

BigData - Semaine 7

Pierre Nerzic

février-mars 2019

Le cours de cette semaine présente le système Pig et son langage Pig Latin. Ce dernier est un langage de programmation de requêtes sur des fichiers HDFS qui se veut plus simple que Java pour écrire des jobs MapReduce. Pig sert à lancer les programmes Pig Latin dans l'environnement Hadoop.

Introduction


Présentation de Pig

Apache Pig est un logiciel initialement créé par Yahoo!. Il permet d'écrire des traitements utiles sur des données, sans subir la complexité de Java. Le but est de rendre Hadoop accessible à des non-informaticiens scientifiques : physiciens, statisticiens, mathématiciens. . .

Pig propose un langage de scripts appelé « Pig Latin ». Ce langage est qualifié de « Data Flow Language ». Ses instructions décrivent des traitements sur un flot de données. Conceptuellement, ça ressemble à un tube Unix ; chaque commande modifie le flot de données qui la traverse. Pig Latin permet également de construire des traitements beaucoup plus variés et non-linéaires.

Pig traduit les programmes Pig Latin en jobs MapReduce et intègre les résultats dans le flot.

Exemple de programme Pig

Ce programme affiche les 10 adultes les plus jeunes extraits d'un fichier csv contenant 3 colonnes : identifiant, nom et age. 

```
personnes = LOAD 'personnes.csv' USING PigStorage(';')
           AS (userid:int, nom:chararray, age:int);
jeunesadultes = FILTER personnes BY age >= 18 AND age < 24;
classement = ORDER jeunesadultes BY age;
resultat = LIMIT classement 10;
DUMP resultat;
```

Pour l'exécuter : `pig programme.pig`. Ça lance un job MapReduce dans Hadoop. On peut aussi taper les instructions une par une dans le shell de Pig.

Le but de ce cours : comprendre ce script et en écrire d'autres.

Comparaison entre SQL et Pig Latin

Il y a quelques ressemblances apparentes entre SQL et Pig Latin. Il y a plusieurs mots clés en commun (JOIN, ORDER, LIMIT...) mais leur principe est différent :

- En SQL, on construit des requêtes qui décrivent les données à obtenir. On ne sait pas comment le moteur SQL va faire pour calculer le résultat. On sait seulement qu'en interne, la requête va être décomposée en boucles et en comparaisons sur les données et en utilisant au mieux les index.
- En Pig Latin, on construit des programmes qui contiennent des instructions. On décrit exactement comment le résultat doit être obtenu, quels calculs doivent être faits et dans quel ordre.

Également, Pig a été conçu pour les données incertaines de Hadoop, tandis que SQL tourne sur des SGBD parfaitement sains.

Langage Pig Latin

Structure d'un programme

Les commentaires sont placés entre `/*...*/` ou à partir de `--` et la fin de ligne.

Un programme Pig Latin est une succession d'instructions. Toutes doivent être terminées par un `;`

Comme dans SQL, il n'y a pas de notion de variables, ni de fonctions/procédures.

Le résultat de chaque instruction Pig est une collection de n-uplets. On l'appelle *relation*. On peut la voir comme une table de base de données.

Chaque instruction Pig prend une relation en entrée et produit une nouvelle relation en sortie.

```
sortie = INSTRUCTION entree PARAMETRES ...;
```


Exécution d'un programme

Lorsque vous lancez l'exécution d'un programme, Pig commence par l'analyser. Chaque instruction, si elle est syntaxiquement correcte, est rajoutée à une sorte de plan d'action, une succession de MapReduce, et c'est seulement à la fin du programme que ce plan d'action est exécuté en fonction de ce que vous demandez à la fin.

L'instruction `EXPLAIN relation` affiche le plan d'action prévu pour calculer la relation. C'est assez indigeste quand on n'est pas spécialiste.

Relations et alias

La syntaxe `nom = INSTRUCTION ... ;` définit un *alias*, c'est à dire un nom pour la relation créée par l'instruction. Ce nom étant généralement employé dans les instructions suivantes, c'est ça qui construit un flot de traitement.

```
nom1 = LOAD ... ;  
nom2 = FILTER nom1 ... ;  
nom3 = ORDER nom2 ... ;  
nom4 = LIMIT nom3 ... ;
```

Le même alias peut être réutilisé dans des instructions différentes, ce qui crée des bifurcations dans le flot de traitement : séparations ou regroupements.

Il n'est pas recommandé de réaffecter le même alias.

Enchaînement des instructions

Pig permet soit d'enchaîner les instructions par le mécanisme des alias, soit par un appel imbriqué.

```
nom4 = LIMIT (ORDER (FILTER (LOAD ...) ...) ...) ... ;
```

Vous choisirez celui que vous trouvez le plus lisible.

Toutefois, les appels imbriqués ne permettent pas de faire facilement des séparations de traitement, au contraire des alias :

```
nom1 = LOAD ... ;  
nom2 = FILTER nom1 ... ;  
nom3 = FILTER nom1 ... ;  
nom4 = JOIN nom2 ..., nom3 ;
```

Relations et types

Une *relation* est une collection ordonnée de n-uplets qui possèdent tous les mêmes champs. Voici les types possibles.

Les *types scalaires* sont :

- `int` et `long` pour les entiers, `float` et `double` pour les réels
- `chararray` pour des chaînes quelconques.
- `bytearray` pour des objets binaires quelconques

Il y a aussi trois *types complexes* :

- dictionnaires (*maps*) : `[nom#mickey, age#87]`
- n-uplets (*tuples*) de taille fixe : `(mickey, 87, hergé)`
- sacs (*bags*) = ensembles sans ordre de tuples : `{(mickey, 87), (asterix, 56), (tintin, 86)}`

Schéma d'une relation

La liste des champs d'une relation est appelé *schéma*. C'est un n-uplet. On l'écrit (nom1:type1, nom2:type2, ...)

Par exemple, une relation contenant des employés aura le schéma suivant :

```
(id:long, nom:chararray, prenom:chararray, photo:bytearray,  
  ancienneté:int, salaire:float)
```

L'instruction `LOAD 'fichier.csv' AS schéma;` permet de lire un fichier CSV et d'en faire une relation d'après le schéma indiqué.

Schémas complexes (tuples)

Pig permet la création d'une relation basée sur un schéma incluant des données complexes. Soit un fichier contenant des segments 3D :

```
S1 ⇨ (3,8,9) ⇨ (4,5,6)
S2 ⇨ (1,4,7) ⇨ (3,7,5)
S3 ⇨ (2,5,8) ⇨ (9,5,8)
```

J'utilise le caractère '⇨' pour représenter une tabulation.

Voici comment lire ce fichier :

```
segments = LOAD 'segments.csv' AS (
    nom:chararray,
    P1:tuple(x1:int, y1:int, z1:int),
    P2:tuple(x2:int, y2:int, z2:int));
DUMP segments;
```

Schémas complexes (bags)

On peut également lire des sacs, c'est à dire des ensembles de données de mêmes types mais en nombre quelconque :

```
L1 ⇨ {(3,8,9), (4,5,6)}
```

```
L2 ⇨ {(4,8,1), (6,3,7), (7,4,5), (5,2,9), (2,7,1)}
```

```
L3 ⇨ {(4,3,5), (6,7,1), (3,1,7)}
```

Le schéma de ce fichier est :

```
(nom:chararray, Points:{tuple(x:int, y:int, z:int)})
```

Explications :

- Le deuxième champ du schéma est spécifié ainsi :
« nom du champ » : { « type du contenu du sac » }
- Les données de ce champ doivent être au format
{ « liste de valeurs correspondant au type » }

Schémas complexes (maps)

Pour finir, avec les dictionnaires, voici le contenu du fichier `heros.csv` :

```
1 ↷ [nom#asterix,metier#guerrier]
2 ↷ [nom#tintin,metier#journaliste]
3 ↷ [nom#spirou,metier#groom]
```

On en fait une relation par :

```
heros = LOAD 'heros.csv' AS (id:int, infos:map[chararray])
```

Remarque : toutes ces constructions, tuple, map et bags peuvent être imbriquées, mais certaines combinaisons sont difficiles à spécifier.

Nommage des champs

Il y a deux syntaxes pour nommer les champs d'une relation. Soit on emploie leur nom en clair, soit on les désigne par leur position \$0 désignant le premier champ, \$1 le deuxième et ainsi de suite.

On emploie la seconde syntaxe quand les noms des champs ne sont pas connus ou qu'ils ont été générés dynamiquement.

Quand il y a ambiguïté sur la relation concernée, on préfixe le nom du champ par le nom de la relation : `relation.champ`

- Lorsqu'un champ est un *tuple*, ses éléments sont nommés `relation.champ.element`. Par exemple `segments.P1.z1`
- Pour un champ de type *map*, ses éléments sont nommés `relation.champ#element`. Par exemple `heros.infos#metier`
- Il n'y a pas de syntaxe pour l'accès aux champs de type *bag*.

Instructions Pig

Introduction

Il y a plusieurs catégories d'instructions : interaction avec les fichiers, filtrage, jointures. . .

Pour commencer, il y a également des instructions d'accès aux fichiers HDFS à l'intérieur de Pig. Ces commandes fonctionnent comme dans Unix ou comme le suggère leur nom et elles sont plutôt destinées à être tapées dans le shell de Pig.

- dossiers : `cd`, `ls`, `mkdir`, `rmf` (`rmf = rm -f -r`)
- fichiers : `cat`, `cp`, `copyFromLocal`, `copyToLocal`, `mv`, `rm`
- divers : `help`, `quit`, `clear`

Chargement et enregistrement de fichiers

- `LOAD 'fichier' USING PigStorage('sep') AS schema;`
Charge le fichier qui doit être au format CSV avec des champs séparés par `sep` et en leur attribuant les noms et types du schéma. Des données qui ne correspondent pas restent vides.
 - Il n'est pas nécessaire de mettre la clause `USING` quand le séparateur est la tabulation.
 - NB: le fichier doit être présent dans HDFS ; rien ne signale l'erreur autrement que l'échec du programme entier.
- `STORE relation INTO 'fichier' USING PigStorage('sep');`
Enregistre la relation dans le fichier, sous forme d'un fichier CSV séparé par `sep`, ex: `';', ':'`...

Dans les deux cas, si le fichier porte une extension `.gz`, `.bz2`, il est (dé)compressé automatiquement.

Affichage de relations

- `DUMP relation;`
Lance le calcul MapReduce de la relation et affiche les résultats à l'écran. C'est seulement à ce stade que la relation est calculée.
- `SAMPLE relation;`
Affiche quelques n-uplets choisis au hasard, une sorte d'échantillon de la relation.
- `DESCRIBE relation;`
Affiche le schéma, c'est à dire les champs, noms et types, de la relation. C'est à utiliser dès qu'on a un doute sur le programme.

Instruction ORDER

Elle classe les n-uplets dans l'ordre croissant (ou décroissant si DESC) des champs indiqués

```
ORDER relation BY champ [ASC|DESC], ...
```

Ça ressemble totalement à ce qu'on fait avec SQL. Voir l'exemple du transparent suivant.


```
RANK relation BY champ [ASC|DESC], ...
```

Retourne une relation ayant un premier champ supplémentaire, le rang des n-uplets par rapport au critère indiqué.


Instruction LIMIT

Elle ne conserve de la relation que les N premiers n-uplets.

LIMIT relation N

On l'utilise en général avec ORDER. Par exemple, cette instruction affiche les 10 plus gros achats : 

```
triparmontant = ORDER achats BY montant DESC;  
meilleurs = LIMIT triparmontant 10;  
DUMP meilleurs;
```

On peut aussi l'écrire de manière imbriquée : 


```
DUMP LIMIT (ORDER achats BY montant DESC) 10;
```

Instruction FILTER

Elle sert à créer une relation ne contenant que les n-uplets qui vérifient une condition.

```
FILTER relation BY condition;
```

La condition :

- comparaisons : mêmes opérateurs qu'en C et en Java
- nullité (vide) d'un champ : IS NULL, IS NOT NULL
- connecteurs logiques : (mêmes opérateurs qu'en SQL) AND, OR et NOT 

```
clients = LOAD 'clients.csv'  
        AS (idclient:int, age:int, adresse:chararray);  
tresvieuxclients = FILTER clients  
        BY age > 1720 AND adresse IS NOT NULL;
```


Instruction DISTINCT

Elle sert à supprimer les n-uplets en double. Cela ressemble à la commande Unix `uniq` (sauf qu'ils n'ont pas besoin d'être dans l'ordre).

```
DISTINCT relation;
```

Note: la totalité des champs des tuples sont pris en compte pour éliminer les doublons.

Si on veut éliminer les doublons en se basant sur une partie des champs, alors il faut employer un `FOREACH`, voir plus loin.

Instruction FOREACH GENERATE

C'est une instruction qui peut être très complexe. Dans sa forme la plus simple, elle sert à générer une relation à partir d'une autre, par exemple faire une projection.

```
FOREACH relation GENERATE expr1 AS champ1, ...;
```

Crée une nouvelle relation contenant les champs indiqués. Ça peut être des champs de la relation fournie ou des valeurs calculées ; la clause AS permet de leur donner un nom.

Exemple, on génère des bons d'achats égaux à 5% du total des achats des clients :



```
bonsachat = FOREACH totalachatsparclient  
    GENERATE idclient, montant*0.05 AS bon;
```

Énumération de champs

Lorsqu'il y a de nombreux champs dans la relation d'entrée et aussi dans celle qu'il faut générer, il serait pénible de tous les écrire. Pig propose une notation `champA .. champB` pour énumérer tous les champs compris entre les deux. Si `champA` est omis, alors ça part du premier ; si `champB` est omis, alors l'énumération va jusqu'au dernier champ.

Par exemple, une relation comprend 20 champs, on veut seulement retirer le 8^e :



```
relation_sans_champ8 = FOREACH relation_complexe  
    GENERATE .. champ7 champ9 ..;
```

Attention, il faut écrire deux points bien espacés.

Instruction GROUP BY

L'instruction `GROUP relation BY champ` rassemble tous les tuples de la relation qui ont la même valeur pour le champ. Elle construit une nouvelle relation contenant des couples (`champ`, {tuples pour lesquels *champ* est le même}).

Soit une relation appelée `achats` (`idachat`, `idclient`, `montant`) :

```
1, 1, 12.50  
2, 2, 21.75  
3, 3, 56.25  
4, 1, 34.00  
5, 3, 3.30
```

`GROUP achats BY idclient` produit ceci :

```
(1, {(4,1,34.0), (1,1,12.5)})  
(2, {(2,2,21.75)})  
(3, {(3,3,56.25), (5,3,3.30)})
```

Remarque sur GROUP BY

L'instruction GROUP BY crée une nouvelle relation composée de couples dont les champs sont nommés :

- *group* : c'est le nom donné au champ qui a servi à construire les groupements, on ne peut pas changer le nom
- *relation* : le nom de la relation groupée est donnée au *bag*. Il contient tous les n-uplets dont le champ BY a la même valeur.

Il aurait été souhaitable que GROUP achats BY idclient produise des couples (idclient, achats). Mais non, ils sont nommés (group, achats). Donc si on fait

```
achatsparclient = GROUP achats BY idclient;
```

on devra mentionner `achatsparclient.group` et `achatsparclient.achats`.

Instruction GROUP ALL

L'instruction `GROUP relation ALL` regroupe tous les n-uplets de la relation dans un seul n-uplet composé d'un champ appelé `group` et valant `all`, et d'un second champ appelé comme la relation à grouper.


```
montants = FOREACH achats GENERATE montant;  
montants_tous = GROUP montants ALL;
```

Crée ce seul n-uplet : `(all, {(12.50), (21.75), (56.25), (34.00), (3.30)})`

Utilisation de GROUP BY et FOREACH

On utilise en général FOREACH pour traiter le résultat d'un GROUP. Ce sont des couples (rappel) :

- Le premier champ venant du GROUP BY s'appelle group,
- Le second champ est un sac (*bag*) contenant tous les n-uplets liés à group. Ce sac porte le nom de la relation qui a servi à le créer.

On utilise FOREACH pour agréger le sac en une seule valeur, par exemple la somme de l'un des champs. 

```
achats = LOAD 'achats.csv' AS (idachat, idclient, montant);
achatsparclient = GROUP achats BY idclient;
totalachatsparclient = FOREACH achatsparclient
    GENERATE group AS idclient, SUM(achats.montant) AS total;
```

Opérateurs


Pig propose plusieurs opérateurs permettant d'agréger les valeurs d'un sac. Le sac est construit par un `GROUP BY` ou `GROUP ALL`.

- `SUM,AVG` calcule la somme/moyenne des valeurs numériques.
- `MAX,MIN` retournent la plus grande/petite valeur
- `COUNT` calcule le nombre d'éléments du sac sans les `null`
- `COUNT_STAR` calcule le nombre d'éléments avec les `null`

Il y a d'autres opérateurs :

- `CONCAT(v1, v2, ...)` concatène les valeurs fournies.
- `DIFF(sac1, sac2)` compare les deux sacs et retourne un sac contenant les éléments qui ne sont pas en commun.
- `SIZE` retourne le nombre d'éléments du champ fourni.

Utilisation de GROUP et FOREACH

Dans certains cas, on souhaite aplatir le résultat d'un GROUP, c'est à dire au lieu d'avoir des sacs contenant tous les n-uplets regroupés, on les veut tous à part. Ça va donc créer autant de n-uplets séparés qu'il y avait d'éléments dans les sacs. 

```
achatsparclient = GROUP achats BY idclient;  
plusieurs = FILTER achatsparclient BY COUNT(achats)>1;  
clientsmultiples = FOREACH plusieurs  
    GENERATE group AS idclient, FLATTEN(achats.montant);
```


produit ceci, les achats des clients qui en ont plusieurs :

```
(1, 1, 34.0)  
(4, 1, 12.5)  
(3, 3, 56.25)  
(5, 3, 3.30)
```

Instruction FOREACH GENERATE complexe

FOREACH relation { traitements... ; GENERATE ... }
permet d'insérer des traitements avant le GENERATE.

- sousrelation = FILTER relation BY condition;

Exemple, on veut offrir un bon d'achat seulement aux clients qui ont fait de gros achats, le bon d'achat étant égal à 15% du montant total de ces gros achats : 

```
achatsparclient = GROUP achats BY idclient;
grosbonsachat = FOREACH achatsparclient {
    grosachats = FILTER achats BY montant>=30.0;
    GENERATE group, SUM(grosachats.montant)*0.15 AS grosbon;
};
```

La relation achat du FILTER désigne le second champ du GROUP achatsparclient traité par le FOREACH.

Instruction FOREACH GENERATE complexe (suite)

Il est possible d'imbriquer d'autres instructions dans un FOREACH :


- `sousrelation = FOREACH relation GENERATE...`;
- `sousrelation = LIMIT relation N`;
- `sousrelation = DISTINCT relation`;

Exemple, on cherche les clients ayant acheté des produits différents :

```
achats = LOAD 'achats.csv'  
        AS (idachat:int, idclient:int, idproduit:int, montant:float)  
achatsparclient = GROUP achats BY idclient;  
nombreparclient = FOREACH achatsparclient {  
    produits = FOREACH achats GENERATE idproduit;  
    differents = DISTINCT produits;  
    GENERATE group AS idclient, COUNT(differents) AS nombre;  
}  
resultat = FILTER nombreparclient BY nombre>1;
```

DISTINCT sur certaines propriétés

On a vu que l'instruction `DISTINCT` filtre seulement les n-uplets exactement identiques. Voici comment supprimer les n-uplets en double sur certains champs seulement. Ça veut dire qu'on garde seulement l'un des n-uplets au hasard quand il y en a plusieurs qui ont les mêmes valeurs sur les champs considérés.

Soit une relation `A` de schéma `(a1,a2,a3,a4,a5)`. On veut que les triplets `(a1,a2,a3)` soient uniques. 

```
A = LOAD 'data.csv' AS (a1,a2,a3,a4,a5);
A_unique = FOREACH (GROUP A BY (a1,a2,a3)) {
    seul = LIMIT A 1;
    GENERATE group.a1,group.a2,group.a3,
             FLATTEN(seul.a4),FLATTEN(seul.a5);
}
```

Instruction JOIN

Les jointures permettent, comme en SQL de créer une troisième table à partir de plusieurs tables qui ont un champ en commun.

```
JOIN relation1 BY champ1a, relation2 BY champ2b, ...
```

Cela crée une relation ayant autant de champs que toutes les relations mentionnées. Les n-uplets qu'on y trouve sont ceux du produit cartésien entre toutes ces relations, pour lesquels le champ1a de la relation1 est égal au champ2b de la relation2.

Dans la nouvelle relation, les champs sont nommés `relation1::champ1a`, `relation1::champ1b`, ...

Exemple de jointure

Soit une relation `clients` (`idclient`, `nom`) :

```
1 lucien  
2 andré  
3 marcel
```

Et une relation `achats` (`idachat`, `idclient`, `montant`) :

```
1, 1, 12.50  
2, 2, 21.75  
3, 3, 56.25  
4, 1, 34.00  
5, 3, 3.30
```

Exemple de jointure (suite)

L'instruction JOIN clients BY idclient, achats BY idclient crée (idclient, nom, idachat, idclient, montant) :

```
(1,lucien,4,1,34.25)
(1,lucien,1,1,12.5)
(2,andr ,2,2,21.75)
(3,marcel,3,3,56.25)
(3,marcel,5,3,3.30)
```

Il y a d'autres types de jointure :

- JOIN relationG BY champG LEFT, relationD BY champD pour une jointure   gauche
- JOIN relationG BY champG RIGHT, relationD BY champD pour une jointure   droite
- CROSS relation1, relation2, ... produit cart sien

Instruction UNION


Cette instruction regroupe les n-uplets des relations indiquées.

```
UNION ONSCHEMA relation1, relation2, ...
```

Il est très préférable que les relations aient les mêmes schémas.
Chaque champ crée sa propre colonne.

Conclusion

Comparaison entre SQL et Pig (le retour)

Revenons sur une comparaison entre SQL et Pig. Soit une petite base de données de clients et d'achats. La voici en SQL ; en Pig, ça sera deux fichiers CSV. 

```
CREATE TABLE clients (  
  idclient INTEGER PRIMARY KEY,  
  nom VARCHAR(255));  
CREATE TABLE achats (  
  idachat INTEGER PRIMARY KEY,  
  idclient INTEGER,  
  FOREIGN KEY (idclient) REFERENCES clients(idclient),  
  montant NUMERIC(7,2));
```

On veut calculer le montant total des achats par client.

Affichage nom et total des achats

Voici la requête SQL :



```
SELECT nom, SUM(montant) FROM clients JOIN achats
      ON clients.idclient = achats.idclient
      GROUP BY clients.idclient;
```

C'est très différent en Pig Latin, à méditer :



```
achats = LOAD 'achats.csv' AS (idachat, idclient, montant);
achatsparclients = GROUP achats BY idclient;
totaux = FOREACH achatsparclients
          GENERATE group, SUM(achats.montant) AS total;
clients = LOAD 'clients.csv' AS (idclient, nom);
jointure = JOIN clients BY idclient, totaux BY group;
resultat = FOREACH jointure GENERATE nom, total;
DUMP resultat;
```