Aperçu du modèle d'exécution Map-Reduce et Spark - Optimisation logique SQL

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi
mohamed-amine.baazizi@lip6.fr

2020-2021

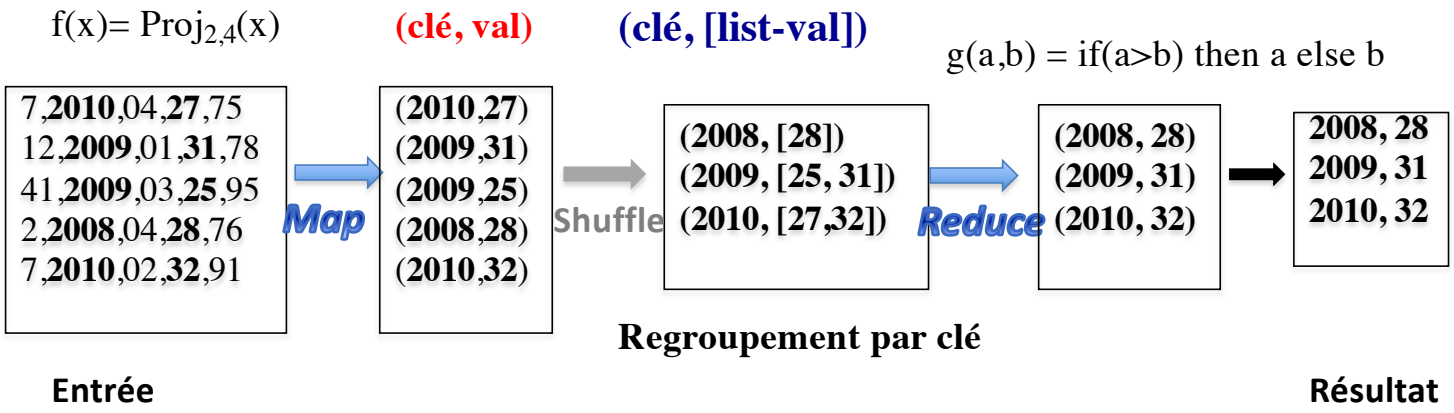
Plan

- Modèle d'exécution de Map-Reduce
- Modèle d'exécution de Spark
  - Algèbre RDD
  - Algèbre Dataset

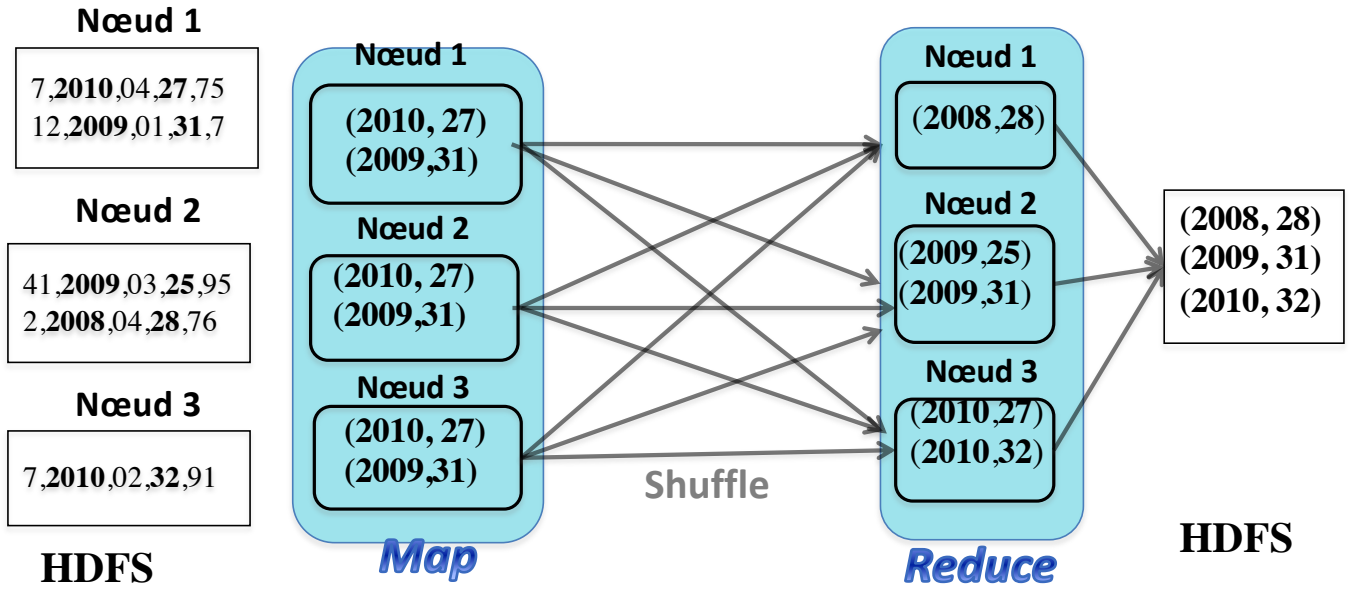
...le but est de comprendre l'exécution pour savoir optimiser (les 3 séances suivantes)

Map Reduce : rappel par un exemple

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

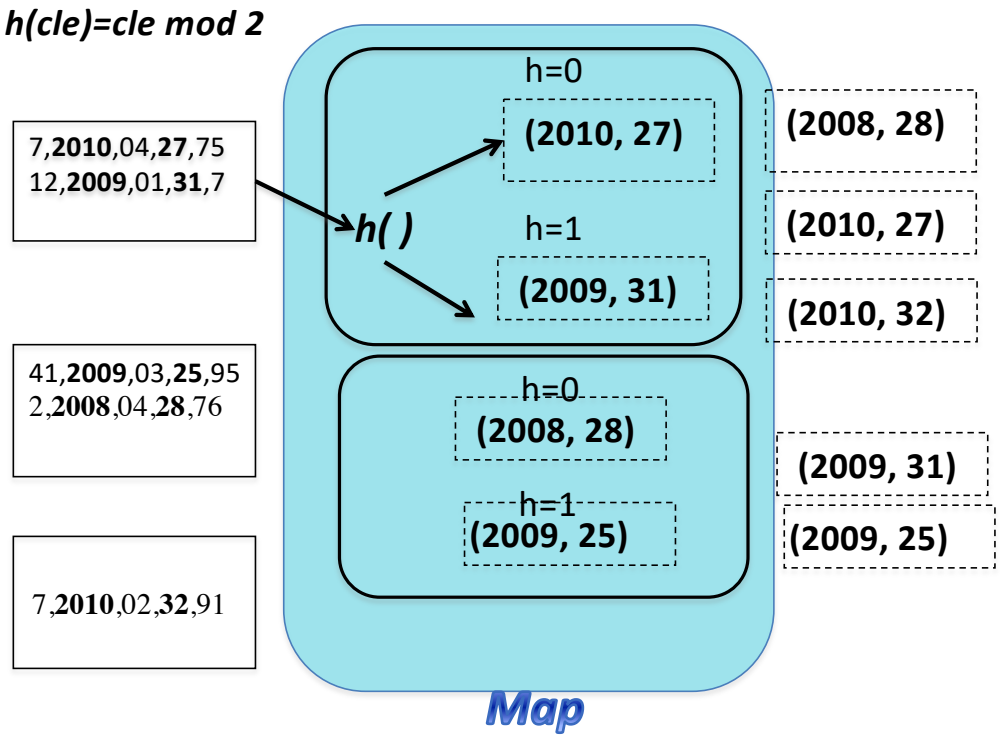


Exécution **Hadoop** Map Reduce

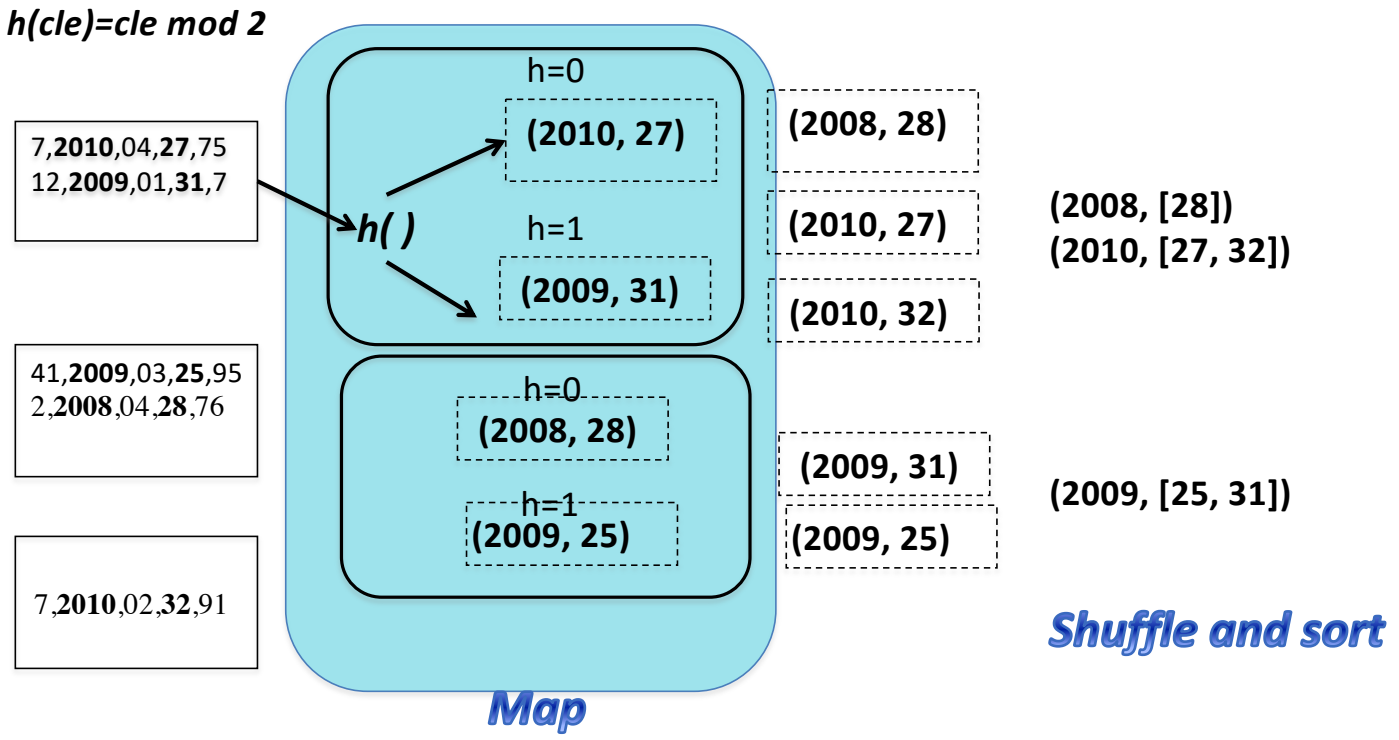


Entrée : n-uplets (station, **annee**, mois, **temp**, dept)
Résultat : select annee, Max(temp) group by annee

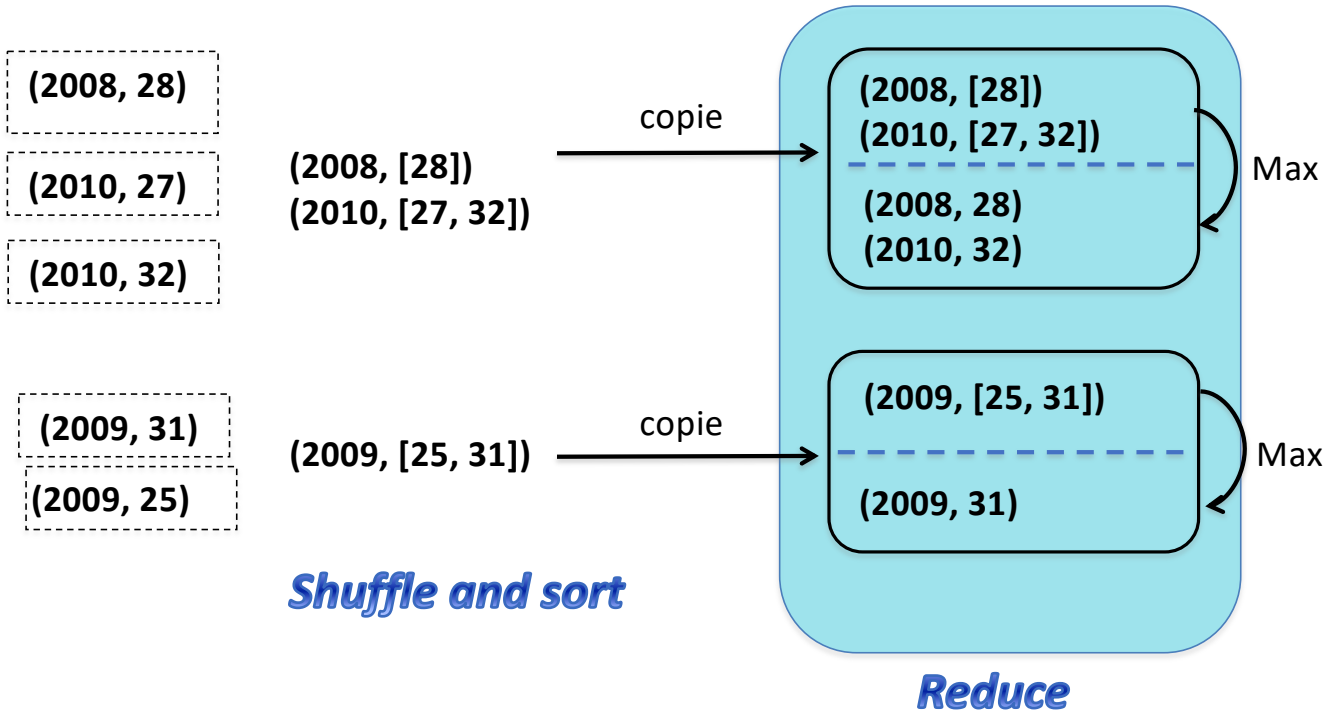
Phase Map


h(cle)=cle mod 2
7,2010,04,27,75
12,2009,01,31,7
41,2009,03,25,95
2,2008,04,28,76
7,2010,02,32,91
h( )
h=0
(2010, 27)
h=1
(2009, 31)
h=0
(2008, 28)
h=1
(2009, 25)
(2008, 28)
(2010, 27)
(2010, 32)
(2009, 31)
(2009, 25)
Map

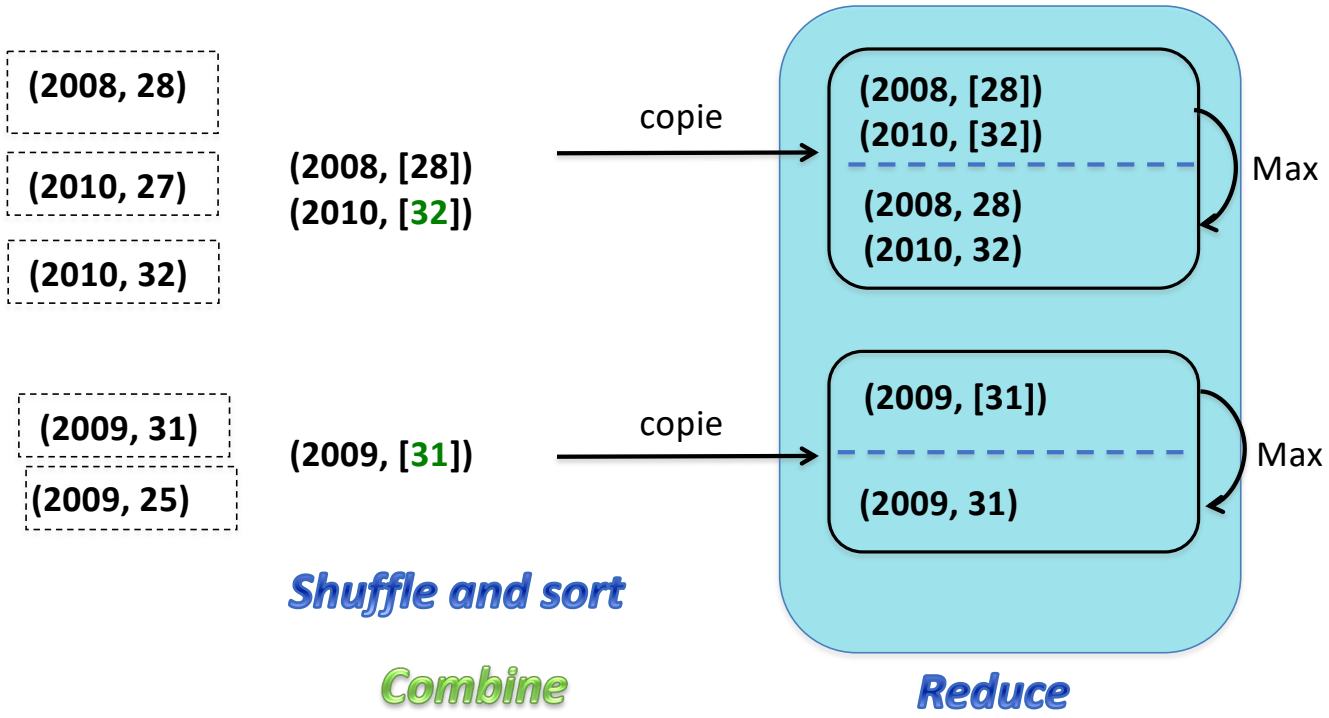
Phase Shuffle & Sort



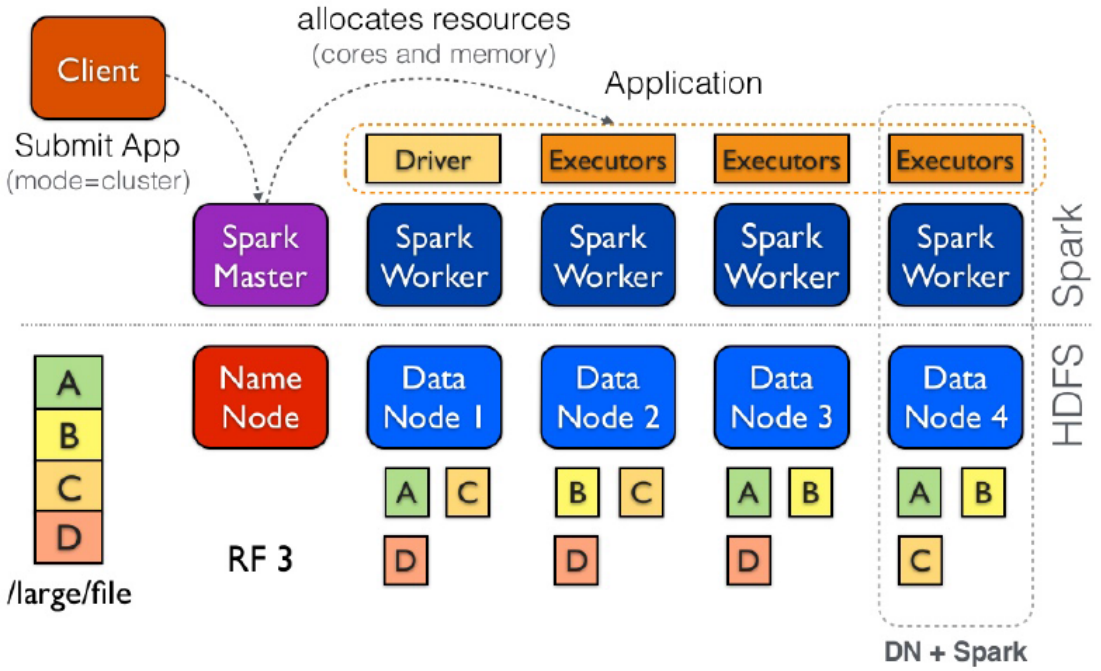
Phase Reduce



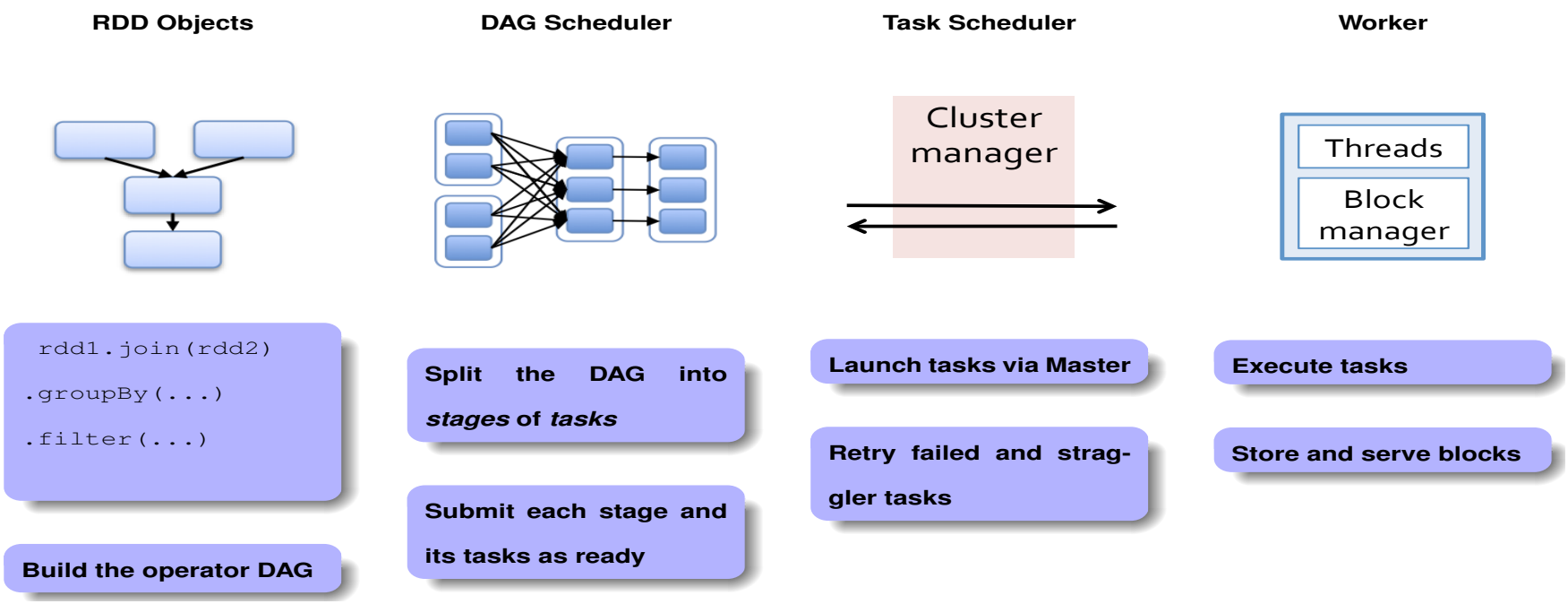
Combine


(2008, 28)
(2010, 27)
(2010, 32)
(2009, 31)
(2009, 25)
(2008, [28])
(2010, [32])
(2009, [31])
copie
copie
(2008, [28])
(2010, [32])
(2008, 28)
(2010, 32)
(2009, [31])
(2009, 31)
Max
Max
Shuffle and sort
Combine
Reduce

Architecture Spark

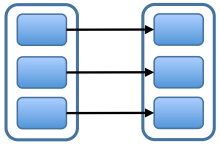


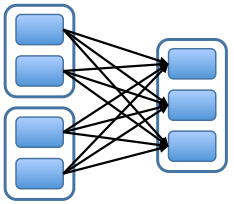
Cycle de vie d'un programme Spark



Décomposition d'un programme

- Opération
  - Transformation : crée une nouvelle RDD à partir d'autre(s) RDD
    - locale : ex. map, filter
    - distribuée : ex. join, reduceByKey
  - Action : évalue la RDD en exécutant la chaine de transformation
    - retourne type de base ou *User Defined Type*



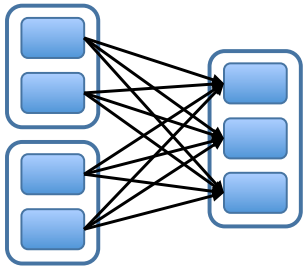


- Stage
  - Séquence de transformations locales
  - terminée par une transformation distribuée ou par une action

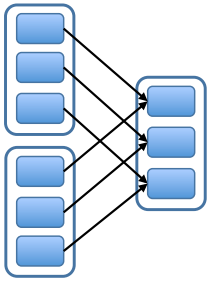
- Plan
  - Séquence de *stages* terminée par une action

Optimisation

- Tirer profit du partitionnement effectué par transformation précédentes
  - jointure sur une clé pour laquelle les deux relations sont déjà partitionnées

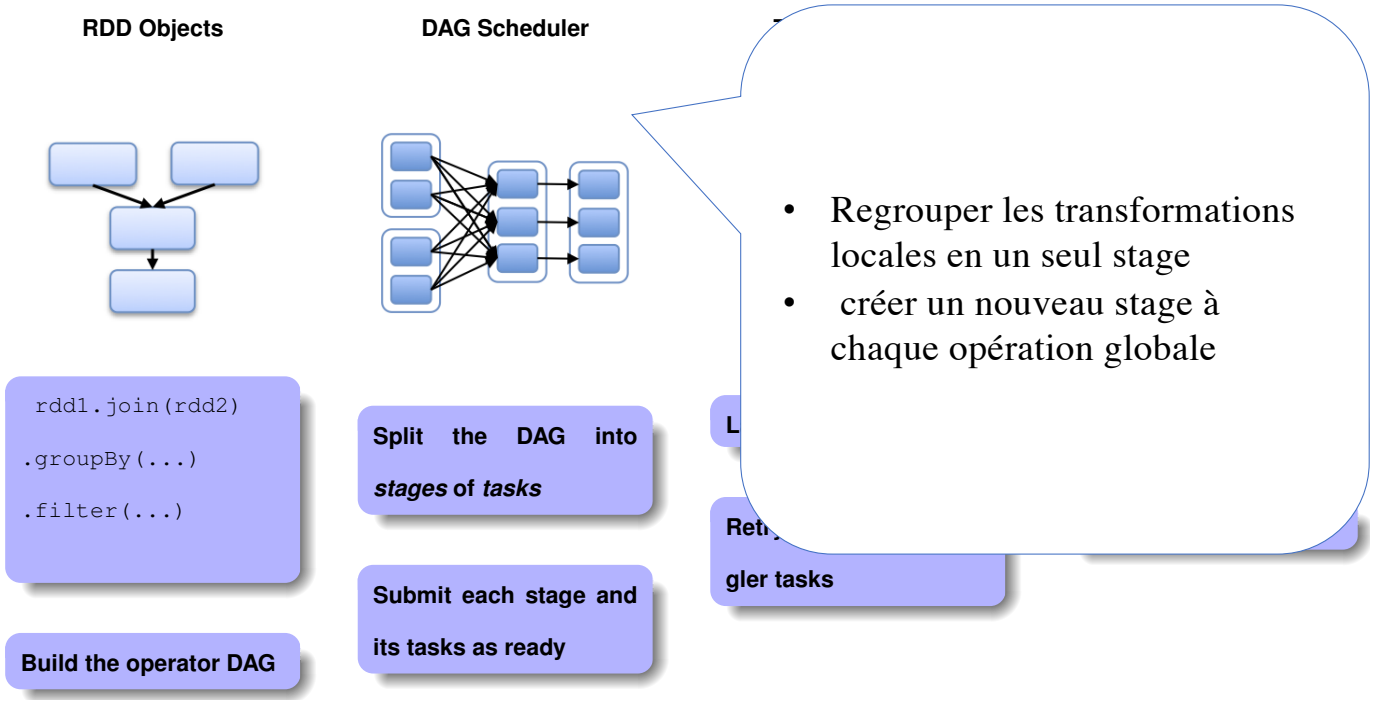


join with inputs not co-partitioned



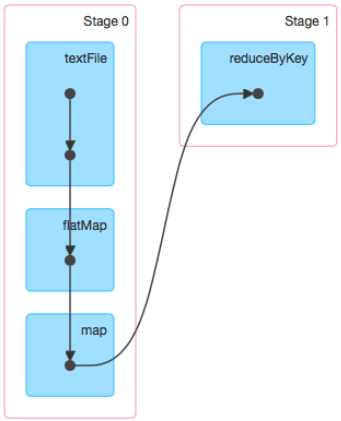
join with inputs co-partitioned

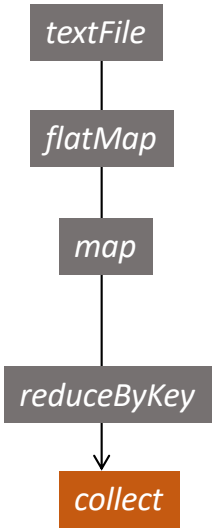
Génération du plan d'exécution



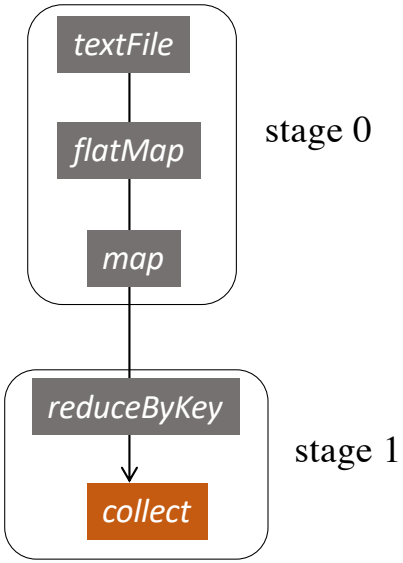
Exemple : wordcount

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_+_)
val results = counts.collect
```



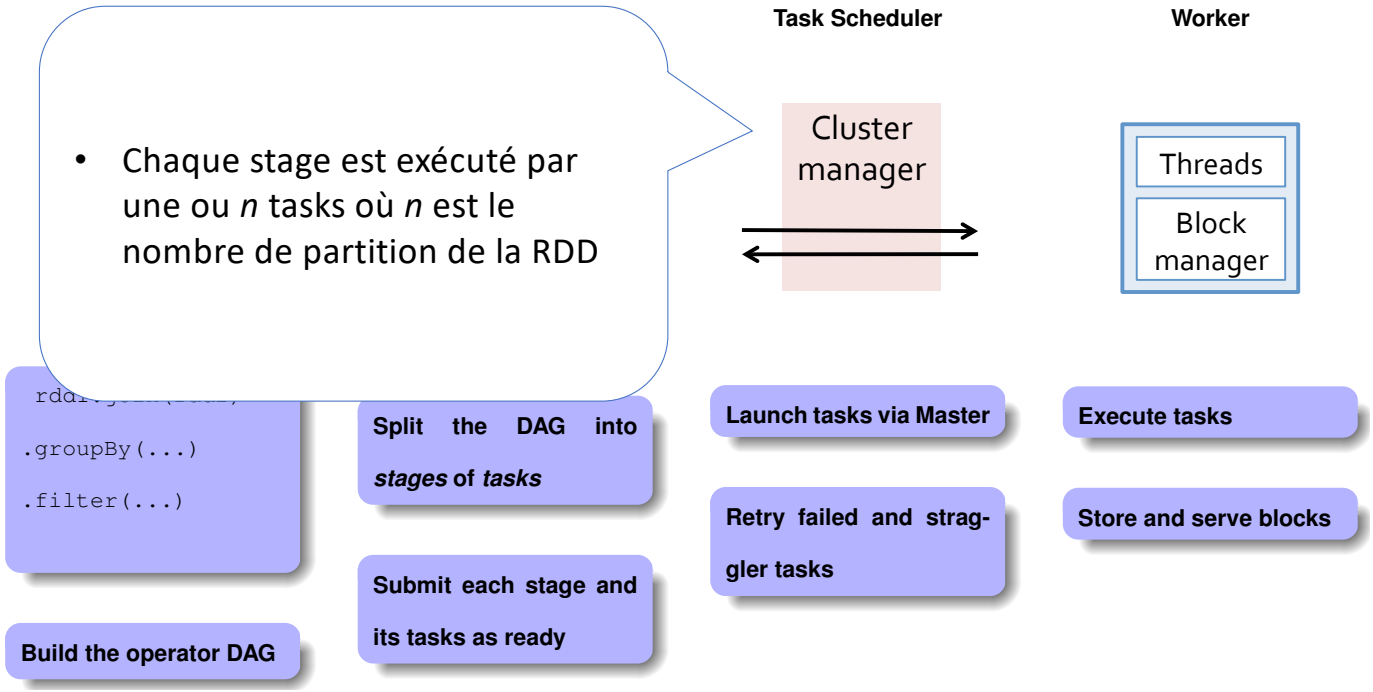


DAG d'opérateurs



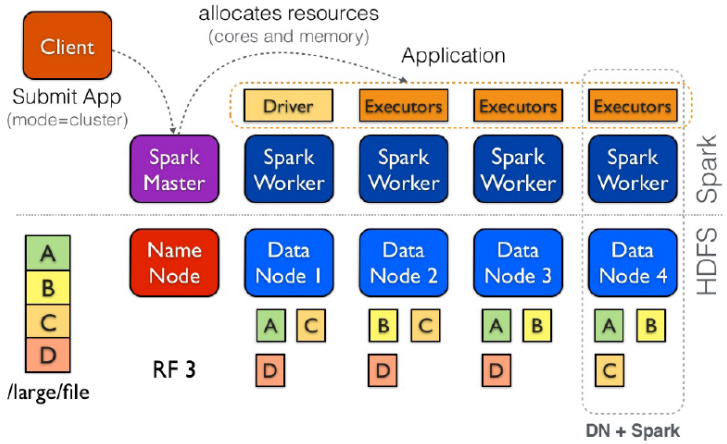
DAG de Stages

Exécution du plan



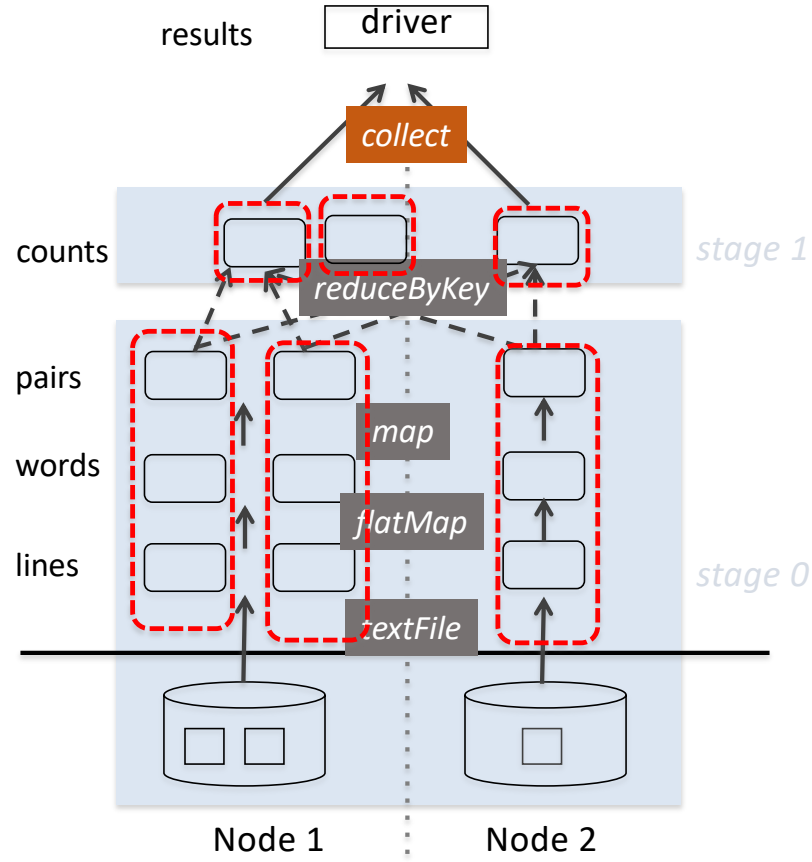
Exécution du plan

- Affectation de *tasks* aux stages en privilégiant la localité des données
  - exécuter une tâche dans le nœud de la partition
- Matérialisation des résultats produits pour chaque stage
  - en cas de panne, par exemple perte de certaines partitions, ne recalculer que celles-ci



Exécution du plan Wordcount

```
val lines = sc.textFile(filename)
val words = lines.flatMap(x=>x.split(" "))
val pairs = words.map(x=>(x,1))
val counts = pairs.reduceByKey(_+_)
val results = counts.collectAsMap
```



partition d'une RDD

task

action

transformation

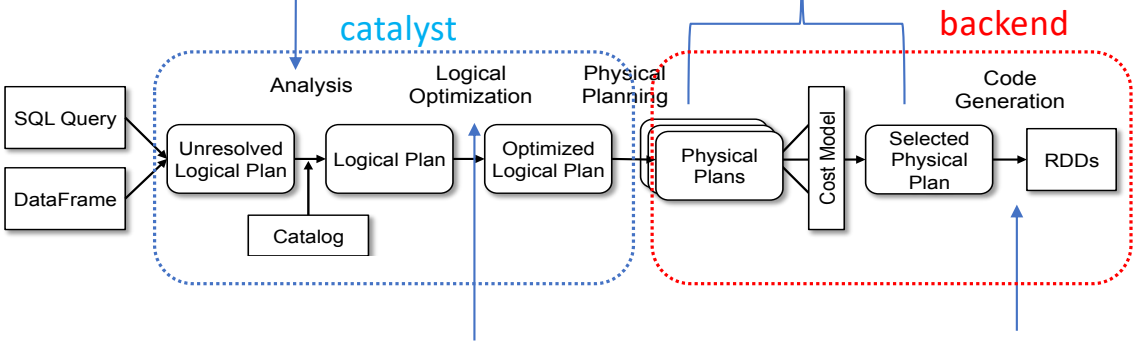
Bilan

- RDD comme couche physique
  - Optimisation minimale
    - Regrouper opérations en stage
    - Exploiter partitionnement existant
  - Absence d'optimisation logique
- Autres pistes d'optimisation
  - Persistence des données fréquemment lues
    - Responsabilité du programmeur

Evaluation de SQL sur Spark

(1) Vérifier l'existence des attributs, extraire leurs types

(3) Générer plusieurs plans physiques et choisir l'optimal en terme de cout



(2) Appliquer les équivalences algébriques, éliminer sous-requetes, simplifier les expressions

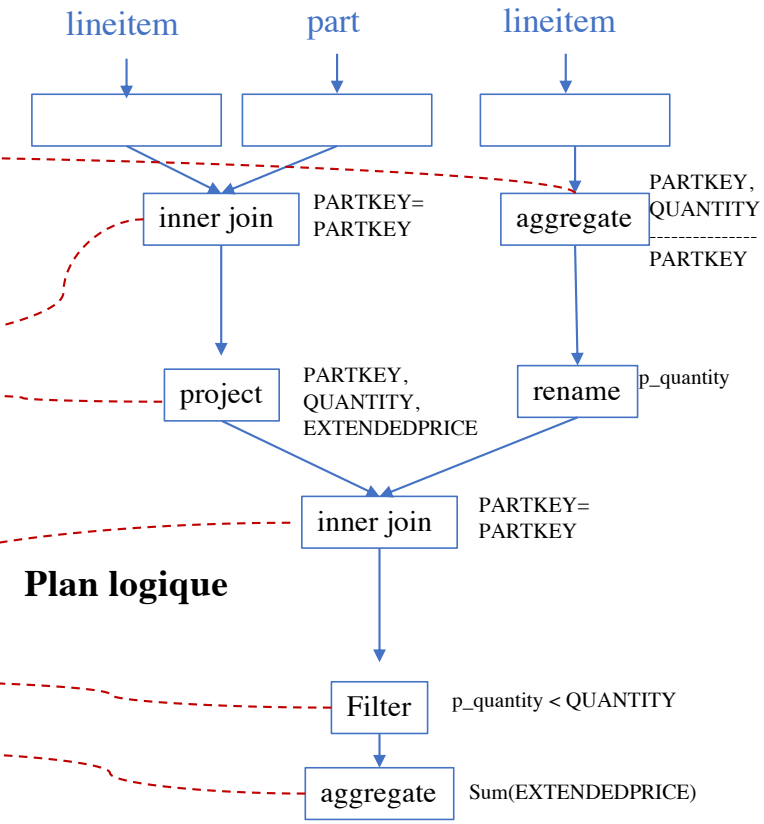
(4) Générer un code Spark optimisé

(1) Génération du plan logique

- Plan logique = arbre d'opérateurs logique
- Analyse statique
  - résolution des noms d'attributs en utilisant le catalogue
  - Vérification du référencement des attributs
- Traduction SQL vers algèbre interne
  - Opérations arithmétiques : +, -, …
  - Fonctions d'agrégations : sum, avg, …
  - Algèbre interne (DSL) :
    - Project, Filter, Limit, Join, Union, Sort, Aggregate, UDFs, …

Illustration

```
val lineitem = spark.read. ... .load(lineitem_t)
val part = spark.read...load(part_t)

val inner = lineitem.groupBy("PARTKEY")
                    avg("QUANTITY").

            rename("avg(QUANTITY)","p_quantity")

val outer = lineitem.join(part, "PARTKEY")
                    .select("PARTKEY",
                            "QUANTITY",

            "EXTENDEDPRICE")


val q17 = inner.join(outer, "PARTKEY")

            .where("p_quantity<QUANTITY")

            .agg(sum($"EXTENDEDPRICE")/7)
```



Plan logique

Tpch Q17 simplifiée

(2) Optimisation du plan logique

- Catalyst : réécriture de l'arbre d'opérateurs
  - Spark 2.4 : plus de 100 règles regroupées par lots (batch)
  - Quasiquotes Scala, pas de documentation précise (analyse code)
  - Déclenchement : une seule fois, point fixe (nb itérations 100)
  - Quelques règles utiles
    - Elimination des sous-requêtes
    - *ColumnPruning* : Elimination des attributs inutiles
    - *CollapseProject* : Combinaison des projections
    - *PushDownPredicate* et *PushPredicateThroughJoin* : évaluation des filtres le plus en amont possible et/ou en concomitance des jointures
    - *InferFiltersFromConstraints* : rajouter des filtres en fonction de la sémantique des opérateurs
    - Elimination des distincts et des sorts
    - expansion des constantes, réécriture des filtres

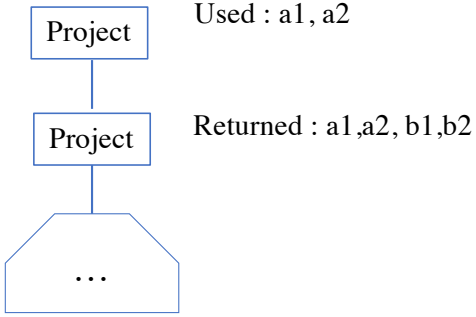
Signatures de opérateurs algébriques

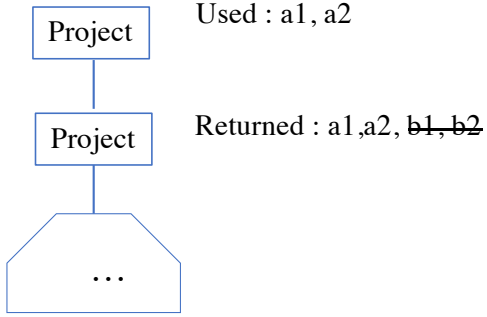
- Structure récursive
  - Op(arg_1, …, arg_n, child), où child est un arbre désignant un plan logique
  - op.used désignent la liste des attributs utilisés
  - op.returned désignent la liste des attributs retournés
  - op.arg_i pour accéder à l'argument (comme pour un objet)
- Quelques exemples
  - Filter(cond, child) : cond est la condition à vérifier
  - Project(projList, child) : projList est la liste d'attributs à projeter
  - Aggregate(grpExp, aggExp, child)
    - grpExp : attributs de partitionnement
    - aggExp : fonctions d'agrégation
  - Join(left, right, joinType, condition)
    - left et right: plans
    - joinType : inner, outer, full, cross, …

Lot *ColumnPruning*
Règle d'élimination des attributs inutiles

cas op = Project(_, p_2 : Project)

- si p2.returned ⊄ op.used /*certains attributs de p2 inutiles pour op*/
- alors p2.projList := p2.projList ∩ op.user /*les éliminer*/

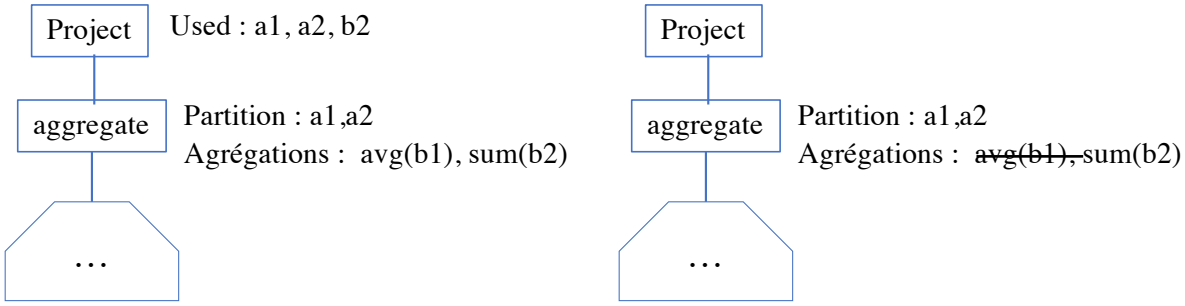




Lot *ColumnPruning*
Règle d'éliminations des agrégations inutiles

cas op = Project(_, a: Aggregate)

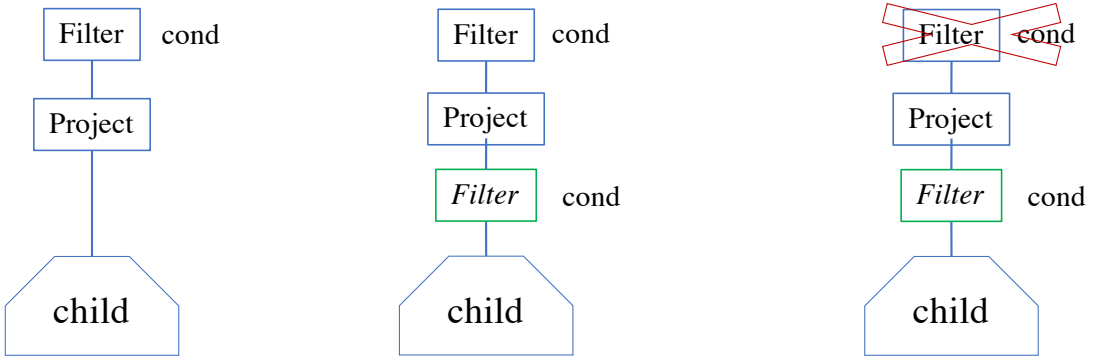
- si a.returned $\not\subset$ op.used /*certains attributs de a inutiles pour op*/
- alors a.AggExp = a.AggExp.filter(op.used) /*n'effectuer que les agrégations sur les attributs utiles pour op*/



Règle *PushDownPredicate*

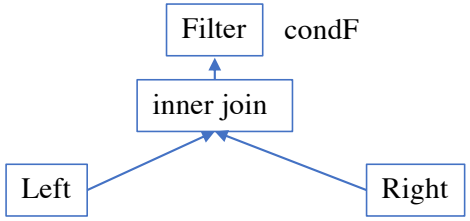
cas op=Filter(cond, Project(_, child))

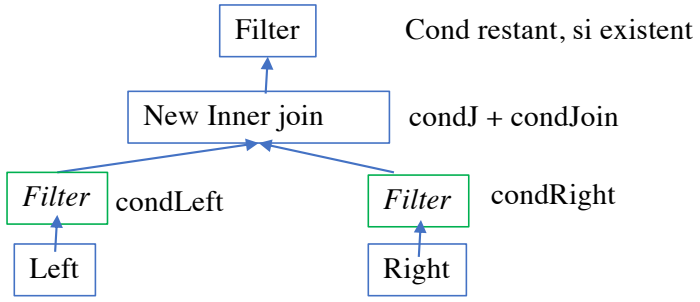
- si cond est déterministe et peut être poussée
- alors Filter(cond, Project(_, Filter(cond, child)))
- Filter le plus externe sera éliminé après des vérifications



Règle *PushPredicateThroughJoin*

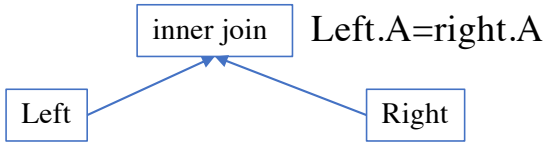
- Diviser la condition en 2 sous-conditions qui pourraient être évaluées dans une des branches de la jointure et une évaluée avec la jointure
- Filter(condF, Join(left, right, joinType, CondJ)
  - Diviser condF en leftCond, rightCond et joinCond
  - cas joinType = Inner
    - newLeft := left où child = Filter(condLeft, child)
    - newRight idem
    - newCondJ := CondJ + joinCond

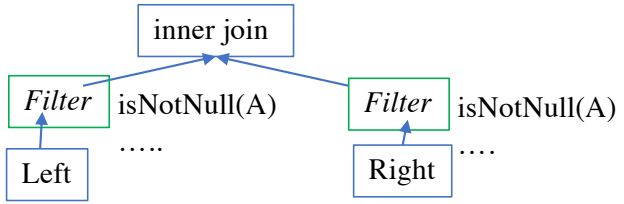




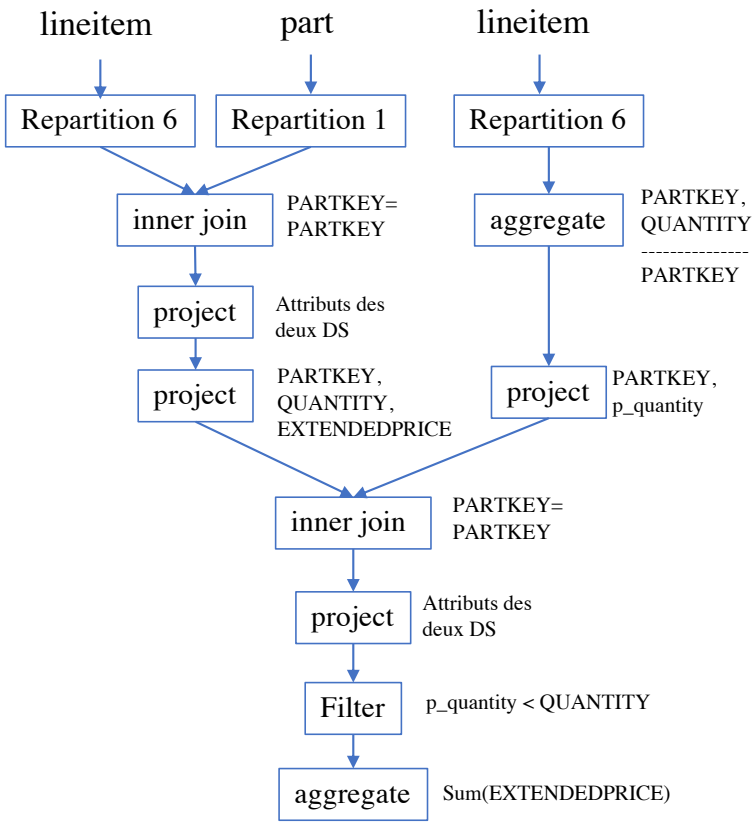
Règle *InferFiltersFromConstraints*

- Conditions qui s'ajoutent à des filtres existants ou à des jointures en function des contraintes sémantiques des opérateurs
  - Ex. l'attribut de jointure ne doit pas être *Null*
- Comme pour *PushPredicateThroughJoin*. diviser la condition en sous-conditions à propager
- Join(left, right, type, CondJ)
  - cas joinType = Inner
    - Extraire toutes les contraintes pour left et right et en extraire les filtres
    - Rajouter le filtre isNotNull sur les attributs de jointure

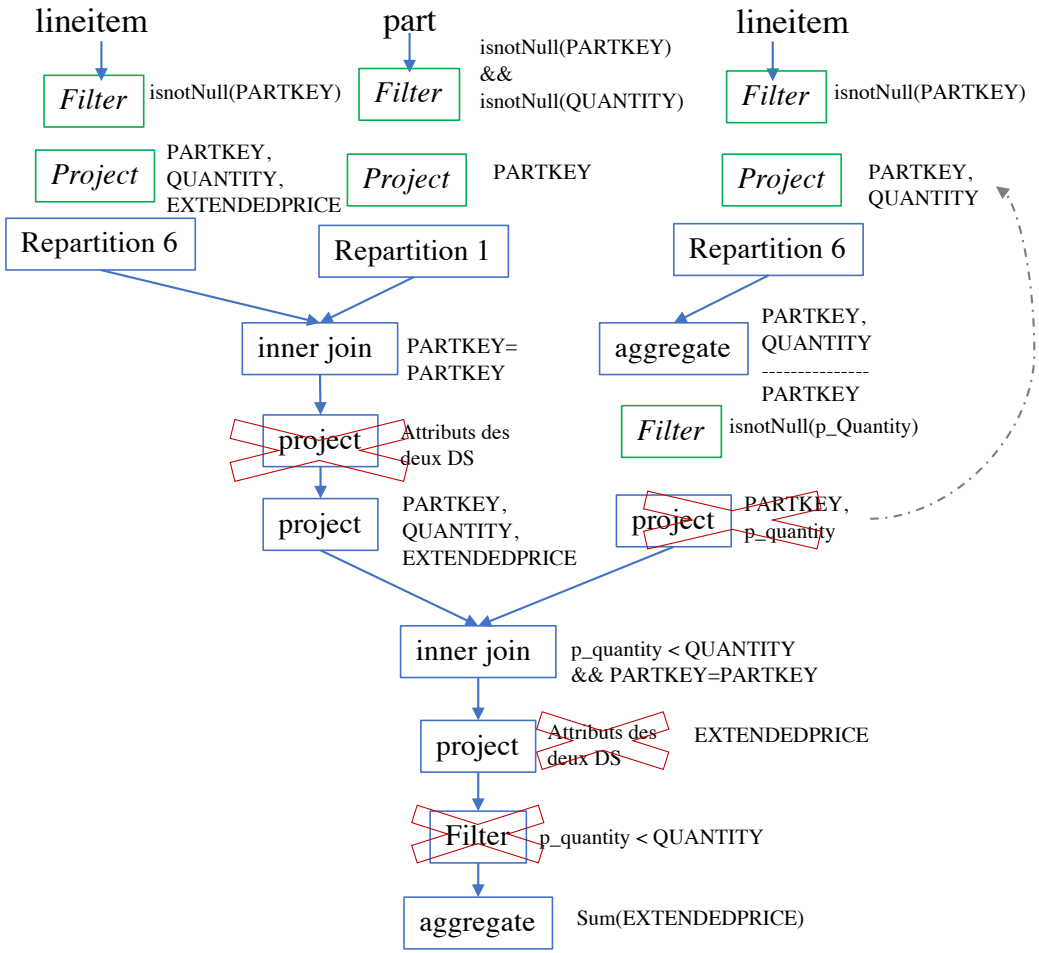




Plan logique Q17



Plan logique Q17 optimisé

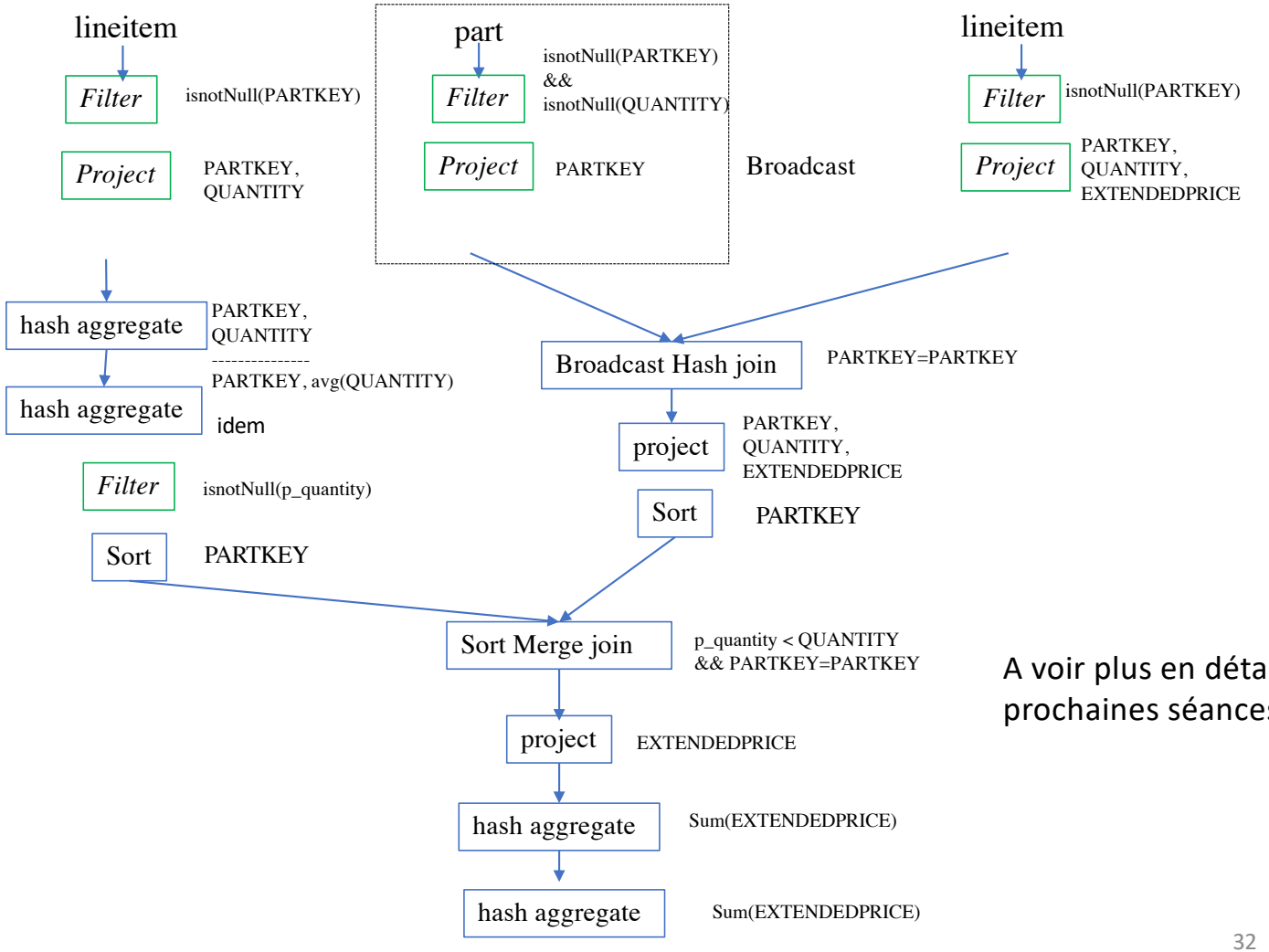


Attention sens de lecture inversé

(3) Génération du plan physique

- Phase 1 : Transformer le plan logique en un plan physique
- Phase 2 : appliquer règles d'optimisation
- Stade préliminaire de développement dans Spark 2.4
  - Pipelining d'opération (filter et project)
  - Choix entre jointure par hachage et diffusion en fonction de la taille des données
  - Utilisation du cout pour réordonner les jointures?
- Optimisation adaptative en Spark 3.0 (à découvrir)

Plan physique Q17



A voir plus en détail les prochaines séances

32

Autres pistes d'optimization

- L'organisation physique des données compte!
  - Format de stockage colonnes : ORC, parquet
  - Partitionnement sur disque
  - persistance de données accédés en répétition (ex. jeux d'entrainement)