BigData - Semaine 4

Pierre Nerzic

février-mars 2019

Le cours de cette semaine présente le système de programmation Spark, un autre mécanisme pour écrire des programmes de type MapReduce sur HDFS, nettement plus performant et plus polyvalent que YARN.

Pierre Nerzic

Introduction

Pierre Nerzic

Présentation de Spark

Spark est une API de programmation parallèle sur des données.

L'objet principal de Spark est le RDD : *Resilient Distributed Dataset*. C'est un dispositif pour traiter une collection de données par des algorithmes parallèles robustes. Un RDD ne contient pas vraiment de données, mais seulement un traitement.

Ce traitement n'est effectué que lorsque cela apparaît nécessaire. On appelle cela l'évaluation paresseuse. D'autre part, Spark fait en sorte que le traitement soit distribué sur le cluster, donc calculé rapidement, et n'échoue pas même si des machines tombent en panne.

Avantages de Spark

Spark permet d'écrire des traitements complexes composés de plusieurs phases *map-reduce*. On peut le faire également avec YARN, mais les données issues de chaque phase doivent être stockées sur HDFS, pour être réutilisées immédiatement après dans la phase suivante. Cela prend beaucoup de temps et d'espace.

Les jobs YARN sont assez longs à lancer et exécuter. Il y a des temps de latence considérables.

Au contraire, Spark utilise beaucoup mieux la mémoire centrale des machines du cluster et gère lui-même l'enchaînement des tâches.

Les traitements peuvent être écrits dans plusieurs langages : Scala, Java et Python. On utilisera ce dernier pour sa simplicité pédagogique et le fait que vous l'apprenez dans d'autres cours.

Premier exemple Spark

Soit un fichier de données de type CSV provenant de http://opendata.paris.fr décrivant des arbres remarquables à Paris. Chaque ligne décrit un arbre : position GPS, arrondissement, genre, espèce, famille, année de plantation, hauteur, circonférence, etc. Le séparateur est `';'`. La première ligne contient les titres.

On souhaite afficher l'année de plantation (champ n°6) de l'arbre le plus grand (champ n°7).

Avec des commandes Unix, ce traitement s'écrirait : 

```
cat arbres.csv | cut -d';' -f6,7 | egrep -v 'HAUTEUR|;$' |\
    sort -t';' -k2 -n -r | head -n 1
```

Par contre, j'aurais apprécié que cut permette de changer l'ordre des champs, ça aurait facilité le classement.

Principe du traitement

Voyons comment faire la même chose avec Spark. Une fois que le fichier `arbres.csv` est placé sur HDFS, il faut :

1. séparer les champs de ce fichier.
2. extraire le 7e et 6e champs dans cet ordre – ce sont la hauteur de l'arbre et son année de plantation. On en fait une paire (clé, valeur). La clé est la hauteur de l'arbre, la valeur est son année de plantation.
3. éliminer la clé correspondant à la ligne de titre du fichier et les clés vides (hauteur inconnue).
4. convertir les clés en `float`
5. classer les paires selon la clé dans l'ordre décroissant.
6. afficher la première des paires. C'est le résultat voulu.

Programme pySpark

Voici le programme « pySpark » arbres.py :

```
#!/usr/bin/python
from pyspark import SparkConf, SparkContext
sc = SparkContext(conf=SparkConf().setAppName("arbres"))

arbres = sc.textFile("hdfs:/share/paris/arbres.csv")
tableau = arbres.map(lambda ligne: ligne.split(';'))
paires = tableau.map(lambda champs: (champs[6],champs[5]))
pairesok1 = paires.filter(
    lambda (hauteur,annee): hauteur!='' and hauteur!='HAUTEUR')
pairesok2 = pairesok1.map(
    lambda (hauteur,annee): (float(hauteur), annee))
classement = pairesok2.sortByKey(ascending=False)
print classement.first()
```

Remarques

Les deux premières instructions consistent à extraire les données du fichier. C'est d'assez bas niveau puisqu'on travaille au niveau des lignes et des caractères.

Dans *MapReduce* sur YARN, ces aspects avaient été isolés dans une classe `Arbres` qui masquait les détails et fournissait des méthodes pratiques, comme `getHauteur` et `getAnnee`.

Comparé aux programmes *MapReduce* en Java, Spark paraît plus rustique. Mais c'est sa rapidité, entre 10 et 100 fois supérieure à YARN qui le rend extrêmement intéressant.

On programme en Spark un peu comme dans le TP2 : la problématique est 1) d'arriver à construire des RDD contenant ce dont on a besoin et 2) d'écrire des fonctions de traitement.

Pierre Nerzic

Fonction *lambda* ou fonction nommée ?

Le programme précédent fait appel à des *lambda*. Ce sont des fonctions sans nom (anonymes).

Voici une fonction avec un nom employée dans un *map* Python :

```
def double(nombre):
    return nombre * 2

map(double, [1,2,3,4])
```

Cela peut s'écrire également avec une *lambda* :

```
map(lambda nombre: nombre * 2, [1,2,3,4])
```

La syntaxe est `lambda paramètres: expression`. Ça crée une fonction qui n'a pas de nom mais qu'on peut placer dans un *map*.

Fonction *lambda* ou fonction nommée ?

Faut-il employer une *lambda* ou bien une fonction nommée ?

- Complexité
  - Une *lambda* ne peut pas contenir un algorithme complexe. Elles sont limitées à une expression seulement.
  - Au contraire, une fonction peut contenir des boucles, des tests, des affectations à volonté
- Lisibilité
  - Les *lambda* sont beaucoup moins lisibles que les fonctions, impossibles à commenter, parfois cryptiques. . .
- Praticité
  - Les *lambda* sont très courtes et plus pratiques à écrire sur place, tandis que les fonctions doivent être définies ailleurs que là où on les emploie.

Pierre Nerzic

Fonction *lambda* ou fonction nommée ?

En conclusion :

- Les *lambda* sont intéressantes à connaître, plutôt pratiques
- On ne les emploiera que pour des expressions très simples, immédiates à comprendre.

Exemples pySpark (extrait de l'exemple initial) :

```
# chaque ligne est découpée en liste de mots
...map(lambda ligne: ligne.split(';'))
# on retourne un tuple composé des champs 6 et 5
...map(lambda champs: (champs[6],champs[5]))
# on ne garde que si clé n'est ni vide ni HAUTEUR
...filter(lambda (cle,val): cle!='' and cle!='HAUTEUR')
# on convertit la clé en float
...map(lambda (cle,val): (float(cle), val))
```


Pierre Nerzic

Fonction *lambda* ou fonction nommée ?

Le même exemple complet avec des fonctions nommées : 

```
def separ(ligne):
    return ligne.split(';')
def hauteurannee(champs):
    return (champs[6],champs[5])
def garderok( (cle,val) ):
    return cle!='' and cle!='HAUTEUR'
def convfloat( (cle, val) ):
    return (float(cle), val)

tableau = arbres.map(separ)
paires = tableau.map(hauteurannee)
pairesok1 = paires.filter(garderok)
pairesok2 = pairesok1.map(convfloat)
```

Les traitements sont éloignés de leur définition.

Pierre Nerzic

Dernière remarque sur les fonctions

Spark fait tourner les fonctions sur des machines différentes afin d'accélérer le traitement global. Il ne faut donc surtout pas affecter des variables globales dans les fonctions — elles ne pourront pas être transmises d'une machine à l'autre.

Chaque fonction doit être autonome, isolée. Donc ne pas faire :

```
total = 0
def cumuler(champs):
    global total
    total += float(champ[6])
    return champ[5]

annees = tableau.map(cumuler)
```

Il y a quand même des variables globales dans Spark, mais on n'en parlera pas dans ce cours.

Début d'un programme

Un programme pySpark doit commencer par ceci :

```
#!/usr/bin/python
from pyspark import SparkConf, SparkContext

nomappli = "essai1"
config = SparkConf().setAppName(nomappli)
#config.setMaster("spark://master:7077")
sc = SparkContext(conf=config)
```

`sc` représente le contexte Spark. C'est un objet qui possède plusieurs méthodes dont celles qui créent des RDD.

La ligne commentée `config.setMaster()` permet de définir l'URL du *Spark Master*, c'est à dire le cluster sur lequel lancer l'exécution.

Lancement

Spark offre plusieurs manières de lancer le programme, dont :

- Lancement en local :

  ```
  spark-submit ageMaxH.py
  ```

- Lancement sur un cluster de *Spark Workers* :

  ```
  spark-submit --master spark://master:7077 ageMaxH.py
  ```

  L'option `--master spark://master:7077` indique de faire appel au cluster de machines sur lesquelles tournent des *Spark Workers*. Ce sont des processus clients chargés de faire les calculs distribués pour Spark.

- Spark permet aussi de lancer l'exécution sur YARN :

  ```
  spark-submit --master yarn-cluster ageMaxH.py
  ```

Ce sont les esclaves YARN qui exécutent le programme Spark.

Éléments de l'API Spark

Pierre Nerzic

Principes

Spark est facile à apprendre car il repose sur des principes peu nombreux et simples. Consulter la documentation.

- Données :
  - **RDD** : ils représentent des données distribuées modifiées par une *transformation*, par exemple un *map* ou un *filter*.
  - Variables partagées entre des traitements et distribuées sur le cluster de machines.
- Méthodes :
  - **Transformations** : ce sont des fonctions (au sens mathématique) du type : RDD $\leftarrow$ *transformation*(RDD). Elles créent un nouveau RDD à partir d'un existant.
  - **Actions** : ce sont des fonctions qui permettent d'extraire des informations des RDD, par exemple les afficher sur l'écran ou les enregistrer dans un fichier.

RDD

Un RDD est une collection de données abstraite, résultant de la transformation d'un autre RDD ou d'une création à partir de données existantes. Un RDD est distribué, c'est à dire réparti sur plusieurs machines afin de paralléliser les traitements.

On peut créer un RDD de deux manières :

- Paralléliser une collection

Si votre programme contient des données itérables (tableau, liste. . .), elles peuvent devenir un RDD.

```
donnees = ['veau', 'vache', 'cochon', 'couvée']
RDD = sc.parallelize(donnees)
```

On le nomme « collection parallélisée ».

RDD (suite)

- Jeux de données externes

Spark peut utiliser de nombreuses sources de données Hadoop : HDFS, HBase... et il connait de nombreux types de fichiers : texte et les formats Hadoop tels que SequenceFile vu en semaine 2. Il y a d'autres formats de lecture. Consulter la documentation.

Voici comment lire un simple fichier texte ou CSV : 

```
RDD = sc.textFile("hdfs:/share/data.txt")
```

Comme avec MapReduce, chaque ligne du fichier constitue un enregistrement. Les transformations appliquées sur le RDD traiteront chaque ligne séparément. Les lignes du fichier sont distribuées sur différentes machines pour un traitement parallèle.

Lire et écrire des *SequenceFile*

Certains traitements Spark font appel à la notion de paires (clé,valeur). C'est le cas de l'exemple initial. Les clés permettent par exemple de classer des valeurs dans un certain ordre.

Pour stocker efficacement ce genre de RDD, on emploie un *SequenceFile*. Voir la documentation Hadoop.

- Lecture d'un `SequenceFile` dans un RDD

Cette fonction lit les paires du fichier et crée un RDD : 

```
RDD = sc.sequenceFile("hdfs:/share/data1.seq")
```

- Écriture d'un RDD dans un `SequenceFile`

Cette méthode enregistre les paires (clé, valeur) du RDD : 

```
RDD.saveAsSequenceFile("hdfs:/share/data2.seq")
```


Actions

Avant de voir les transformations, voyons les actions. Ce sont des méthodes qui s'appliquent à un RDD pour retourner une valeur ou une collection.

- `liste = RDD.collect()` retourne le RDD sous forme d'une liste Python. Attention à la taille si c'est du BigData.
- `nombre = RDD.count()` retourne le nombre d'éléments
- `premier = RDD.first()` retourne le premier élément
- `premiers = RDD.take(n)` retourne les *n* premiers éléments. Note: il n'y a pas de méthode `last` pour retourner le ou les derniers éléments.
- `resultat = RDD.reduce(fonction)` applique une fonction d'agrégation (associative) du type `fn(a,b)`→`c` 

```
grand = RDD.reduce(lambda a,b: max(a,b))
```


Transformations

Les RDD possèdent plusieurs méthodes qui ressemblent aux fonctions `map`, `filter`, *etc.* de Python.

En Python ordinaire, `map` est une fonction dont le premier paramètre est une *lambda* ou le nom d'une fonction, le second paramètre est la collection à traiter :

```
liste = [1,2,3,4]
doubles = map(lambda n: n*2, liste)
```

En pySpark, `map` est une méthode de la classe RDD, son seul paramètre est une *lambda* ou le nom d'une fonction :

```
liste = sc.parallelize([1,2,3,4])
doubles = liste.map(lambda n: n*2)
```

Les deux retournent les résultats, liste ou RDD.

Transformations de type *map*

Chacune de ces méthodes retourne un nouveau RDD à partir de celui qui est concerné (appelé `self` en Python).

- `RDD.map(fonction)` : chaque appel à la fonction doit retourner une valeur qui est mise dans le RDD sortant.

```
RDD = sc.parallelize([1,2,3,4])
print RDD.map(lambda n: n+1).collect()
```

- `RDD.filter(fonction)` : la fonction retourne un booléen. Il ne reste du RDD que les éléments pour lesquels la fonction retourne `True`.

```
RDD = sc.parallelize([1,2,3,4])
print RDD.filter(lambda n: (n%2)==0).collect()
```

Transformations de type *map* (suite)

- `RDD.flatMap(fonction)` : chaque appel à la fonction doit retourner une liste (vide ou pas) et toutes ces listes sont concaténées dans le RDD sortant.

```
RDD = sc.parallelize([0,1,2,3])
print RDD.flatMap(lambda n: [n*2, n*2+1]).collect()
```

La fonction lambda retourne une liste, le double et le double+1 du nombre traité. Appliquée à la collection par un simple *map*, on obtiendrait la liste imbriquée : `[[0, 1], [2, 3], [4, 5], [6, 7]]`. Avec un *flatMap*, les résultats sont concaténés ensemble, donc on récupère : `[0, 1, 2, 3, 4, 5, 6, 7]`.

Transformations ensemblistes

Ces transformations regroupent deux RDD, `self` et celui passé en paramètre.

- `RDD.distinct()` : retourne un seul exemplaire de chaque élément.

```
RDD = sc.parallelize([1, 2, 3, 4, 6, 5, 4, 3])
print RDD.distinct().collect()
```

- `RDD1.union(RDD2)` : contrairement à son nom, ça retourne la concaténation et non pas l'union des deux RDD. Rajouter `distinct()` pour faire une vraie union.

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print RDD1.union(RDD2).collect()
print RDD1.union(RDD2).distinct().collect()
```

Transformations ensemblistes (suite)

- `RDD1.intersection(RDD2)` : retourne l'intersection des deux RDD.

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print RDD1.intersection(RDD2).collect()
```


Pierre Nerzic

Transformations sur des paires (clé, valeur)

Les transformations suivantes manipulent des RDD dont les éléments sont des paires (clé, valeur) (*(K,V)* en anglais)

- `RDD.groupByKey()` : retourne un RDD dont les éléments sont des paires (clé, liste des valeurs ayant cette clé dans le RDD concerné).
- `RDD.sortByKey(ascending)` : retourne un RDD dont les clés sont triées. Mettre `True` ou `False`.
- `RDD.reduceByKey(fonction)` : regroupe les valeurs ayant la même clé et leur applique la fonction `(a,b)→c`.

```
RDD = sc.parallelize([ (1,"paul"),(2,"anne"),
    (1,"emile"),(2,"marie"),(1,"victor") ])
print RDD.reduceByKey(lambda a,b: a+"-"+b).collect()
```

retourne `[(1, "paul-emile-victor"), (2, "anne-marie")]`

Transformations de type jointure

Spark permet de calculer des jointures entre RDD1={(K1,V1)...} et RDD2={(K2,V2)...} et partageant des clés K identiques.

- `RDD1.join(RDD2)` : retourne toutes les paires (K, (V1, V2)) lorsque V1 et V2 ont la même clé.
- `RDD1.leftOuterJoin(RDD2)` : retourne les paires (K, (V1, V2)) ou (K, (V1, None)) si (K,V2) manque dans RDD2
- `RDD1.rightOuterJoin(RDD2)` : retourne les paires (K, (V1, V2)) ou (K, (None, V2)) si (K,V1) manque dans RDD1
- `RDD1.fullOuterJoin(RDD2)` : retourne toutes les paires (K, (V1, V2)), (K, (V1, None)) ou (K, (None, V2))

```
RDD1 = sc.parallelize([ (1,"tintin"),(2,"asterix"),(3,"spirou")
RDD2 = sc.parallelize([ (1,1930),(2,1961),(1,1931),(4,1974) ])
print RDD1.join(RDD2).collect()
```

SparkSQL

Présentation

SparkSQL rajoute une couche simili-SQL au dessus des RDD de Spark. Ça s'appuie sur deux concepts :

DataFrames Ce sont des tables SparkSQL : des données sous forme de colonnes nommées. On peut les construire à partir de fichiers JSON, de RDD ou de tables Hive (voir le dernier cours).

RDDSchema C'est la définition de la structure d'un DataFrame. C'est la liste des colonnes et de leurs types. Un RDDSchema peut être défini à l'aide d'un fichier JSON.

Il y a des liens entre DataFrame et RDD. Les RDD ne sont que des données, des n-uplets bruts. Les DataFrames sont accompagnées d'un schéma.

Début d'un programme

Un programme pySparkSQL doit commencer par ceci : 

```
#!/usr/bin/python
from pyspark import SparkConf, SparkContext, SQLContext
from pyspark.sql.functions import *

nomappli = "essai1"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)
sqlContext = SQLContext(sc)
```

`sqlContext` représente le contexte SparkSQL. C'est un objet qui possède plusieurs méthodes dont celles qui créent des DataFrames et celles qui permettent de lancer des requêtes SQL.

Créer un DataFrame

Il y a plusieurs manières de créer un DataFrame. Il faut à la fois fournir le schéma (noms et types des colonnes) et les données. L'une des méthodes simples consiste à utiliser un fichier JSON.

Un fichier JSON est pratique car il contient à la fois les données et le schéma, mais ça ne convient que pour de petites données.

Un fichier JSON contient la sérialisation d'une structure de données JavaScript. Pour les données qui nous intéressent, c'est simple. Chaque n-uplet est englobé par `{...}` ; les champs sont écrits `"nom":"valeur"`. Voici un exemple de trois n-uplets : 

```
{"nom":"Paul"}
{"nom":"Émile", "age":30}
{"nom":"Victor", "age":19}
```


Créer un DataFrame à partir d'un fichier JSON

Voici comment créer un DataFrame :

```
df = sqlContext.read.json("fichier.json")
```

Elles retournent un DataFrame `df` contenant les données. Voir plus loin ce qu'on peut en faire.

A savoir qu'un DataFrame ainsi créé ne connaît pas les types des colonnes, seulement leurs noms.

Créer un DataFrame à partir d'un RDD

C'est plus compliqué car il faut indiquer le schéma. Un schéma est une liste de `StructField`. Chacun est un couple (nom, type).

```
# création d'un RDD sur le fichier personnes.csv
fichier = sc.textFile("hdfs:/tmp/personnes.csv")
tableau = fichier.map(lambda ligne: ligne.split(";"))
# définition du schéma
champ1 = StructField("nom",    StringType)
champ2 = StructField("prenom", StringType)
champ3 = StructField("age",    IntType)
schema = [champ1, champ2, champ3]
# création d'un DataFrame sur le RDD
personnes = sqlContext.createDataFrame(tableau, schema)
```

`personnes` est un DataFrame contenant les données et le schéma.

Extraction d'informations d'un DataFrame

Il est facile d'extraire une colonne d'un DataFrame :

```
colonneAge = personnes.age
```

Note: si une propriété est vide ou vaut `null`, Python voit `None`.

La propriété `columns` retourne la liste des noms des colonnes :

```
print personnes.columns
```

La classe DataFrame possède de nombreuses méthodes qui seront présentées plus loin, page 42.

Pierre Nerzic

Donner un nom de table SQL à un DataFrame

Cela consiste à donner un nom désignant la future table SQL contenue dans le DataFrame. C'est une opération nécessaire pour exécuter des requêtes SQL. En effet, la variable `personnes` contenant le DataFrame ne connaît pas son propre nom.

```
personnes.registerTempTable("personnes")
```

Le DataFrame pourra être utilisé dans une requête SQL sous le nom `personnes`. Il est donc commode de remettre le même nom que le DataFrame.

NB: ce n'est qu'une table temporaire, elle disparaît à la fin du programme.

Pierre Nerzic

Exemple de requête SQL

Une fois que le DataFrame est rempli et nommé, on peut l'interroger. Il y a plusieurs moyens. Le premier est d'écrire directement une requête SQL.

```
resultat = sqlContext.sql("SELECT nom FROM personnes")

for nuplet in resultat.collect():
  print nuplet.nom
```

Le résultat de la méthode `sql` est un nouveau DataFrame contenant les n-uplets demandés (documentation). On les affiche à l'aide d'une simple boucle et d'un appel à `collect()` comme en pySpark.

Un autre moyen pour écrire des requêtes est d'appeler les méthodes de l'API.

API SparkSQL

Aperçu

L'API SparkSQL pour Python est très complète. Elle comprend plusieurs classes ayant chacune de nombreuses méthodes :

`DataFrame`	représente une table de données relationnelles
`Column`	représente une colonne d'un DataFrame
`Row`	représente l'un des n-uplets d'un DataFrame

Ces classes permettent d'écrire une requête SQL autrement qu'en SQL, à l'aide d'appels de méthodes enchaînés.

Exemple de requête par l'API

Soit une table de clients (idclient, nom) et une table d'achats (idachat, idclient, montant). On veut afficher les noms des clients ayant fait au moins un achat d'un montant supérieur à 30.

En SQL, on l'écrirait :

```
SELECT DISTINCT nom FROM achats JOIN clients
    ON achats.idclient = clients.idclient
    AND achats.montant > 30.0;
```

En pySparkSQL :

```
resultat = achats.filter(achats.montant > 30.0)          \
    .join(clients, clients.idclient == achats.idclient) \
    .select("nom")                                       \
    .distinct()
```

Classe DataFrame

C'est la classe principale. Elle définit des méthodes à appliquer aux tables. Il y en a quelques unes à connaître :

- `filter(condition)` retourne un nouveau DataFrame qui ne contient que les n-uplets qui satisfont la condition. Cette condition peut être écrite dans une chaîne SQL ou sous forme d'une condition Python.

```
resultat = achats.filter("montant > 30.0")
resultat = achats.filter(achats.montant > 30.0)
```

Remarquer la différence de nommage des champs.

Méthodes de DataFrame

- `count()` retourne le nombre de n-uplets du DataFrame concerné.
- `distinct()` retourne un nouveau DataFrame ne contenant que les n-uplets distincts
- `limit(n)` retourne un nouveau DataFrame ne contenant que les n premiers n-uplets
- `join(autre, condition, type)` fait une jointure entre `self` et autre sur la condition. Le type de jointure est une chaîne parmi `"inner"` (défaut), `"outer"`, `"left_outer"`, `"right_outer"` et `"semijoin"`
- `collect()` retourne le contenu de `self` sous forme d'une liste de `Row`. On peut l'utiliser pour un affichage final :

```
print achats.filter(achats.idclient == 1).collect()
```

Agrégation

- `groupBy(colonnes)` regroupe les n-uplets qui ont la même valeur pour les colonnes qui sont désignées par une chaîne SQL. Cette méthode retourne un objet appelé `GroupedData` sur lequel on peut appliquer les méthodes suivantes :
- `count()` : nombre d'éléments par groupe
- `avg(colonnes)` : moyenne des colonnes par groupe
- `max(colonnes)`, `min(colonnes)` : max et min des colonnes par groupe
- `sum(colonnes)` : addition des colonnes par groupe

```
tapc = achats.groupBy("idclient").sum("montant")
napc = achats.groupBy("idclient").count()
```

L'agrégation crée des colonnes appelées d'après la fonction : `"AVG(montant)"`, `"MAX(montant)"`, etc.

Classement

- `sort(colonnes)` classe les n-uplets de `self` selon les colonnes, dans l'ordre croissant. Si on spécifie la colonne par un nom pyspark (*table.champ*, on peut lui appliquer la *méthode* `desc()` pour classer dans l'ordre décroissant ; sinon, il faut employer la *fonction* `desc(colonnes)` pour classer dans l'ordre décroissant.

```
topa = achats.groupBy("idclient").sum("montant")    \
       .sort(desc("SUM(montant)")).first()
```

La méthode `first` retourne le premier n-uplet de `self`.