimer chaque élément de la liste
scala> l.foreach(x=>print(x))
123
//syntaxe équivalente
scala> l.foreach(print)
123
```

60

Scala : rapide tour d'horizon

- Tableaux
  - Collections d'objets typés
  - Initialisation directe ou avec *apply*()
  - Mise à jour directe ou avec *update*()

```
scala> val b=Array.apply("1","2","3") //Initialisation avec apply
b: Array[String] = Array(1, 2, 3)
scala> b(0)="33" //mise à jour directe
scala> b.update(1,"22 ") //mise à jour avec en utilisant update
```

61

Scala : rapide tour d'horizon

- Listes et ensembles
  - Collections d'objets typés **immuables**
  - Initialisation directe
  - Mise à jour impossible

```
scala> val da=List(1,2,3) //initialisation directe
da: List[Int] = List(1, 2, 3)
scala> da(2) //accès indexé
res53: Int = 3
scala> da(0)=1 //tentative de mise à jour
<console>:9: error: value update is not a member of List[Int]
        da(0)=1
        ^
```

62

Scala : rapide tour d'horizon

- Opérations sur les listes
  - Concaténation avec :::, ajout en tête avec ::
  - Inverser l'ordre d'une liste *reverse*()
  - Et plein d'autres méthodes (cf Annexe A)

```
scala> val l1=List(1,2,3)
l1: List[Int] = List(1, 2, 3)
scala> val l2=List(4,5)
l2: List[Int] = List(4, 5)
scala> l1:::l2
res44: List[Int] = List(1, 2, 3, 4, 5)
```

```
scala> (6::l2)
res47: List[Int] = List(6, 4, 5)
scala> val l1bis=1::2::3::Nil
l1bis: List[Int] = List(1, 2, 3)
//deviner la sorti de cette instruction
scala> l1:::(6::l2.reverse).reverse
```

63

Scala : rapide tour d'horizon

- Tuples
  - Différents types pour chaque élément (12, '22', <1,2,3>)
  - Accès indexé avec ._*index* où *index* commence par 1

```
//creation d'un tuple complexe
scala> val co=(12, "text", List(1,2,3))
co: (Int, String, List[Int]) = (12,text,List(1, 2, 3))
scala> co._0
<console>:9: error: value _0 is not a member of (Int,
String, List[Int])
scala> co._1
res56: Int = 12
scala> co._3
res58: List[Int] = List(1, 2, 3)
```

64

Scala : rapide tour d'horizon

- Tableaux imbriqués dans des tuples
  - Rappel : les élément des tableaux peuvent changer
  - Un tableau imbriqué dans un tuple est une référence

```
scala> b //reprendre le b de
res66: Array[String] = Array(33, 22, 3)
scala> b(0)="100"
scala> co
res69: (Int, String, Array[String]) = (12,txt,Array(100, 22, 3))
```

65

Scala : rapide tour d'horizon

- Les tableaux associatifs (*Map*)
  - Associer à chaque entrée un élément
  - Extension avec +

```
scala> var capital = Map("US" -> "Washington", "France" -> "Paris")
capital: scala.collection.immutable.Map[String,String] = Map(US ->
Washington, France -> Paris)
scala> capital("US")
res2: String = Washington
scala>   capital += ("Japan" -> "Tokyo")
```

66

Scala : rapide tour d'horizon

- Fonctions d'ordre supérieur *Map* et *Reduce*
  - Fonctionnement : déjà vu
  - Notation abrégée

```
scala> List(1, 2, 3) map (z=>z+1) //est équivalent à la ligne suivante
scala> List(1, 2, 3).map (_ + 1)
res71: List[Int] = List(2, 3, 4)
//rappel: capital désigne Map(US -> Washington, France -> Paris)
scala> capital.map(z=>(z._1.length))
res77: scala.collection.immutable.Iterable[Int] = List(2, 6)

scala> capital.reduce((a,b) =>  if(a._1.length>b._1.length) a else b)
res7: (String, String) = (France,Paris)
scala> capital+=("Algeria"->"Algiers")
scala> capital.reduce((a,b) =>  if(a._1.length>b._1.length) a else b)
res10: (String, String) = (Algeria,Algiers)
```

67

Scala : rapide tour d'horizon

- Plein d'autres fonctionnalités (consulter références)
- But de ce cours : utiliser Scala sous Spark

68

Scala sous Spark : fonctions

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

Figure tirée de [Spark]

69

Scala sous Spark : illustration MR

Préparation des données

```
scala> val lines=sc.textFile("/user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] …
scala> lines.count
res3: Long = 5
scala> lines.collect
res4: Array[String] = Array(7,2010,04,27,75,
12,2009,01,31,7, ….
```

```
7,2010,04,27,75
12,2009,01,31,78
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
```

/user/cours/mesures.txt

70

Scala sous Spark : illustration

Préparation des données

```
scala> val lines=sc.textFile("/user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] …
scala> lines.count
res3: Long = 5
scala> lines.collect
res4: Array[String] = Array(7,2010,04,27,75,
12,2009,01,31,7, ….
```

```
7,2010,04,27,75
12,2009,01,31,78
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
```

/user/cours/mesures.txt

- *Map* (*f*:T⇒U)

```
scala> lines.map(x=>x.split(",")).collect
res8: Array[Array[String]] = Array(Array(7, 2010, 04, 27, 75), Array(12, 2009, 01, 31,
7), …

scala> lines.map(x=>x.split(",")).map(x=>(x(1),x(3))).collect
res12: Array[(String, String)] = Array((2010,27), (2009,31), …
```

71

Scala sous Spark : illustration

Préparation des données

```
scala> val lines=sc.textFile("/user/cours/mesures.txt")
lines: org.apache.spark.rdd.RDD[String] …
scala> lines.count
res3: Long = 5
scala> lines.collect
res4: Array[String] = Array(7,2010,04,27,75,
12,2009,01,31,7, ….
```

```
7,2010,04,27,75
12,2009,01,31,78
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
```

/user/cours/mesures.txt

- *Map* (*f*:T⇒U)

```
scala> lines.map(x=>x.split(",")).collect
res8: Array[Array[String]] = Array(Array(7, 2010, 04, 27, 75), Array(12, 2009, 01, 31,
7), …

scala> lines.map(x=>x.split(",")).map(x=>(x(1),x(3))).collect
res12: Array[(String, String)] = Array((2010,27), (2009,31), …
```

72

Scala sous Spark : illustration

- *Map* (*f:*T⇒U)

```
scala> lines.map(x=>x.split(",")).collect
res8: Array[Array[String]] = Array(Array(7, 2010, 04, 27, 75), Array(12, 2009, 01, 31, 7), …

scala> lines.map(x=>x.split(",")).map(x=>(x(1),x(3))).collect
res12: Array[(String, String)] = Array((2010,27), (2009,31), …
```

- *ReduceByKey* (*f:*(V,V) ⇒ V)

```
//convertir l'entrée en entier pour pouvoir utiliser ReduceByKey!
scala> val v=lines.map(x=>x.split(",")).map(x=>(x(1).toInt,x(3)))
v: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[7] at map at <console>:14
scala> val max=v.reduceByKey((a,b)=>if (a>b)a else b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,31))
```

73

Scala sous Spark : illustration

- *ReduceByKey* (*f:*(V,V) ⇒ V)

```
//convertir l'entrée en entier pour pouvoir utiliser ReduceByKey!
scala> val v=lines.map(x=>x.split(",")).map(x=>(x(1).toInt,x(3)))
v: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[7] at map at <console>:14
scala> val max=v.reduceByKey((a,b)=>if (a>b)a else b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,31))
```

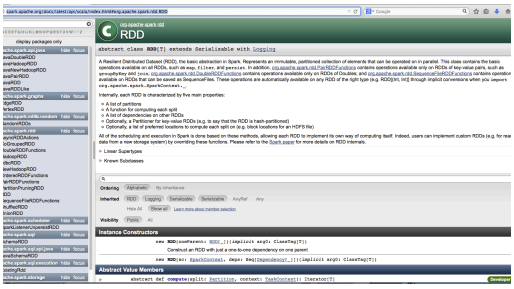
- Comportement du *ReduceByKey*

```
scala> val max=v.reduceByKey((a,b)=>a).take(10)
max: Array[(Int, String)] = Array((2010,27), (2008,28), (2009,31))

scala> val max=v.reduceByKey((a,b)=>b).take(10)
max: Array[(Int, String)] = Array((2010,32), (2008,28), (2009,25))
```

74

Où trouver la doc?



75

Utiliser Spark en TME

Consulter la notice
http://dac.lip6.fr/master/ues-2014-2015/bdle-2014-2015/bdle-notes-de-td-tme/

76

Annexe A : manipulation des listes

What it is	What it does
List() or Nil	The empty List
List("Cool", "tools", "rule")	Creates a new List[String] with the three values "Cool", "tools", and "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Creates a new List[String] with the three values "Will", "fill", and "until"
List("a", "b") ::: List("c", "d")	Concatenates two lists (returns a new List[String] with values "a", "b", "c", and "d")
thrill(2)	Returns the element at index 2 (zero based) of the thrill list (returns "until")
thrill.count(s => s.length == 4)	Counts the number of string elements in thrill that have length 4 (returns 2)
thrill.drop(2)	Returns the thrill list without its first 2 elements (returns List("until"))
thrill.dropRight(2)	Returns the thrill list without its rightmost 2 elements (returns List("Will"))
thrill.exists(s => s == "until")	Determines whether a string element exists in thrill that has the value "until" (returns true)
thrill.filter(s => s.length == 4)	Returns a list of all elements, in order, of the thrill list that have length 4 (returns List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Indicates whether all elements in the thrill list end with the letter "l" (returns true)
thrill.foreach(s => print(s))	Executes the print statement on each of the strings in the thrill list (prints "Willfilluntil")
thrill.foreach(print)	Same as the previous, but more concise (also prints "Willfilluntil")

77

Annexe A : manipulation des listes

thrill.head	Returns the first element in the thrill list (returns "Will")
thrill.init	Returns a list of all but the last element in the thrill list (returns List("Will", "fill"))
thrill.isEmpty	Indicates whether the thrill list is empty (returns false)
thrill.last	Returns the last element in the thrill list (returns "until")
thrill.length	Returns the number of elements in the thrill list (returns 3)
thrill.map(s => s + "y")	Returns a list resulting from adding a "y" to each string element in the thrill list (returns List("Willy", "filly", "untily"))
thrill.mkString(", ")	Makes a string with the elements of the list (returns "Will, fill, until")
thrill.remove(s => s.length == 4)	Returns a list of all elements, in order, of the thrill list *except those* that have length 4 (returns List("until"))
thrill.reverse	Returns a list containing all elements of the thrill list in reverse order (returns List("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)	Returns a list containing all elements of the thrill list in alphabetical order of the first character lowercased (returns List("fill", "until", "Will"))
thrill.tail	Returns the thrill list minus its first element (returns List("fill", "until"))

78