BDLE
Manipulation de données à large échelle avec Spark

1/12/2017

Modèle de données

- Une collection est un ensemble d'éléments de même type
- Type des éléments
  - Un objet quelconque
  - Un N-uplet avec N attributs
    - Un attribut a un nom et un type (nombre, chaine, ...)
    - cf le modèle relationnel

Manipuler une collection

- Opération algébrique
  - Une opération transforme une (ou plusieurs) collections en une collection
  - Opération unaire (1 entrée) : sélection, projection, ...
  - Opération binaire (2 entrées) : jointure, union, différence, ...
- Expression : composition d'opérations
  - Arbre algébrique
  - En général: graphe orienté acyclique d'opérations (DAG)

Optimiser la manipulation

- Expression logique
  - Indépendante du programme qui évalue l'expression
- Expression physique
  - Fixe l'ordre des opérations
  - Fixe l'algorithme pour évaluer chaque opération
- Optimisation d'une expression
  - Expr logique = plusieurs expr physiques equivalentes
  - Expr logique → choisir **une** expr physique
    - Choisir l'ordre des opérations
    - Choisir un algorithme pour évaluer chaque opération

Stocker et organiser les données en **mémoire**

- Une zone mémoire par élément
  - Nombreuses zones, occupe bcp de place en mémoire
- Une zone mémoire pour plusieurs éléments
  - Objets "compressés" dans une page mémoire.
  - Faible surcout pour (dé)compresser les éléments
- Organisation des données de type Nuplets
  - Stockage orienté colonne
  - Découpage vertical : un tableau de valeurs par attribut
    - User(nom, prénom, âge) fragmenté en 3 tableaux
  - Chaque tableau contient les valeurs d'un seul domaine
  - Meilleure compression

Comparaison des structures de collections existant dans Spark

- RDD
  - Type objet
  - Opération algèbrique physique
  - Stockage par objet
- DataFrame (= Dataset de Nuplet)
  - Type Nuplet
  - Opération algébrique logique
  - Stockage compressé et orienté colonne
- Dataset
  - Type objet
  - Opération algébrique logique
  - Stockage compressé

Lire une collection

- RDD
  - val data = sc.textFile(" nom de fichier ").map(ligne => objet)
- DataFrame
  - val data = sc.textFile(" nom de fichier "). ").map(ligne => nuplet).**toDF**(*noms d'attributs*)
- Dataset
  - val data = sc.textFile(" nom de fichier "). map(ligne => objet).**toDS()**

Conversion entre les collections

- Obtenir une RDD à partir d'un DF
  - val r = data.rdd
- Obtenir un Dataset à partir d'un DF
  - Dataset de nuplet
    - Ou val ds = data.as[(Long, String, String)]
  - Dataset d'objet (exple avec la classe Film)
    - val ds = data.as[Film]
- Obtenir un Dataset à partir d'un RDD d'objets T
  - val ds = data.toDS() // donne un dataset d'objets T
- Un DataFrame est un Dataset[Row]

Schéma d'une collection

- Afficher le nom et le type des attributs d'un DataFrame ou d'un Dataset
  - data.**columns**()
- **Renommage** des attributs
  - Val d2 = d1.withColumnRenamed("old", "new")
  - Ou la syntaxe plus proche de SQL :
  - val d2 = d1.select(col("numF") as "numéroFilm", col("numU") as "numéroUser")

Exemple : données sur les avis de films

- Voir TME: fichier tme-dataset-etudiant.scala
- Les utilisateurs

  class Utilisateur (numU: Int, genre: String, age: Int, profession: Int, ville: String)

  val USERS = ...
- Les films

  class Film (numF: Int, titre: String, genre: Array[String])

  val FILMS = …
- Les avis

  class Avis (numU: Int, numF: Int, note: Int, date: Long)

  val AVIS = ...

Sélection, Projection

- Sélection relationnelle
  - Syntaxe SQL: **where** (prédicat SQL)
    - USERS.where(" age=22 AND ville like '75%' ")
- Sélection générale: **filter** (x => condition complexe)
    - FILMS.filter(f => f.titre.upperCase() == "TITANIC")
- Projection relationnelle
  - **select** (attributs avec **renommage** éventuel)
    - FILMS.select(col("numF"), col("titre") as "titreF")

    ou
    - FILMS. select("numF").withColumnRenamed("numF", "numeroFilm")
- Projection générale: **map** (x => nouvel élément)
    - AVIS.map(x => x.note * 2)
- Projection sans doublons : **distinct**()
    - NOTES.select("numU").**distinct**()

Equi-Jointure

- Equi-jointure relationnelle:
  - **join** (collection, attribut de jointure)
    - USERS.join(AVIS, "numU")
    - Le résultat est une collection de nuplet de type Row
      - Le nuplet contient tous les attributs de Users et Avis
- Equi jointure entre objets
  - joinWith(collection, attribut de jointure)
    - USERS.joinWith(AVIS, "numU")
    - Le résultat est une collections de couple d'objets : (User, Avis)
- Types de jointure
  - inner, outer, left_outer, right_outer, leftsemi
- Eviter les ambiguïtés de noms d'attributs
  - Seuls les attributs de jointure ont le même nom
- Jointure générale
  - Préciser le prédicat de jointure (syntaxe SQL)

Produit cartésien

- Syntaxe RDD
  - Val p = C1.**cartesian**(C2)
- Syntaxe Dataset
  - Méthode *join* sans préciser d'attribut
  - Exple: former toutes les paires d'utilisateurs
    - Soit USERS2 identique à USERS mais avec des attributs renommés
    - val paires = USERS.join(USERS2)

Union, Intersection, Différence

- union
- intersect
- RDD: subtract

Tri

- sort
  - USERS.sort("age") dans l'ordre ascendant
  - USERS.sort(col("age").desc) dans l'odre descendant
- orderBy

Désimbriquer une collection d'ensemble

- Collection d'ensemble → Collection d'éléments
  - **flatMap** (e => e)
  - Le résultat est l'union des ensembles *e*.
- Coll. d'élts →Coll. d'ensemble → Coll. d'élts
  - **flatMap** (elt => ensemble)
  - Le résultat est l'union des ensembles obtenus pour chaque *elt*.

Agrégation globale

- Reduce
  - Agréger un ensemble d'objets T en un seul objet T.
  - Est évalué dans un ordre quelconque
    - Contrairement au foldLeft et foldRight
  - Doit être commutatif et associatif

Regroupement puis agrégation

- Regroupement:
  - **groupByKey** (x => clé de regroupement)
    - FILMS.groupByKey(f => f.titre)
  - **groupBy** (attributs de regroupement)
- Agrégation relationnelle:
  - Dénombrer : count() *ne pas confondre cette agrégation avec l'action de même nom*
  - Fonctions sur un attribut: min, max, avg
  - Plusieurs agrégations par groupe: fonction **agg**
    - AVIS.groupBy("numU").agg(max(col("date")), avg(col("note")))
- Agrégation générale (associative et commutative)
  - reduceGroups
- Agrégation par itération sur les éléments d'un groupe
  - mapGroups
  - flatMapGroups

- RDD: regroupement suivi d'une agrégation: reduceByKey

Autres regroupements

- C1.cogroup(C2)
  - Regrouper deux collections C1 t C2 avec la même clé
    - Résultat { (clé, (éléments de C1), (éléments de C2)}
- zipWithIndex
  - Numérotation dense de 1 à n
    - {A, B, A, C} devient { (A, 1), (B, 2), (A, 3), (C, 4) }
- zipWithUniqueIndex
  - Numérotation unique mais **pas** dense
    - {A, B, A, C} devient { (A, 10), (B, 11), (A, 20), (C, 21) }
    - Numéroter plusieurs parties indépendamment
      - Avec des plages de valeurs différentes
    - Très simple à paralléliser

Transformation et Action

- On peut composer des opérations de **transformation**
  - Exple d'expression composée
    - val f = FILMS.filter(...).join(...).select(...)....

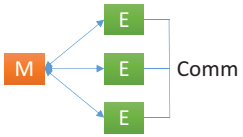
- Une expression peut être évaluée par une **action**
  - Lire le résultat
    - f.take(10) *seulement de début*
    - f.collect() *tout*
  - Dénombrer le résultat
    - f.count()
  - Stocker le résultat
    - f.save(fichier)
  - Aggrégation
    - f.reduce(...)

Architecture et exécution

dans Spark

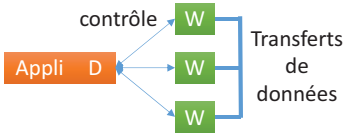
Architecture de la plateforme Spark

- Architecture répartie sur un cluster de machines. Deux types de machines
- 1 machine Master
  - Point d'entrée pour lancer une application
  - Attribution des ressources au démarrage de l'application
- Plusieurs machines Executors
  - ressources ram, cpu, comm



Architecture d'une application utilisant Spark

- Une machine pour le driver
  - Appli + contrôle de l'exécution
- Plusieurs Workers : service de calcul
  - 1 (ou plusieurs) worker par executor.



Gestion répartie des données

- Répartir = distribuer
- Répartir les données = partitionner + placer

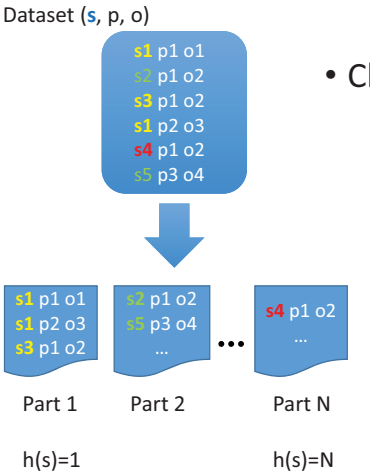
Partitionner les données

- Une collection est partitionnée en plusieurs morceaux appelés partitions.
  - Plusieurs éléments par partition
- Localité
  - Une partition est locale à une machine
  - Les éléments d'une partition sont ensembles sur une même machine
- Parallélisme
  - N partitions = N tâches indépendantes

Clé de répartition

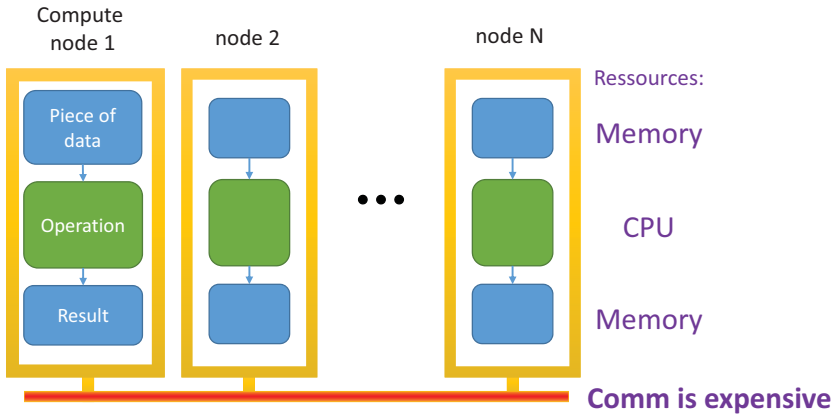
- Une collection C a une clé de répartition k noté C^k
- Par round robin: $C^{\emptyset}$
- Par hachage: C^x
  - h(x) → numéro de partition, avec h() fixée
  - Exple:
    - val A = USERS.repartition("ville"), noté A^{ville}
    - val B = USERS.groupBy("ville").count(), noté B^{ville}
- Par intervalle: $C^{seg(x)}$
  - C est triée selon x
  - seg(age) → numéro de segment du domaine de l'âges
    - Exple: val C = USERS.sort("age"), noté $C^{seg(age)}$

Répartition des données par hachage



- Clé de répartition = l'attribut **s**

Répartir les données sur les machines d'un cluster

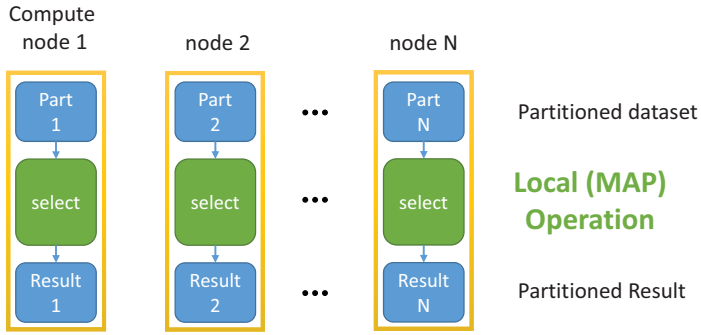


Opération locale

- Opération locale à un élément d'une collection
  - map, select
  - filter, where
- Opération locale aux éléments d'une seule partition
  - textFile
  - aggrégation
  - jointure sur la clé de répartition...

- Une opération locale **préserve** la clé de répartition
  - val User18 = USERS.where("age<18")
    - USERS$^{\varnothing}$ → User18 $^{\varnothing}$
    - USERS^{ville} → User18^{ville}

Exécution d'une opération locale

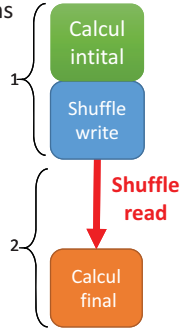


Opération globale

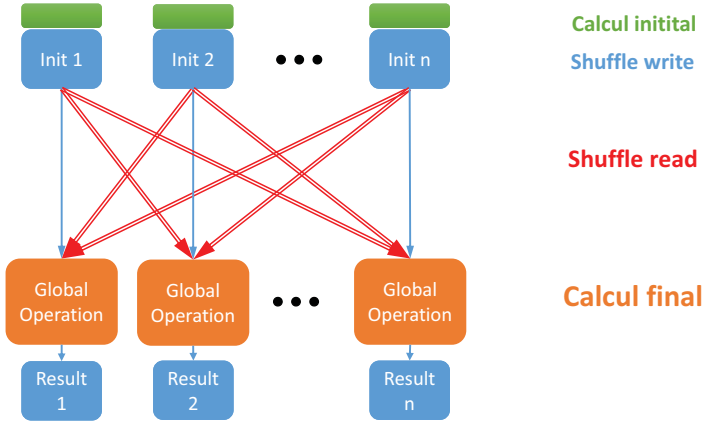
- Doit lire le résultat d'une opération locale évaluée sur plusieurs éléments d'une collection
  - ces éléments peuvent être dans différentes partitions
- Opération définie par
  1) un calcul initial préparant les données à envoyer

  2) un transfert des données servant au calcul final

Exemples
- reduce, reduceByKey
- groupBy, groupByKey, zipWithIndex
- join, cogroup
- sort, distinct, ...

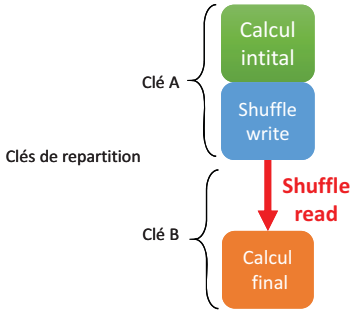


Exécution d'une opération globale



Changement de clé de répartition

- Une opération **globale** peut **modifier** la clé de répartition
  - Donnée^{A} → Résultat^{B}



  - Exemple

    val NotesParis = USERS.where(ville="Paris").join(NOTES,numU)

    On a USERS$^{\emptyset}$, NOTES$^{\emptyset}$, et NotesParis^{numU}

Opération → Clé de répartition

Opération	Clé de répartition
A.repartition(att)	att
A.join(B, att)	att
A.groupBy(att)	att
A.distinct	A
A.dropDuplicates(att)	att
A.intersect(B)	A
A.reduceByKey(k)	k
filter, where	préservée
mapValue	préservée
map, select	perdue
A.mapPartition	perdue / préservée (user defined)

Transfert dépend des clés de répartition

- Ne pas re-répartir les données déjà "bien" réparties
  - Réduire les transferts
- Cas de la jointure
  - Répartition round robin par défaut des Users et Notes
    - val f1 = NOTES.join(USERS, "numU")
  - Notes déjà réparties
    - val N = NOTES.repartition("numU").persist() N.count()
    - val f2 = N.join(USERS, "numU")
  - Users déjà répartis
    - val U = USERS.repartition("numU").persist() U.count()
    - val f3 = NOTES.join(U, "numU")
  - Users et Notes déjà répartis
    - Val f4 = N.join(U, "numU")

Exécution d'une expression composée

- Expression
  - Composition d'opérations locales ou globales
- Plan : graphe d'opérations de transformation
  - Graphe orienté, acyclique et avec racine
  - La racine est l'**action** finale du plan
    - Exple : count, take, collect
- Terminologie
  - Plan = *Job*, Opération = *Task*, *Graphe = DAG*

Exécution d'un plan par étapes

- Un plan a plusieurs étapes
  - Terminologie: Plan=*Job*, Etape=*Stage*
- Une étape
  - Bloc d'opérations **locales** consécutives
- Découpage du plan en étapes
  - Frontière: transfert compris dans une opération globale
    = **changement de la clé de répartition**
- Précédence entre les étapes
  - Début d'une étape
    - recevoir les données préparées par l'étape précédente
  - Fin d'une étape
    - préparer les données pour l'étape suivante

Exemple d'étapes

- Ex1
  - val a = USERS.where("age<30").select("ville").distinct()
- Ex2

val numExperts = AVIS.groupBy("numU").count().where("count >200").

select("numU")

  - numExperts.count

val experts = numExperts.join(USERS, "numU")

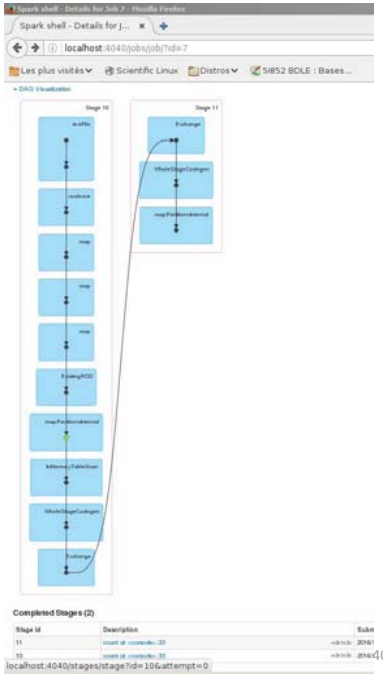
  - experts.columns
  - experts.count

Visualiser un plan d'exécution

- Interface graphique GUI:
  - URL localhost:4040
  - Etapes = blocs juxtaposés horizontalement
  - Transfert = arcs entre les blocs
- Résultats intermédiaires
  - Données persistantes
    - Une expression qui précède un point de persistance n'est pas ré-évaluée.
    - Persistance représenté par un noeud vert dans le graphe
  - Données transférées lors d'un shuffle read
    - Une expression qui précède un shuffle write n'est pas ré-évaluée.
  - Expression non re-évaluée
    - Représentée en "grisé": skiped stage

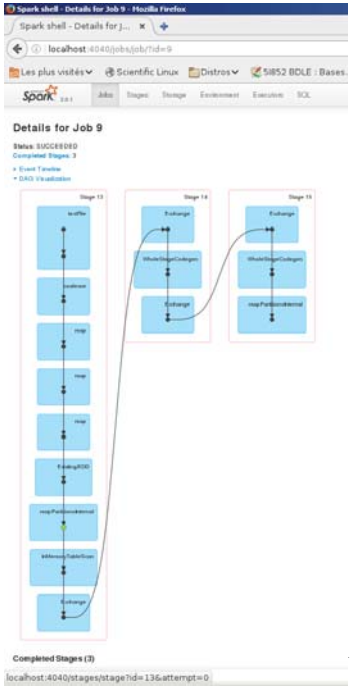
Exemple de plan

val a1 = AVIS.where("numF=1")
a1.count



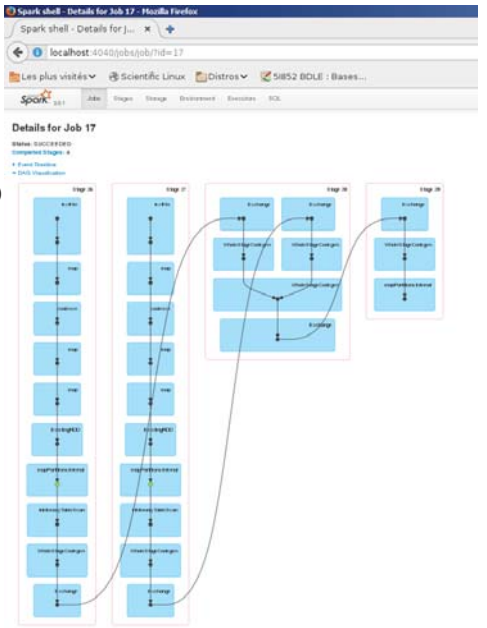
Exemple de plan

```
val s = AVIS.sort(col("note").desc)
s.count
```



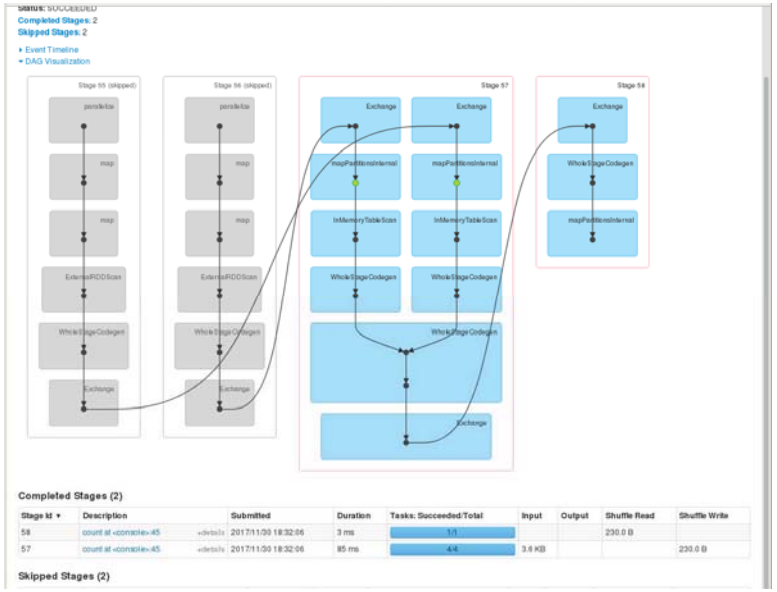
Exemple de plan

```
val t1 = TRIPLES.where("prop = '<book.book.characters>'")
t1.take(3).foreach(println)

val t2 = TRIPLES.where("prop = '<book.book.genre>'")
t2.take(3).foreach(println)

val j1 = t1.join(t2, "sujet")
j1.count
```



- Données partitionnées par round robin

- Données partitionnées par sujet (personne)



Diverses ref

- **Apache Spark: A Unified Engine for Big Data Processing**
  - https://vimeo.com/185645796
  - http://cacm.acm.org/magazines/2016/11/209116-apache-spark/fulltext