

1)

## Big Data

- hadoop est basé sur :

↳ hdfs : qui gère le stockage de données.

↳ yarn/mapreduce : qui répond à comment un programme peut exécuter et traiter plusieurs données.

↳ autres outils :

→ Zookeeper : simplifie les opérations de traitements sur les données et aussi la gestion et coordination de la plateforme.

→ Ambari : outil pour la gestion et monitoring des cluster.

→ "plusieurs machines qui fonctionnent comme une seule".

- hdfs :

- hdfs est un sys. de fichiers conçu pour stocker des fichiers très volumineux dans un cluster.

- stocke les fichiers sur plusieurs nœuds qui forment un cluster.

- réplique les données sur plusieurs nœuds afin d'éviter leur perte en cas de panne de certains nœuds => fiabilité, disponibilité

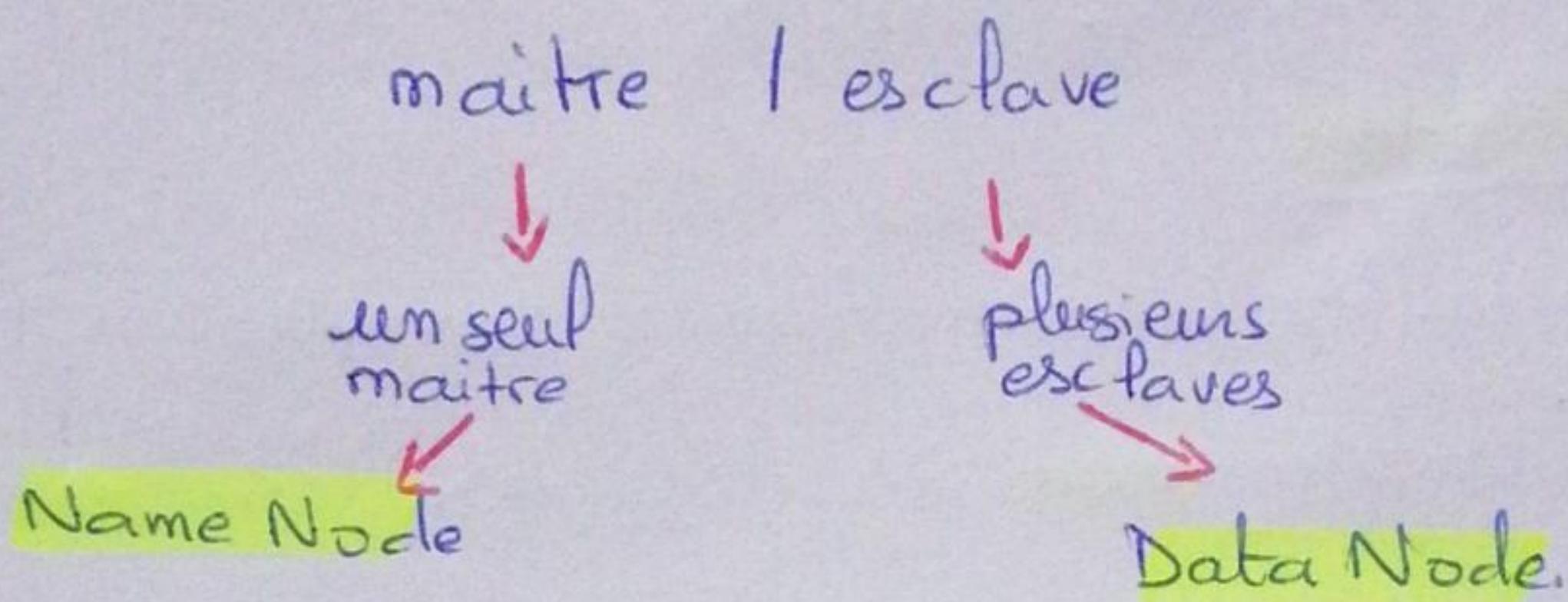
- chaque fichier est décomposé en plusieurs nœuds: blocs.

- la taille du bloc ↑ le temps ↓ pour les localiser.

- taille du bloc est 128 mo (v2), 64 mo (v1).

2) - **had** **hdfs** qui décide la taille du bloc, il les distribue et c'est lui qui les localise.

## Architecture HDFS:



### ↳ Name Node :

↳ est un processus (Patron de tous les autres proc.).

↳ Son rôle est la gestion d'espace de noms du sys. de fichiers, et il a les info de tous les esclaves (nom, facteur de replication...).

(par défaut 3)

↳ Ces info sont stockés sur le disque local sur 2 fichiers:

→ fsimage: Son contenu est chargé dans la RAM au démarrage du Name Node.

- il contient les métadonnées.

→ edit log: - pour enregistrer chaque modif. associée aux métadonnées.

- fichier de journalisation, si on perd les données qui sont dans la RAM on les retrouve ici.

l'accès à la RAM est plus rapide qu'à l'accès au disque.

3) - les blocs de fichiers sont repliqués sur plusieurs nœuds qui sont les Data Node.

- le facteur de replication qui contrôle le nbr de copie de chaque bloc.

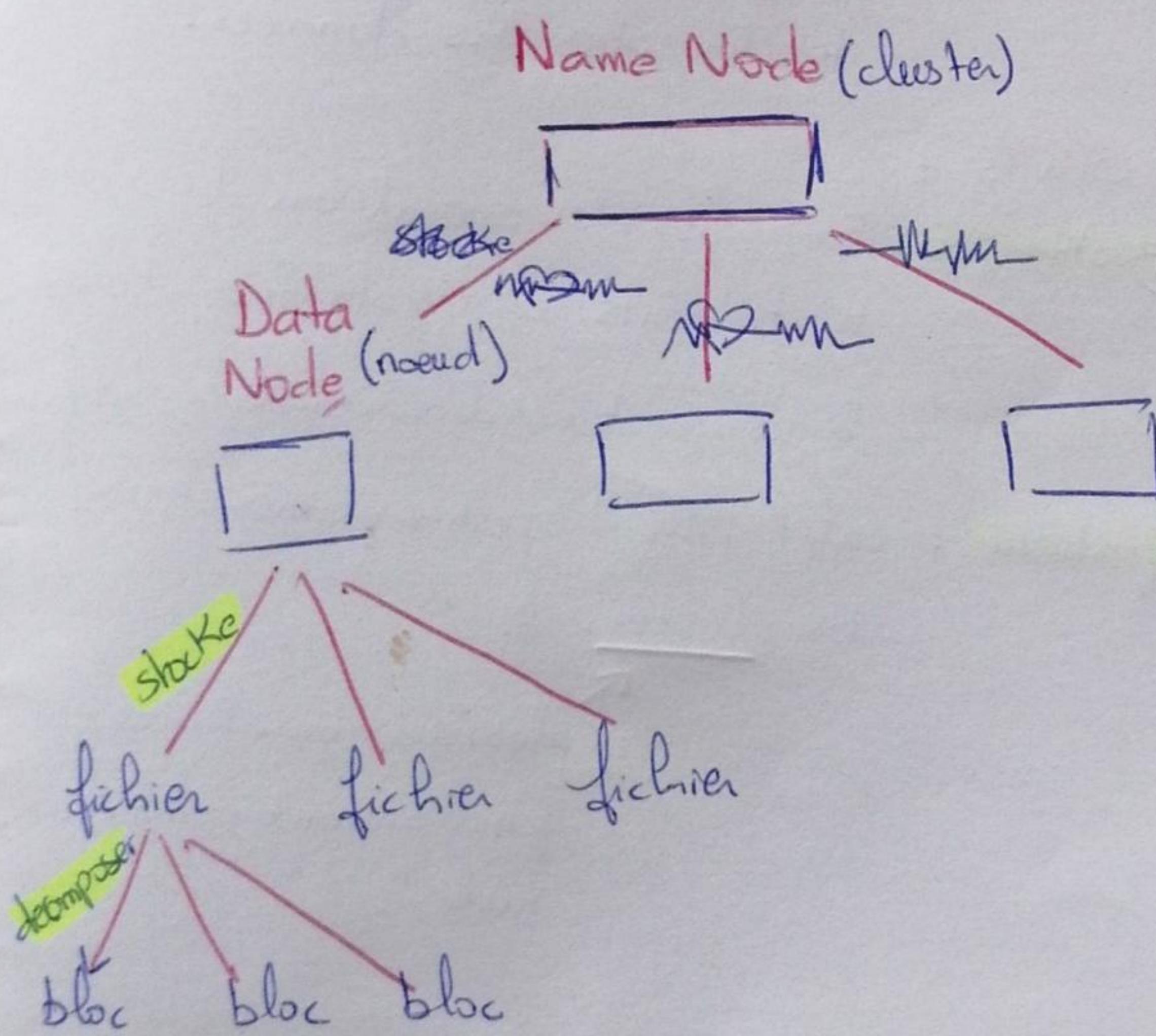
↳ DataNode :

- blocs de fichiers repliqués sur plusieurs nœuds.
- Datanode envoie périodiquement des heartbeats avec un interval (par défaut 3s). (id, espace de stockage, espace utilisé...)
- envoie périodiquement au namenode un rapport sur les blocs qui gère. (chaque 6h).
- Si il tombe en panne, ses blocs seront repliqués sur d'autres Datanodes.

↳ Secondary NameNode :

- il n'a pas le même rôle que le namenode, il fusionne périodiquement l'image de l'espace de nom (fsimage) et le journal de edit log pour éviter qu'il devient très volumineux.
- Pour redemander un nv namenode le temps peut être 30min.
- Pour assurer la haute dispo, on va fonctionner avec le name node (actif & Passif). en cas de panne du actif le passif devient actif.
- Name node actif communique avec les passifs à travers les journal node.

4) Le ZooKeeper qui choisit le namenode passif qui va devenir actif, car tous les namenode A et P maintiennent une session avec le ZooKeeper, alors quand il détecte que l'actif est tombé en panne il revient vers les passifs pour choisir un.



### Lecture d'un fichier hdfs:

- le client communique avec le namenode pour demander la lecture d'un fichier.
- le name node envoie les métadonnées au client.
- le client communique avec le datanode où les blocs sont présent.
- Une fois le client reçoit tous les blocs il les combine pour former un fichier.

- 5) ↳ Ecrire d'un fichier dans hdfs:
- le client appelle la méthode `create` de `DistributedFile` file system pour créer un nouveau fichier.
  - Un objet `fsDataOutputStream` est retourné au client à fin de l'utiliser pour écrire le contenu du file dans hdfs.
  - `DFSOutputStream` créé à fin est à mesure des paquets de (64K), ainsi il décompose les fichiers à écrire en paquets.
  - le proc. de replication commence par la création d'un pipeline à l'aide de `DataNodes`.
  - Le `DataStream` envoie les paquets vers le 1er `DataNode` du pipeline.
  - chaque `DataNode` stocke le paquet reçu et le transmet au `Data node suivant`.

### ↳ Node de fonctionnement:

- \* mode local:
  - install hadoop mais il n'y a pas hdfs.
    - ↓  
sys local.
  - fait des traitements en mode local.
- \* mode pseudo distribué:
  - 1 seul noeud où tournent tous les process
  - facteur de replication est 1
- \* mode totalement distribué:
  - mode d'exécution réel d'hadoop.

6)

## MAP / Reduce

### ↳ Principes:

- le programme doit être écrit selon un modèle bien écrit : mapReduce.

- HDFS est au cœur de mapreduce. il est responsable de la distribution de données à travers le cluster.

### ↳ Intérêt:

- MapReduce permet au développeur de ne s'intéresser qu'à la partie algorithmique.

- Un programme ~~map~~ mapreduce contient 2 fonctions principales Map() et Reduce().

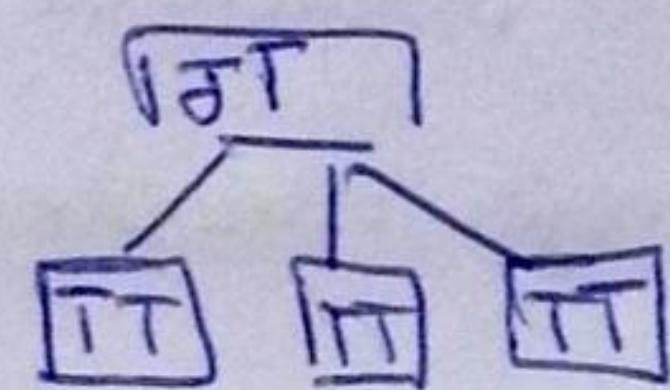
### ↳ Architecture MR 1:

- maître lesclave : le maître est le Jobtracker qui contrôle l'exécution des 2 fonctions sur plusieurs esclaves TaskTracker

### ↳ Jobtracker:

- Reçoit les job envoyés par les clients.
- Communique avec le NameNode pour avoir les emplacements des données à traiter.
- Passer les tâches aux tasktracker.
- Surveille les tâches et status du tasktracker.

- Relance une tâche si elle échoue.
- Algorithme d'ordonnancement des job est par défaut FIFO.
- Le JobTracker fait la gestion des ressources, l'ordonnancement et la surveillance des tâches.



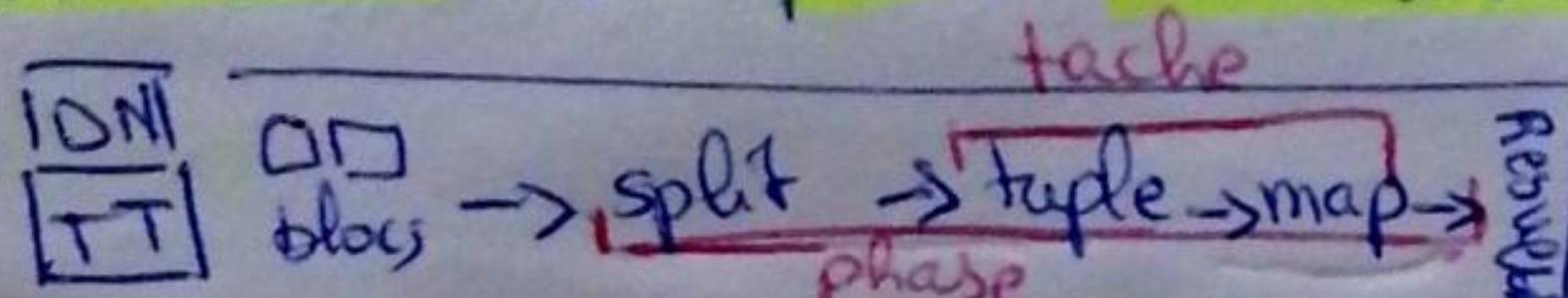
### ↳ Task Trackers:

- exécute les tâches Map et Reduce.
- chaque un possède un nb de slots pour exécuter les tâches map/reduce.
- Communique son statut au Jobtracker via des ~~M2M~~
- Après une durée (mapreduce.jobtracker.expire, trackers.interval) par défaut 10 min le task est considéré perdu.

### ↳ Modèle de programmation "map":

- les fichiers sont divisés en parties logique appelées (Splits).
- Un Split est une division de fichiers qui sont stockés sous forme de blocs hdfs.
- La taille d'un Split est par défaut la taille de bloc.
- chaque split est traité par une tâche map qui applique la fonction map d'une sur chaque tuple.
- les task tracker traitent les splits individuellement et en parallèle.

- Split division logique.  
 - blocs division physique.

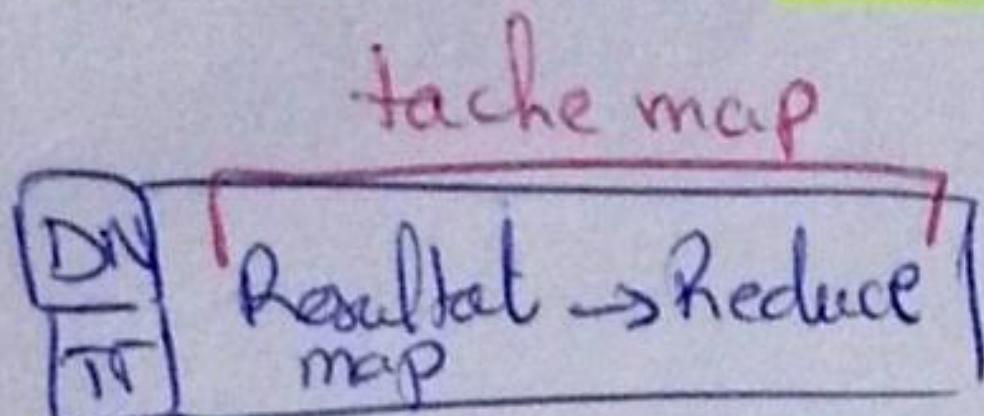


8 - chaque tasktracker stocke le résultat des tâches map dans son sys de fichiers local.

(tuple  $\xrightarrow{\text{clé}}$  valeur)

### ↳ Modèle de programmation "Reduce":

- Une tâche Reduce applique la fct Reduce sur les résultats (clé, valeur) de toutes tâches map.



### ↳ Rappel: Phase de MAP:

- Petit Programme distribué dans le cluster.
- traite une partie de données en entrées. (split).
- Produit des paires <clé, valeur> groupés.

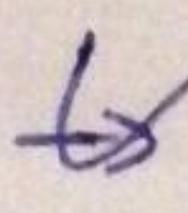
### ↳ Phase de shuffle:

- Le traitement de cette phase est préprogrammé dans mapReduce
- Le résultat <clé, valeur> produit par chaque Map est localement regroupé par clé.
- les paires de même clé sont copiées au même nœud, choisi par le jobtracker qui exécutera la phase Reduce sur ces données.
- Avant l'exécution de la tâche Reduce, les paires de même clé sont fusionnées sur ce nœud pour former un seul pair.

9) ~~SPLIT~~ ~~Map~~ ~~Shuffle & Sort~~ ~~Reduce~~

### ↳ La Phase Reduce:

- Petit programme qui traite toutes les valeurs de la clé dont il est responsable. Ces valeurs sont passées au Reducer sous forme 

<noir, [1,2,3,4]>  <noir, 1>, <noir, 2>, <noir, 3>, <noir, 4>.

### ↳ La Phase Combiner:

- Les résultats de map sont triés et fusionnés (par clé) puis traité par unefct identique à Reduce.
- Ceci permet de minimiser le trafic réseau entre map et Reduce.

<noir, 1>

<noir, 3> → <noir, [1,3]> → <noir, 4>.

### Recap:

- **task map:** exécuté sur chaque split localement.
- " **shuffle:** déplacer les paires ayant la même clé vers le même noeud cible.
- " **Reduce:** calcule les valeurs agrégées pour chaque clé.
- " **Combiner:**

### → YARN: **mapReduce** VS **YARN**

- Job tracker
- task tracker
- Slot

- Resource manager / App master
- Node manager

## • Pig:

↳ une couche d'abstraction au dessus de hadoop.

↳ créer par yahoo.

↳ Constitué de :

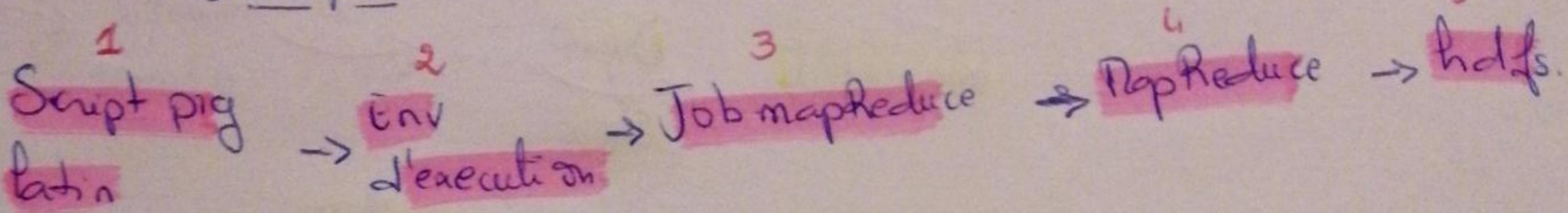
→ Utiliser pour exprimer des flux de données.

→ Environnement d'exécution:

• localement.

• distribué sur un cluster hadoop.

↳ Etapes:



## ↳ Pig Latin:

→ programme constitué d'une série d'opération appliquées aux données d'entrée pour produire des données de sortie.

→ chaque opération donne naissance à un Job mapreduce.

## • Architecture Pig:

↳ Pig Latin script: contient des commandes pig dans un fichier (`.pig`).

↳ Pour l'exécution:

→ soit on l'envoie à un serveur (Pig Server)

→ ou en utilisant un terminal pour lancer (Grunt Shell).

↳ Pour les 2 modes d'exécution on doit passer par:

→ Parsez: - il prend notre texte et l'analyse syntaxiquement

- après il prend toutes les opérations et les organise sous forme d'un graphie acyclique dirigé.

On passe une  
série de frises par  
un noeud.

- un graphie est un ensemble de noeuds connectés.

- Ces noeuds sont les opérations (filtrage, regroupement, classement...)

→ optimiser: - propose le meilleur plan d'exécution qui va réduire le temps d'exécution.  
- après on obtient le plan logique optimisé.

→ Compiler: - prend le plan optimisé et le traduit sous forme de plusieurs job map reduce.

→ exécution engine: - soumet les job au cluster hadoop pour produire le résultat souhaité.

- Ces derniers sont stockés dans un fichier hdfs.

## • Modèle de données:

→ Atom: donnée élémentaire (int, float, char...).

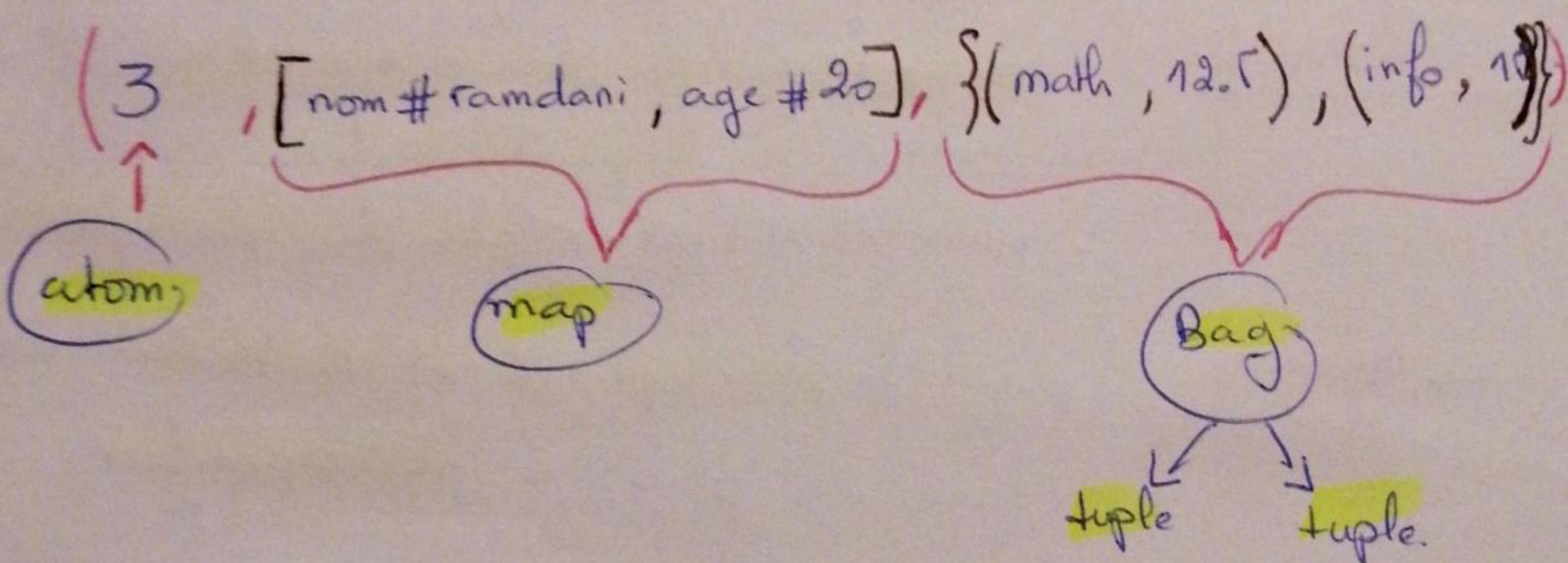
→ tuple: similaire à une ligne dans une table relationnelle  
→ (Rabat, 1, 60)

→ bag: similaire à une table "la modell".  
→ {(Rabat, 1, 60), (Rabat, 3, 40)}.

2) → Map : Série de paire clé - valeur.  
- unique  
- peut être de n'importe quel type.  
type : charactère

- la clé et la valeur sont séparées par #.
- [nom # ali , age # 15].

### exemple tuples:



### Structure pig latin:

- Constitué de 3 étapes :

1 → load: pour charger des données de hdfs en leur donnant un schéma.

→ donner des noms aux différents d. chaque ligne.  
- et aussi préciser leur type.

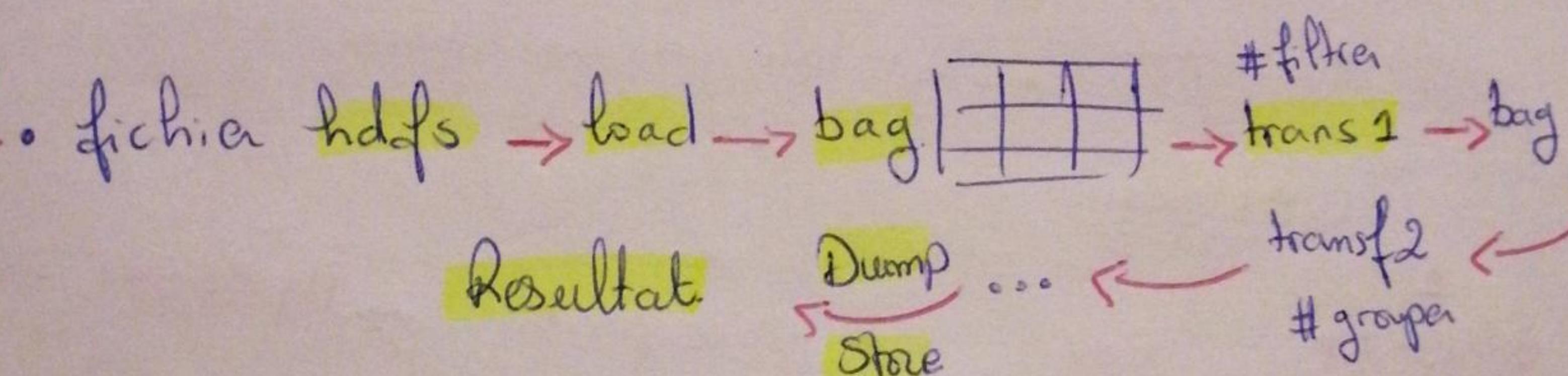
### 2 → Transform / Process:

- opération relationnelles:  
→ filter, select, join, union...

### 3 → Dump / Store:

- afficher le résultat sur l'écran ou le stocker dans un fichier hdfs.
- \* type en pig : - int, long, float, double, chararray, ...
  - " complexe - tuple, Bag, map.

### Load:



- chaque transformation est une opération.

- " " est traduite en job mapreduce.

### Lancement de pig:

Commande :: pig [-x {local | mapreduce}]

- par défaut c'est mapreduce.

désactive optimiser: pig -optimizer-off.

Pour lancer des commandes dans un shell grunt:

grunt > fs - ls lancer l'hadoop (hdfs)

grunt > sh ls (linua).

- 3) Load:  $\text{bag\_resultat} = \text{load} \cdot \text{source}$  [using function] [as schema]
- nom de fichier ou répertoire.
  - nom de fonction de chargement de données.
  - pour donner les noms et les types aux diff membres de bag

example: 1.

$R = \text{load} \cdot \text{'leser / loadpig...'} \text{ Using pigstorage(';) As}$   
 $(\text{year: chararray}, \text{temperature: int}).$

nom ↑ type

Accès au membre:  $R.\$0 = R.\text{year}.$

example: 2.

$\boxed{(\text{chaine}, \text{barme}) | (11, 12)}$

bag qui contient 2 tuples. type par défaut bytarray

$R = \text{load} \cdot \dots \text{ Using pigstorage('l') As (infos: tuple (nom, pren))}$   
notes: tuple notes (n1:int, n2:int));

Affichage:  $\text{dump } R;$

Stockage:  $\text{store } R \text{ into } \dots \text{ using PigStorage();}$

## traitement de donnée:

### • filter:

Dest = filter source by expression;

exp:  
 $F = \text{filter } R \text{ by } x > 12;$

• matches : equals ou like. (=).

### • trier: Dest = order source by champ [asc / desc];

exp:  
 $D = \text{order } R \text{ by year Desc};$

### • grouper: Dest = group source by champ;

exp:  
 $G = \text{group } R \text{ by categorie};$

### • pour créer un tuple:

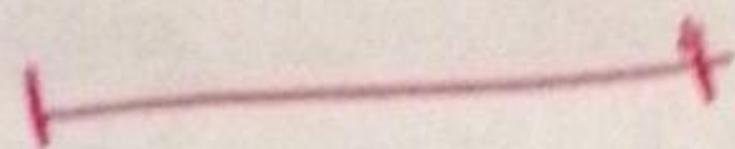
dest = foreach source generate expression;  
tuple ↴ map ↴ bag ↴ function

⚠ le résultat d'un groupement  
d'un bag de 2 champs

$G \{ \text{group: int, R: } \{ \text{year, temp, act} \} \}$

4) Pig -x local : pour lancer pig en mode local.

- Describe est utilisé pour afficher le schéma d'une relation.
- Dump affiche les résultats à l'écran.
- hdft dfs -get ...
- cp ...
- sh pwd : mode local pour savoir l'emplacement.



### Fonctions de calcul:

$x = \text{foreach} \dots \text{generate} \dots \left( \begin{array}{l} \text{AVG} \\ \text{Count} \\ \text{Max} \\ \text{Min} \end{array} \right) (\dots);$

### Imbrication d'opérations:

$\{ \dots \}$   $R = \text{foreach} \dots \text{Generate } C;$   
 $x = \text{Distinct } R;$   
 $\} ;$   $\text{Generate } x;$   
 $\} ;$

Join Permet de faire des jointures entre 2 bag ou plus.

$\text{Bag}_x = \text{Join } \text{Bag}_1 \text{ by champ}_1, \text{Bag}_2 \text{ by champ}_2 \dots;$   
 $x = \text{Join } R_1 \text{ by categorie}, R_2 \text{ by categorie};$

$R_3 = \text{Join } R_2 \text{ by category} \left( \begin{matrix} \text{left} \\ \text{right} \\ \text{full} \\ \text{outer} \end{matrix} \right) \text{ R}_2 \text{ by category}$

Permet de faire le produit cartésien en 2 bags.

bag<sub>3</sub> = Cross bag<sub>1</sub>, bag<sub>2</sub>;

## Opérateurs arithmétique et autres.

- Opérateurs arithmétiques: +, -, \*, /, %.
- .. de Cast : (type.) champ
- .. diviseur : (Condition ? true : false)
- .. de sélection multiple.

### ↳ Case

when Condition<sub>1</sub> then Resultat<sub>1</sub>

.. .. .. .. .. ..

Else résultat

End

↳ Case when val<sub>1</sub> then resultat<sub>1</sub>

.. .. .. .. .. ..

else résultat

End.

## 5) La fonction flatten

- Permet de mettre en plate un champ qui est d'un type complexe (bag, tuple).

exemple:

R: {a:int, t:(b:int, c:int)}  $\Rightarrow$  (2000, (1,2))  
 (2000, (3,4))  
 (2001, (5,6))  
 (2001, (7,8))

F = foreach R generate a, flatten(t);

↓

(2000, 1, 2)  
 (2000, 3, 4)  
 (2001, 5, 6)  
 (2001, 7, 8)

R : {a:int, b:{c:int, d:int}}}

(2000, {(6,7), (4,8)})  
 (2020, {(3,5), (2,9)})

F = foreach R generate a, flatten(b);

↓

(2000, 6, 7)

(2000, 4, 8)

(2020, 3, 5)

(2020, 2, 9)

Le nombre de champ  
dans le bag est  
le nombre de  
ligne qu'on va  
obtenir après flatten

### Tokenize:

Divise une chaîne de caractère en mot et chaque mot sera dans son propre tuple et tous les tuples seront dans un bag.

f = f.read R génère tokenize( ) ;

### Text loader:

transforme chaque ligne charger par un à un tuple.

R = load --> Using Text loader( ) AS (L: chararray);

### Size:

retourne la taille selon le type passé en paramètre

size (map-) => taille de de valeur.

" (tuple) => nbr de champs dans le tuple

" (bag) => " tuples " un bag

### Pour exécuter un script pig:

exec / run.

- pour passer un paramètre

(exec / run) - param param-name = param value ...