Requêtes SQL en Map-Reduce et en Spark

Master DAC – Bases de Données Large Echelle

Mohamed-Amine Baazizi
baazizi@ia.lip6.fr

Octobre-Novembre 2018

Implantations de SQL en distribué

- Native MapReduce
  - Hive 
  - Facebook Presto
  - MapR Drill, LinkedIn Tajo
- Hadoop comme data store
  - Impala 
  - Hadapt 
  - Pivotal HAWQ 
- Systèmes dédiés
  - Spark SQL 
  - FlinkSQL 
  - IBM BigSQL 

Remarques :

- Requêtes analytiques seulement (pas de transactions)
- Souvent sous-langages du standard SQL

Implantations de SQL en distribué : principes

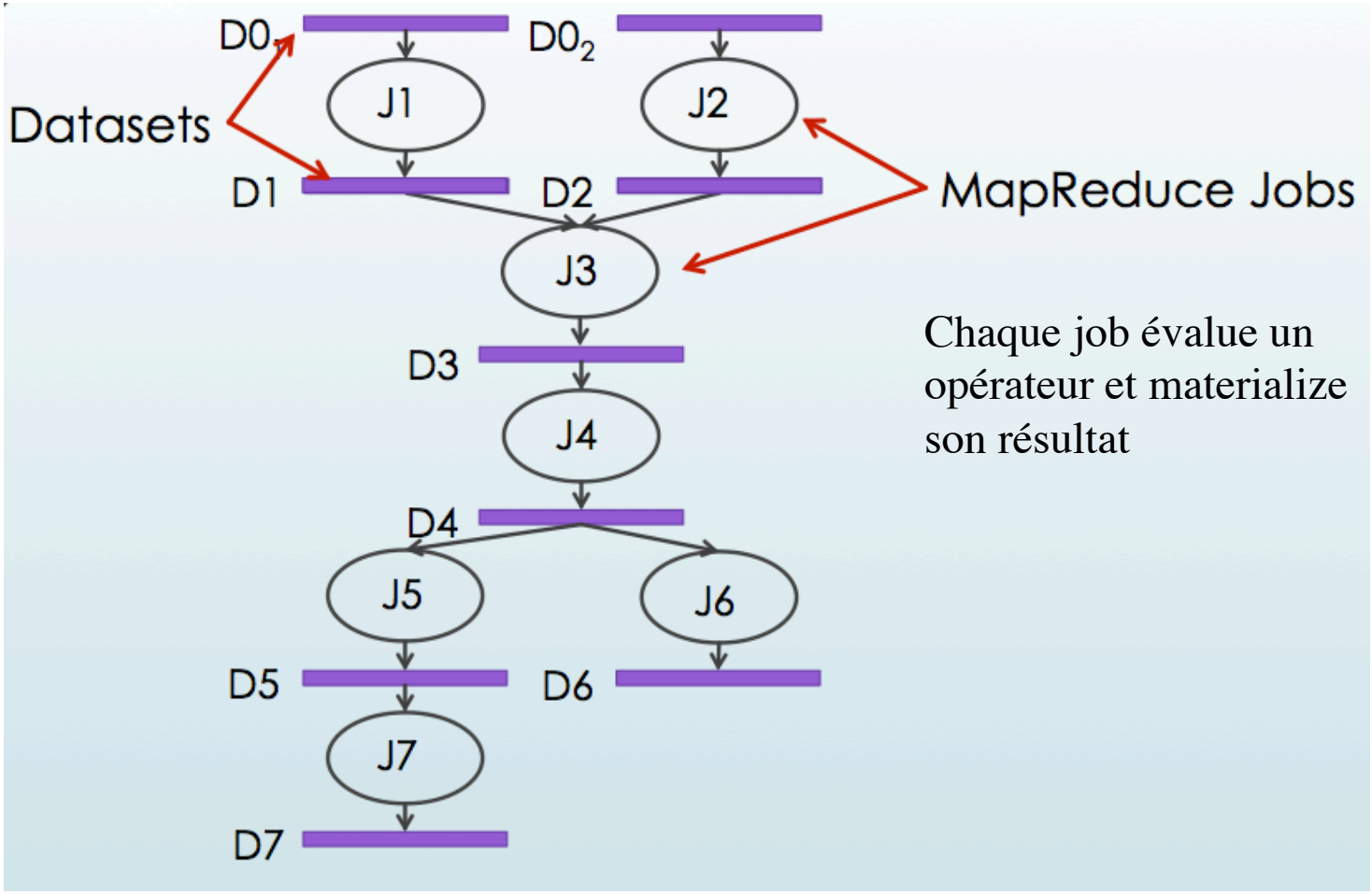
- Native MapReduce 

  - Génération d'un DAG Map-Reduce, optimisations
- Hadoop comme data store  

  - Utilisation de SGBD relationnels (Postgres, mysql)
- Systèmes dédiés 

  - Génération d'un worklow avec algèbre propre

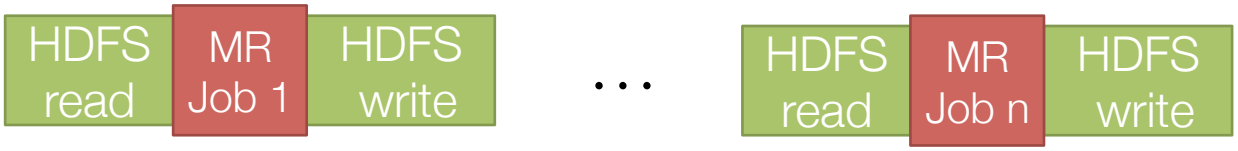
Implantation Native MapReduce



Chaque job évalue un opérateur et materialize son résultat

Implantation Native MapReduce

- Avantages (ceux de MapReduce)
  - Passage à l'échelle, reprise sur panne
- Inconvénients (ceux de MapReduce)
  - Coût de la matérialisation entre chaque job



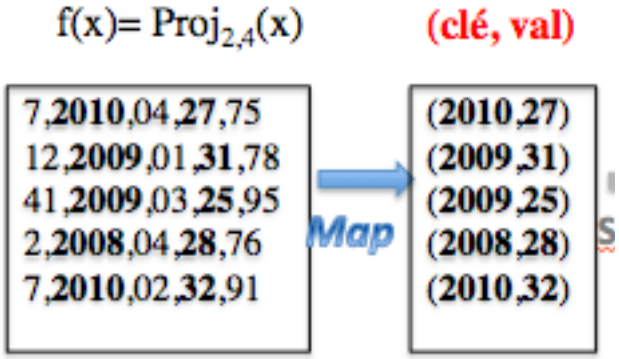
- Pistes pour l'optimisation
  - Réduire le nombre de jobs en combinant les opérateurs
  - Gestion efficace des données intermédiaires

Traduction SQL vers MapReduce

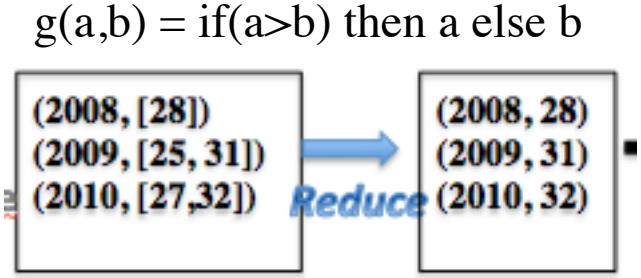
- Opérateurs considérés
  - Unaires : projection, sélection, agrégations
  - Binaires : produit cartésien, jointure, union, intersection, différence
  - Fonctions d'agrégations de SQL : min, max, …

Traduction opérateurs unaires

- Projection, sélection
  - Triviale : UDF du *map*

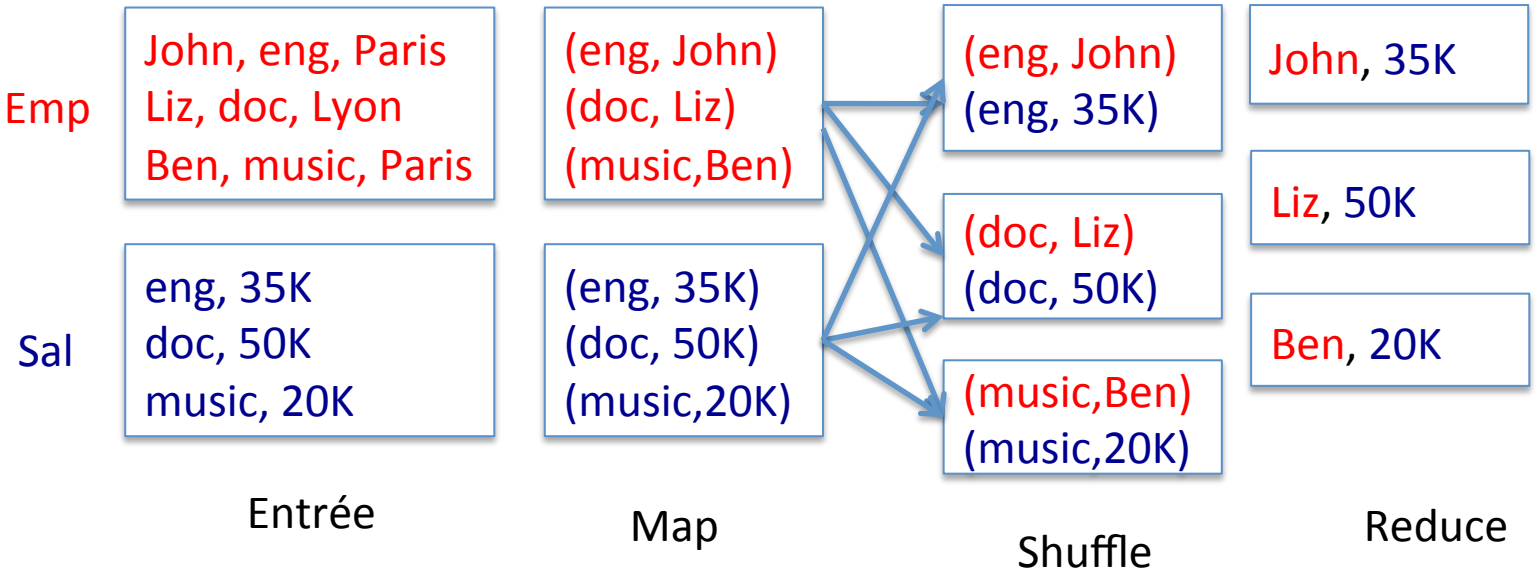


- Agrégation
  - Si données partitionnées, appliquer fonction d'agg. durant *reduce*



Traduction de la jointure

- Jointure par partitionnement
  - *Map :*
    - étiqueter les tuples de chaque relation
    - exposer attribut(s) de jointure comme clé
  - *Reduce :* combiner les valeurs de la même clé

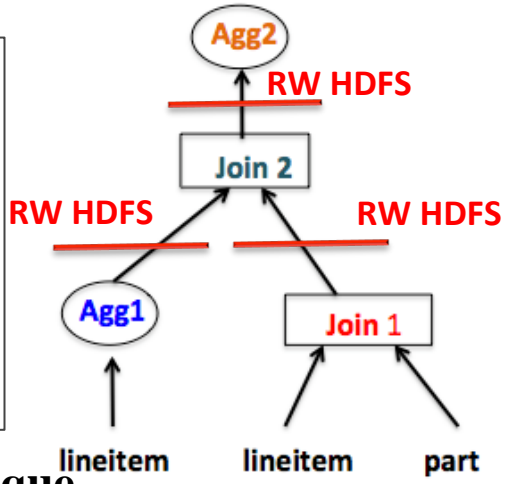


Traduction requête complète : YSmart

- YSmart: Yet Another SQL-to-MapReduce Translator, 2012
- Idée : optimiser DAG Map-Reduce en tentant de fusionner les jobs qui partitionnent sur la même clé ou qui lisent les mêmes données
- Déroulement
  - Traduction naïve : un job par opérateur
  - Identification des *corrélations* entre jobs
    - Input : même données lues -> Map mutualisé -> gain local
    - Transit : partitionnement sur la même clé -> Map et Reduce mutualisés -> gain global

Illustration YSmart

```
SELECT sum(l_extendedprice) / 7.0 AS avg_yearly
FROM (SELECT l_partkey, 0.2* avg(l_quantity) AS t1
        FROM lineitem GROUP BY l_partkey) AS inner,
       (SELECT          l_partkey,l_quantity,l_extendedprice
       FROM lineitem, part
       WHERE p_partkey = l_partkey) AS  outer
WHERE outer.l_partkey = inner.l_partkey; AND outer.l_quantity <
inner.t1;
```



La requête R et son arbre algébrique

Agg1
Map : lineitem→(l_partkey,l_quantity)
Reduce : v = 0.2* **avg(**l_quantity**)**

Join1
Map : lineitem→(l_partkey, (l_quantity, l_extendedprice))
part→(l_partkey, null)
Reduce : jointure

Join2
Map : inner→(l_partkey,**v**)
outer →(l_partkey,(l_quantity , l_extendedprice))
Reduce : jointure

Agg2 …

Traduction naïve

Coût communication !!

Traduction optimisée

Join

Map :

lineitem→(p_partkey , (l_quantity l_extendedprice))

part→(l_partkey, null)

Reduce :

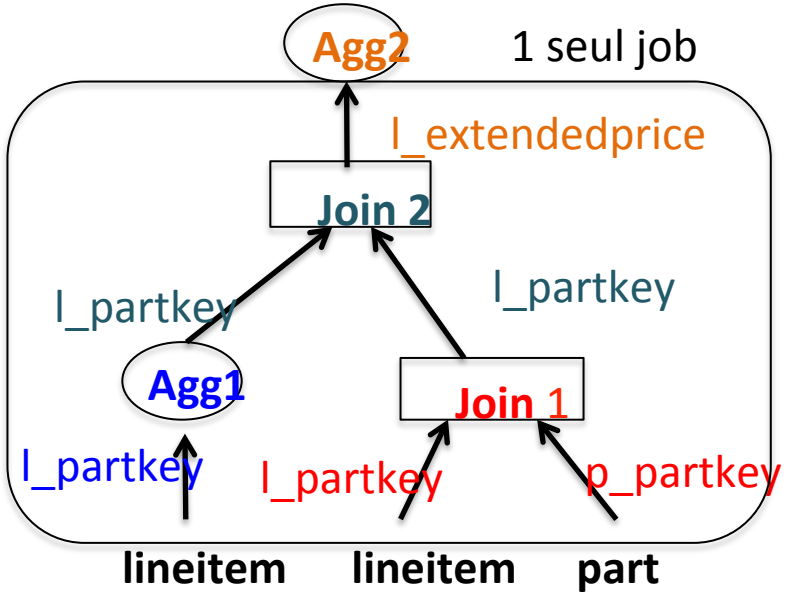
agg1 sur l_quantity

join1 sur l_partkey = p_partkey

join2 sur l_partkey = l_partkey

Agg 2

…



Inconvénient de l'approche : nécessite modification du système sous-jacent

Intégrée à l'optimiseur de Hive (en option)

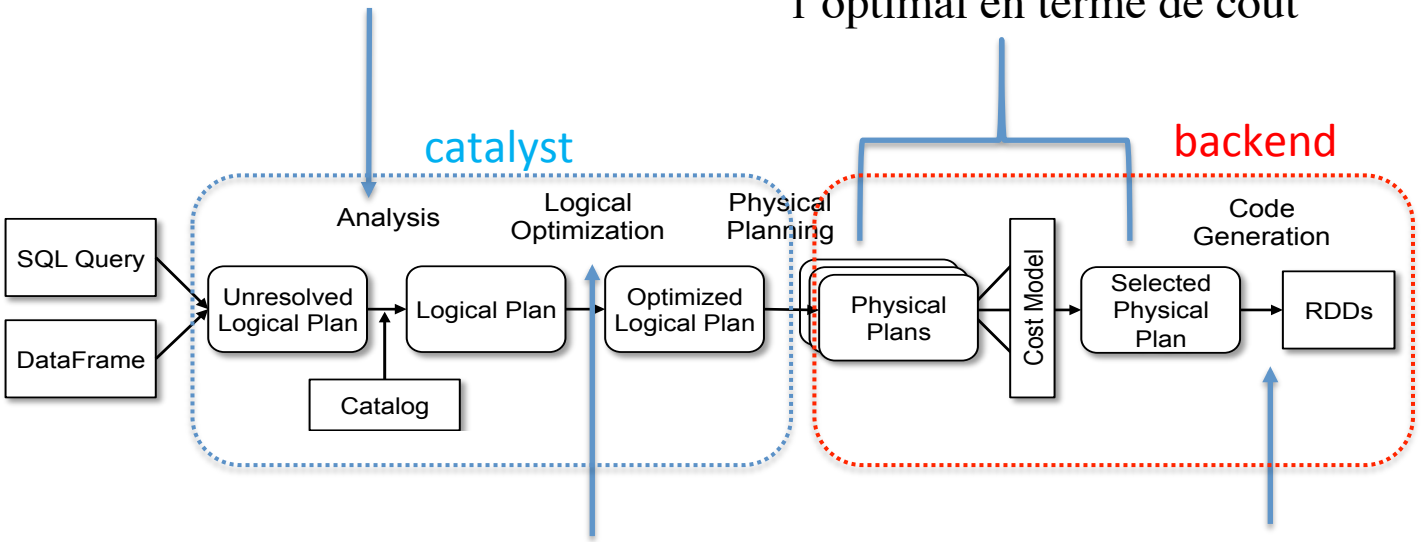
SQL sur Spark

- Deux approches
  - Utiliser l'API RDD et traduire requêtes à la main
    - Peu d'opportunités d'optimisation car par de distinction entre plan logique et physique
  - Tirer profit de Spark SQL et de son optimiseur
    - Distinction entre plan logique et plan physique
    - Optimisation logique basée sur les règles, extensibles
    - Optimisation physique basée sur le cout

Optimisation 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

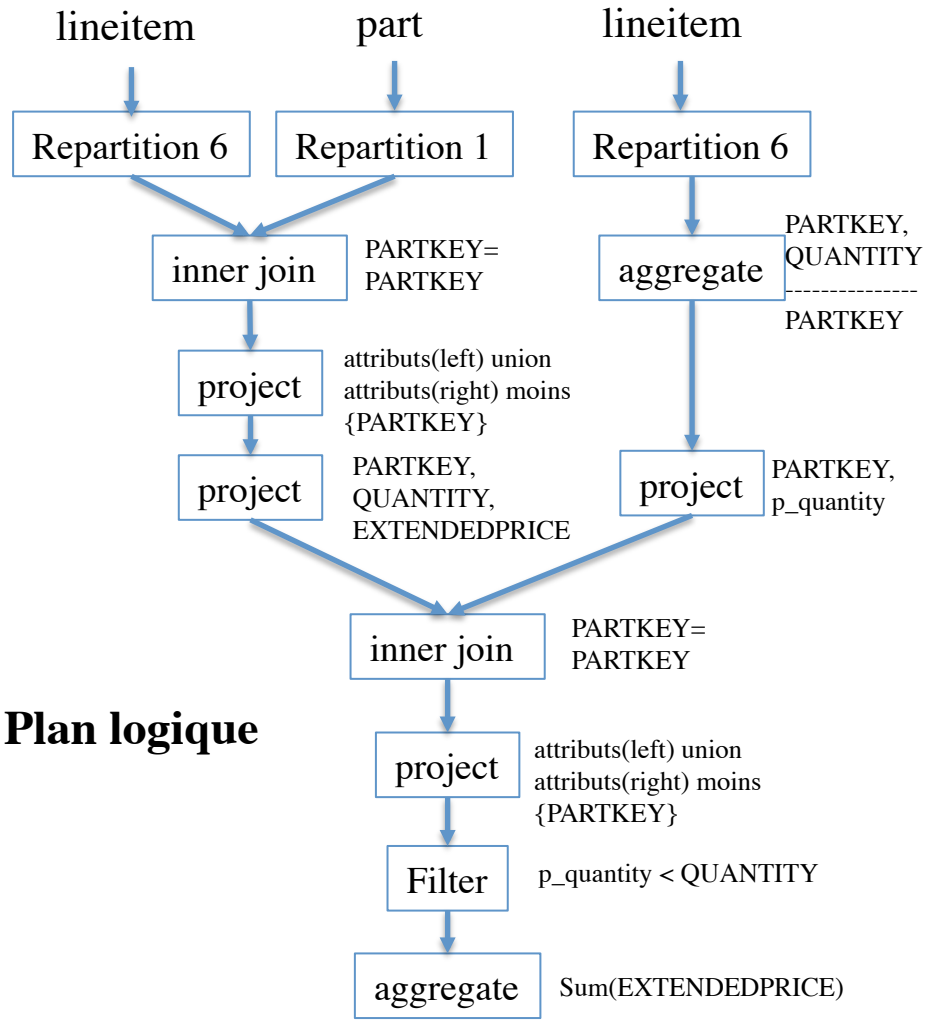
```
val lineitem = spark.read. ... .load(lineitem_t).coalesce(6)
val part = spark.read...load(part_t) coalesce(1)

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)
```

Tpch Q17 simplifiée



Plan logique

Spark Explain

```
scala> q17_simp.explain(true)
== Parsed Logical Plan ==
…..

== Analyzed Logical Plan ==
(sum(EXTENDEDPRICE) / 7): double
Aggregate [(sum(EXTENDEDPRICE#127) / cast(7 as double)) AS (sum(EXTENDEDPRICE) / 7)#148]
+- Filter (p_quantity#92 < cast(QUANTITY#126 as double))
   +- Project [PARTKEY#11, p_quantity#92, QUANTITY#126, EXTENDEDPRICE#127]
      +- Join Inner, (PARTKEY#11 = PARTKEY#123)
         :- Project [PARTKEY#11, avg(QUANTITY)#89 AS p_quantity#92]
         :  +- Aggregate [PARTKEY#11], [PARTKEY#11, avg(cast(QUANTITY#14 as bigint)) AS avg(QUANTITY)#89]
         :     +- Repartition 6, false
         :        +- Relation[ORDERKEY#10,PARTKEY#11,SUPPKEY#12,LINENUMBER#13,QUANTITY#14,EXTENDEDPRICE#15,DISCOUNT#
         +- Project [PARTKEY#123, QUANTITY#126, EXTENDEDPRICE#127]
            +- Project [PARTKEY#123, ORDERKEY#122, SUPPKEY#124, LINENUMBER#125, QUANTITY#126, EXTENDEDPRICE#127, DISC(
               +- Join Inner, (PARTKEY#123 = PARTKEY#53)
                  :- Repartition 6, false
                  :  +- Relation[ORDERKEY#122,PARTKEY#123,SUPPKEY#124,LINENUMBER#125,QUANTITY#126,EXTENDEDPRICE#127,DIS
                  +- Repartition 1, false
                     +- Relation[PARTKEY#53,NAME#54,MFGR#55,BRAND#56,TYPE#57,SIZE#58,CONTAINER#59,RETAILPRICE#60,COMMENT
```

(2) Optimisation du plan logique

- Catalyst : réécriture de l'arbre d'opérateurs
  - Spark 2.4 : plus de 100 règles regroupées par lots (batch)
  - écrites en 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 les attributs inutiles
    - *CollapseProject* : Combinaison des opérateurs de projection
    - *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

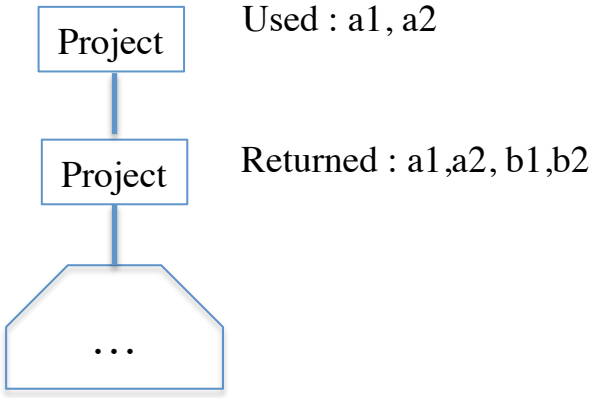
Signatures de opérateurs algébriques

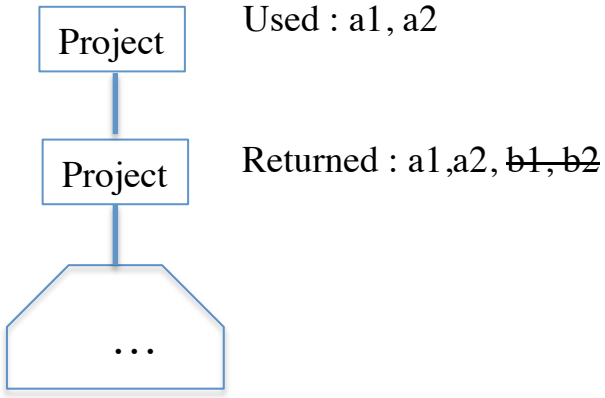
- Structure récursive
  - Op(arg_1, …, arg_n, child), où child est un arbre désignant un plan logique
  - op.used (resp. op.returned) désignent la liste des attributs utilisés (resp. retournés)
  - op.arg_i pour accéder à l'argument (comme pour un objet)
- Quelques exemples
  - Filter(cond, child) : cond est la condition à verifier
  - 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*/



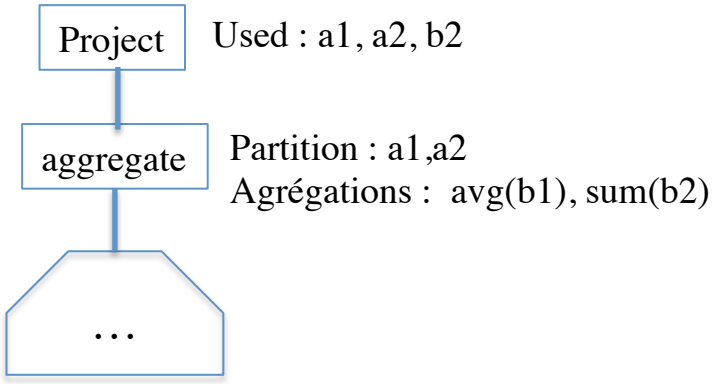


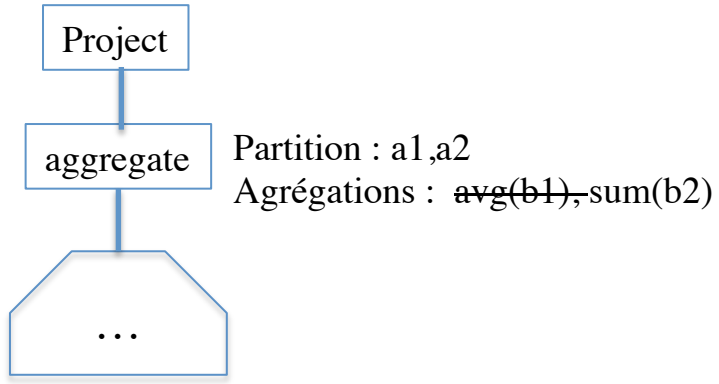
Lot *ColumnPruning*

Règle d'éliminations des agrégations inutiles

cas op = Project(_, a: Aggregate)

- si a.returned ⊄ 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*/

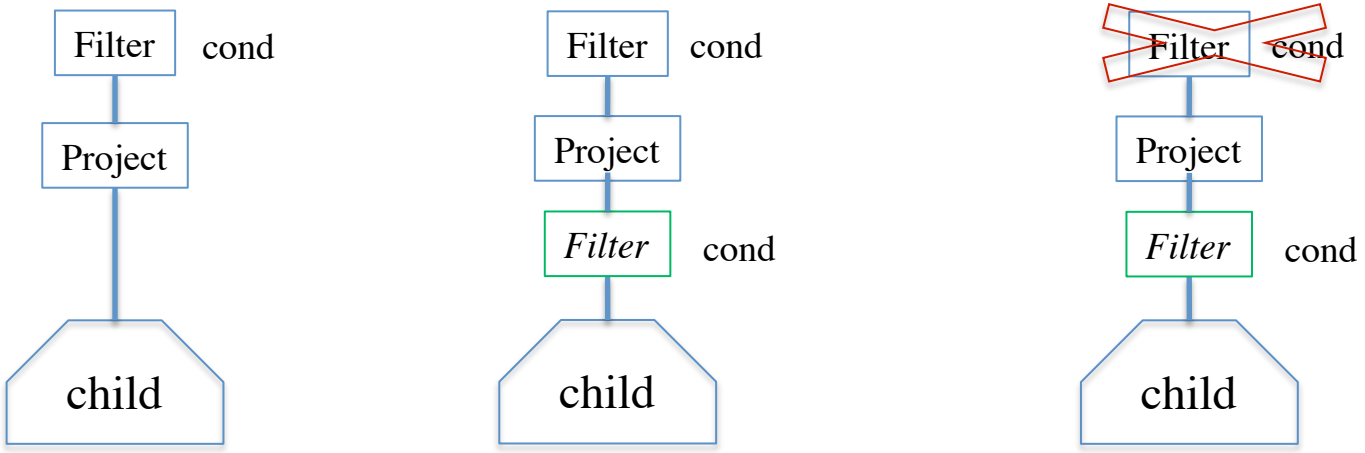




Règle *PushDownPredicate*

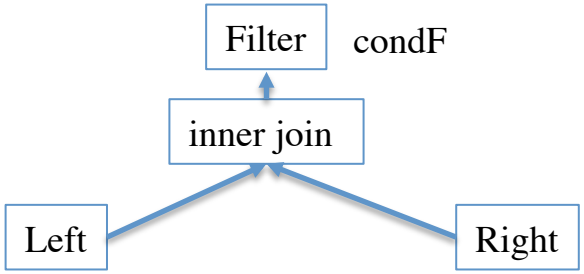
cas op=Filter(cond, Project(_, child))

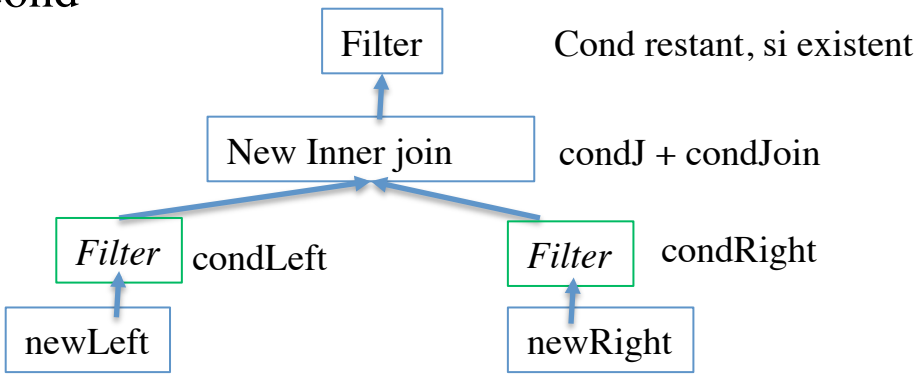
- si cond est déterministe et peut être poussée
- alors Filter(cond, Project(_, Filter(cond, child)))
- Filter le plus externe sera éliminé après des vérifications



Règle *PushPredicateThroughJoin*

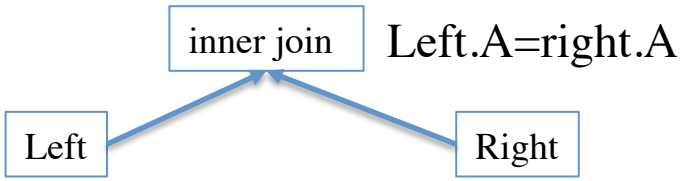
- Diviser la condition en 2 sous-conditions qui pourraient être évaluées dans une des branches de la jointure et une évaluée avec la jointure
- Filter(condF, Join(left, right, joinType, CondJ)
  - Diviser fCond en leftCond, rightCond et joinCond
  - cas joinType = Inner
    - newLeft := left où child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

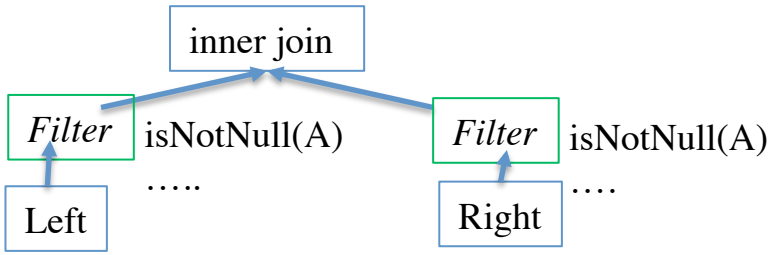




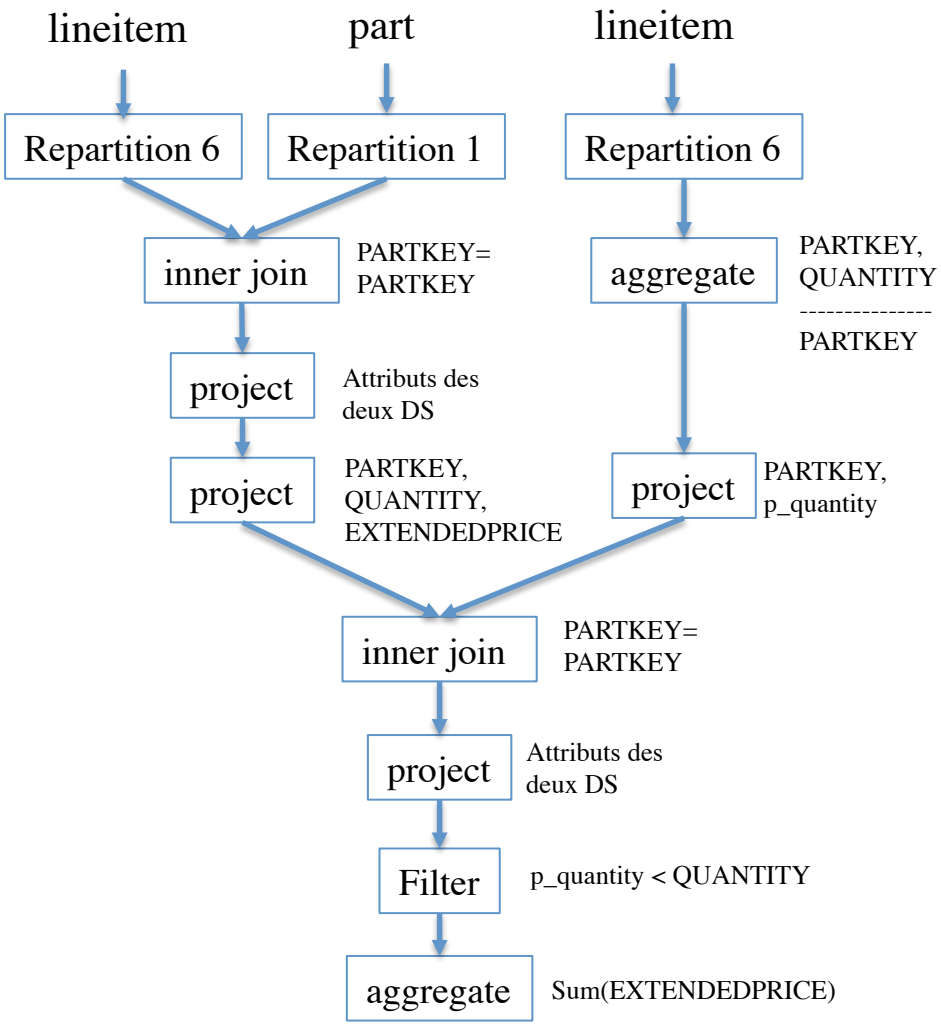
Règle *InferFiltersFromConstraints*

- 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



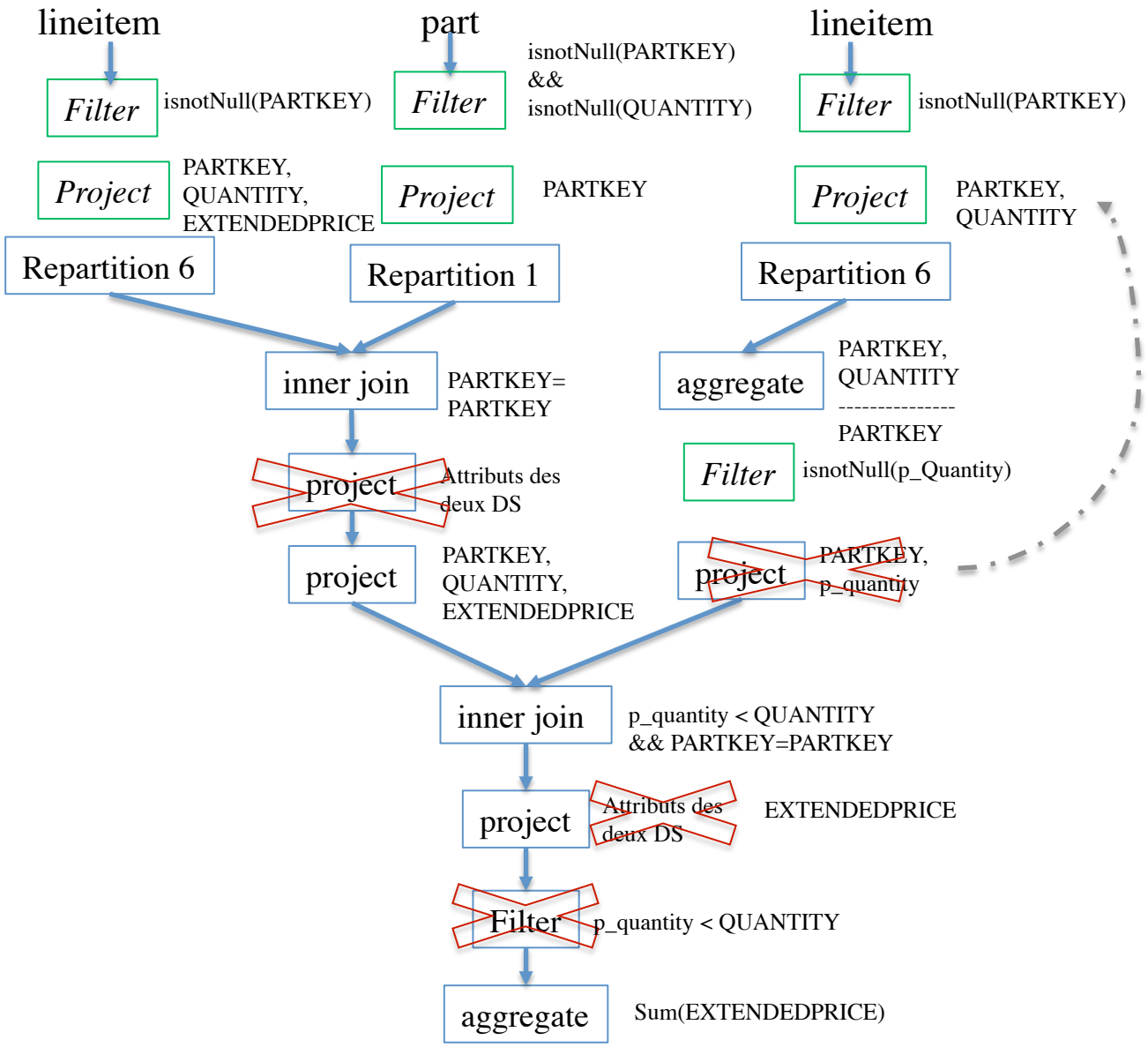


Plan logique Q17



Attention sens de lecture inversé

Plan logique Q17 optimisé



Attention sens de lecture inversé

Spark Explain

```
…
== Analyzed Logical Plan ==
…
== Optimized Logical Plan ==
Aggregate [(sum(EXTENDEDPRICE#127) / 7.0) AS (sum(EXTENDEDPRICE) / 7)#148]
+- Project [EXTENDEDPRICE#127]
  +- Join Inner, ((p_quantity#92 < cast(QUANTITY#126 as double)) && (PARTKEY#11 = PARTKEY#123))
    :- Filter isnotnull(p_quantity#92)
    :  +- Aggregate [PARTKEY#11], [PARTKEY#11, avg(cast(QUANTITY#14 as bigint)) AS p_quantity#92]
    :     +- Repartition 6, false
    :        +- Project [PARTKEY#11, QUANTITY#14]
    :           +- Filter isnotnull(PARTKEY#11)
    :              +- Relation[ORDERKEY#10,PARTKEY#11,SUPPKEY#12,LINENUMBER#13,QUANTITY#14,EXTENDEDPRICE#15,DISCOUN
    +- Project [PARTKEY#123, QUANTITY#126, EXTENDEDPRICE#127]
      +- Join Inner, (PARTKEY#123 = PARTKEY#53)
        :- Repartition 6, false
        :  +- Project [PARTKEY#123, QUANTITY#126, EXTENDEDPRICE#127]
        :     +- Filter (isnotnull(PARTKEY#123) && isnotnull(QUANTITY#126))
        :        +- Relation[ORDERKEY#122,PARTKEY#123,SUPPKEY#124,LINENUMBER#125,QUANTITY#126,EXTENDEDPRICE#127,DIS
        +- Repartition 1, false
          +- Project [PARTKEY#53]
            +- Filter isnotnull(PARTKEY#53)
              +- Relation[PARTKEY#53,NAME#54,MFGR#55,BRAND#56,TYPE#57,SIZE#58,CONTAINER#59,RETAILPRICE#60,COMMENT
```

(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
  - 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?

Spark Explain

```
…
== Optimized Logical Plan ==
…
== Physical Plan ==
*(9) HashAggregate(keys=[], functions=[sum(EXTENDEDPRICE#127)], output=[(sum(EXTENDEDPRICE) / 7)#148])
+- Exchange SinglePartition
   +- *(8) HashAggregate(keys=[], functions=[partial_sum(EXTENDEDPRICE#127)], output=[sum#169])
      +- *(8) Project [EXTENDEDPRICE#127]
         +- *(8) SortMergeJoin [PARTKEY#11], [PARTKEY#123], Inner, (p_quantity#92 < cast(QUANTITY#126 as double))
            :- *(3) Sort [PARTKEY#11 ASC NULLS FIRST], false, 0
            :  +- *(3) Filter isnotnull(p_quantity#92)
            :     +- *(3) HashAggregate(keys=[PARTKEY#11], functions=[avg(cast(QUANTITY#14 as bigint))], output=[PARTKEY#11, p_quantity#9
            :        +- Exchange hashpartitioning(PARTKEY#11, 200)
            :           +- *(2) HashAggregate(keys=[PARTKEY#11], functions=[partial_avg(cast(QUANTITY#14 as bigint))], output=[PARTKEY#11, su
            :              +- Coalesce 6
            :                 +- *(1) Project [PARTKEY#11, QUANTITY#14]
            :                    +- *(1) Filter isnotnull(PARTKEY#11)
            :                       +- *(1) FileScan csv [PARTKEY#11,QUANTITY#14] Batched: false, Format: CSV, Location: InMemoryFileIndex[hdfs://pp
            +- *(7) Sort [PARTKEY#123 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(PARTKEY#123, 200)
                  +- *(6) Project [PARTKEY#123, QUANTITY#126, EXTENDEDPRICE#127]
                     +- *(6) BroadcastHashJoin [PARTKEY#123], [PARTKEY#53], Inner, BuildRight
                        :- Coalesce 6
                        :  +- *(4) Project [PARTKEY#123, QUANTITY#126, EXTENDEDPRICE#127]
                        :     +- *(4) Filter (isnotnull(PARTKEY#123) && isnotnull(QUANTITY#126))
                        :        +- *(4) FileScan csv [PARTKEY#123,QUANTITY#126,EXTENDEDPRICE#127] Batched: false, Format: CSV, Location: Inl
                        +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, true] as bigint)))
                           +- Coalesce 1
                              +- *(5) Project [PARTKEY#53]
                                 +- *(5) Filter isnotnull(PARTKEY#53)
                                    +- *(5) FileScan csv [PARTKEY#53] Batched: false, Format: CSV, Location: InMemoryFileIndex[hdfs://ppti-dac-1:50100/A
```

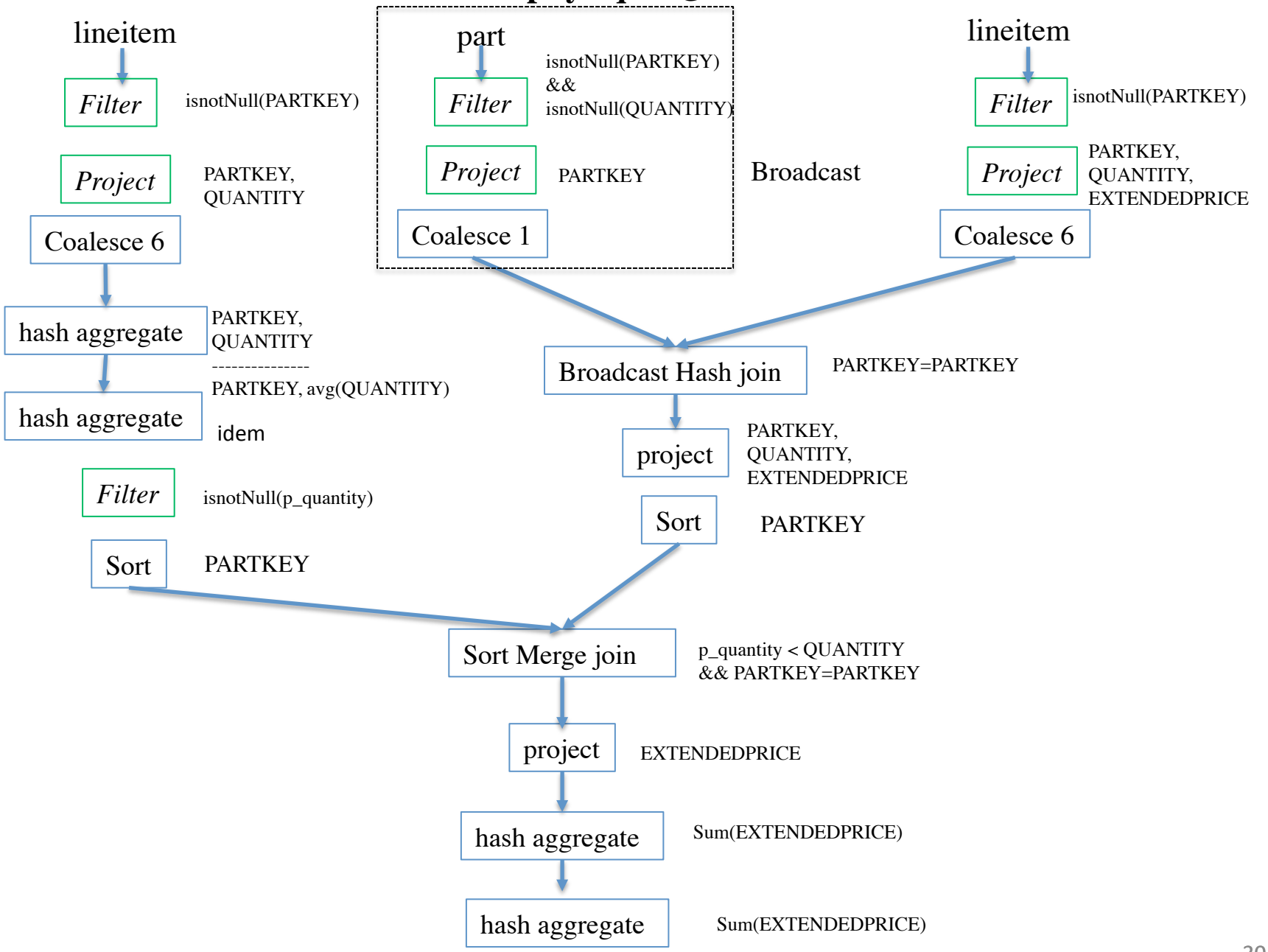
(4) Génération du code optimisé

- Rappel : Scala langage objet et fonctionnel
  - Cout de sérialisation, switch context

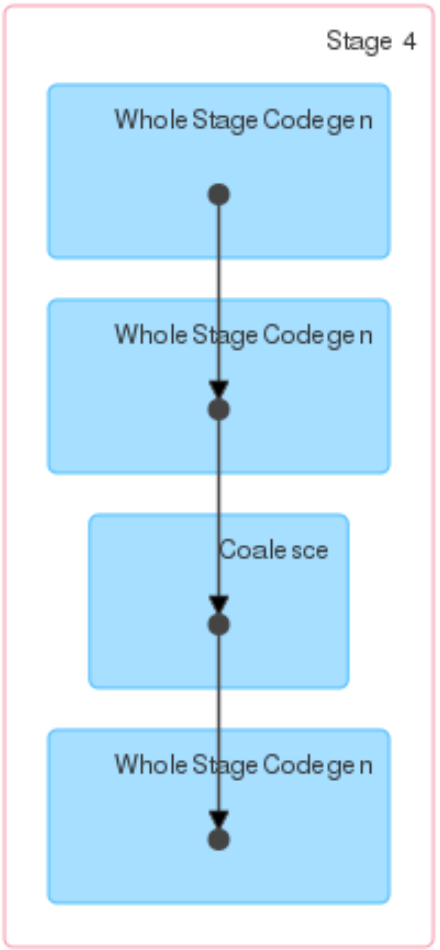
  →utilisation de techniques connues pour diminuer de calcul (ex. inlining)

  Out of the scope pour nous

Plan physique Q17

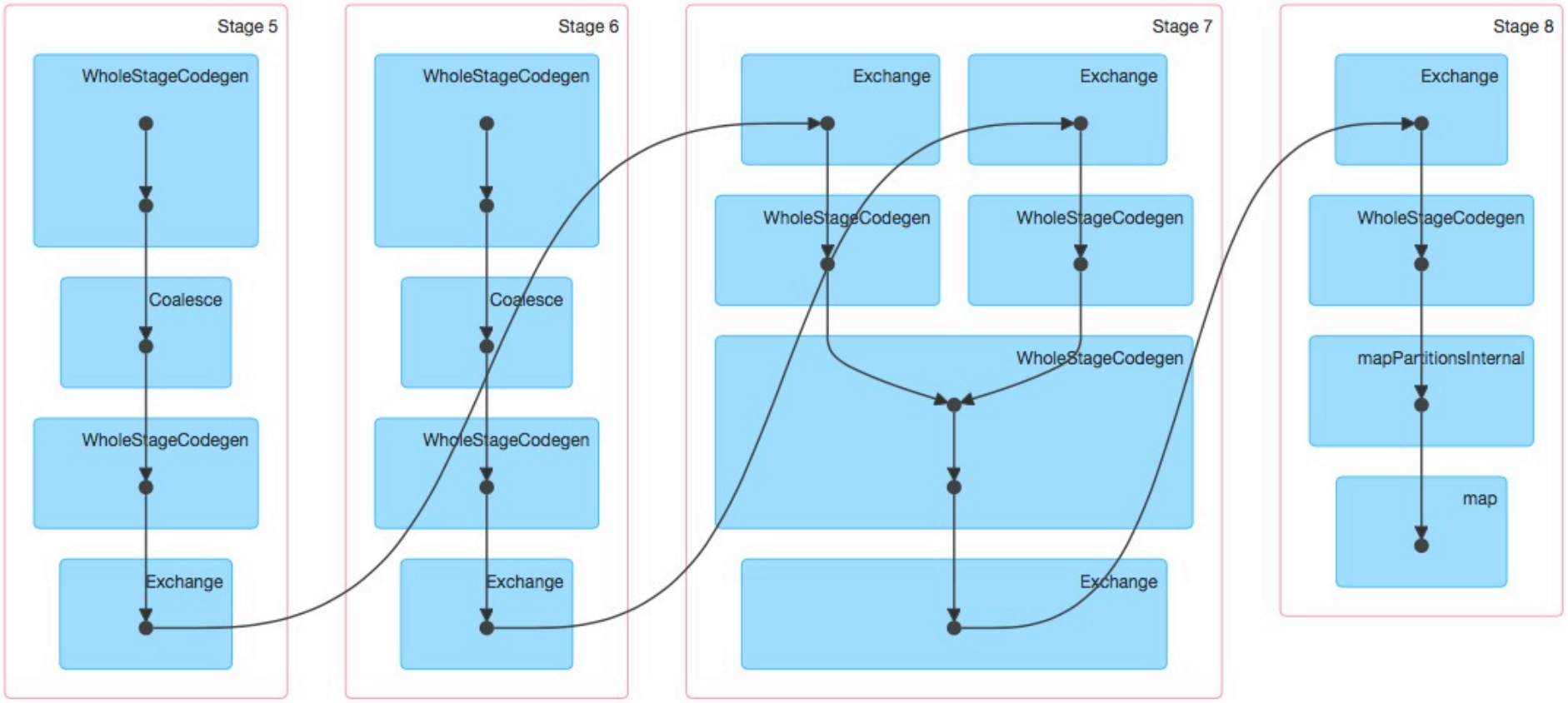


Visualisation plan



Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	run at ThreadPoolExecutor.java:1142 +details	2018/11/07 16:45:13	1 s	1/1	23.2 MB			

Visualisation plan



Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
8	show at <console>:26	+details	2018/10/22 19:36:26	84 ms	1/1			11.5 KB	
7	show at <console>:26	+details	2018/10/22 19:36:24	2 s	200/200			98.0 MB	11.5 KB
6	show at <console>:26	+details	2018/10/22 19:36:11	13 s	6/6	721.1 MB			86.2 MB
5	show at <console>:26	+details	2018/10/22 19:36:10	12 s	6/6	721.1 MB			11.8 MB

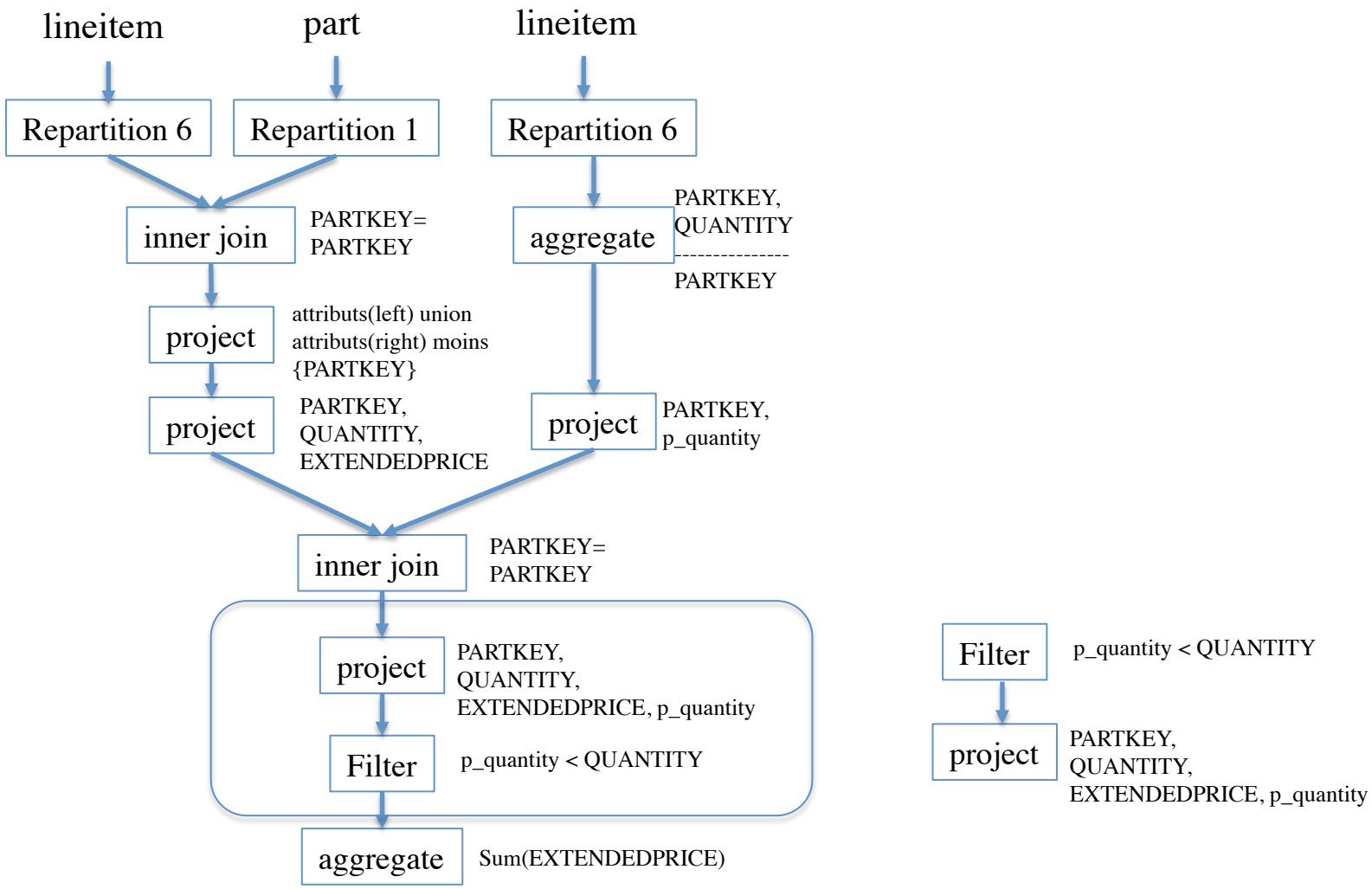
Autres pistes d'optimisation

- L'organisation physique des données compte!
  - Format de stockage colonnes : ORC, parquet
  - Partitionnement sur disque
  - Caching de donées accédés en répétition

Règles déclenchées pour tpch-q17

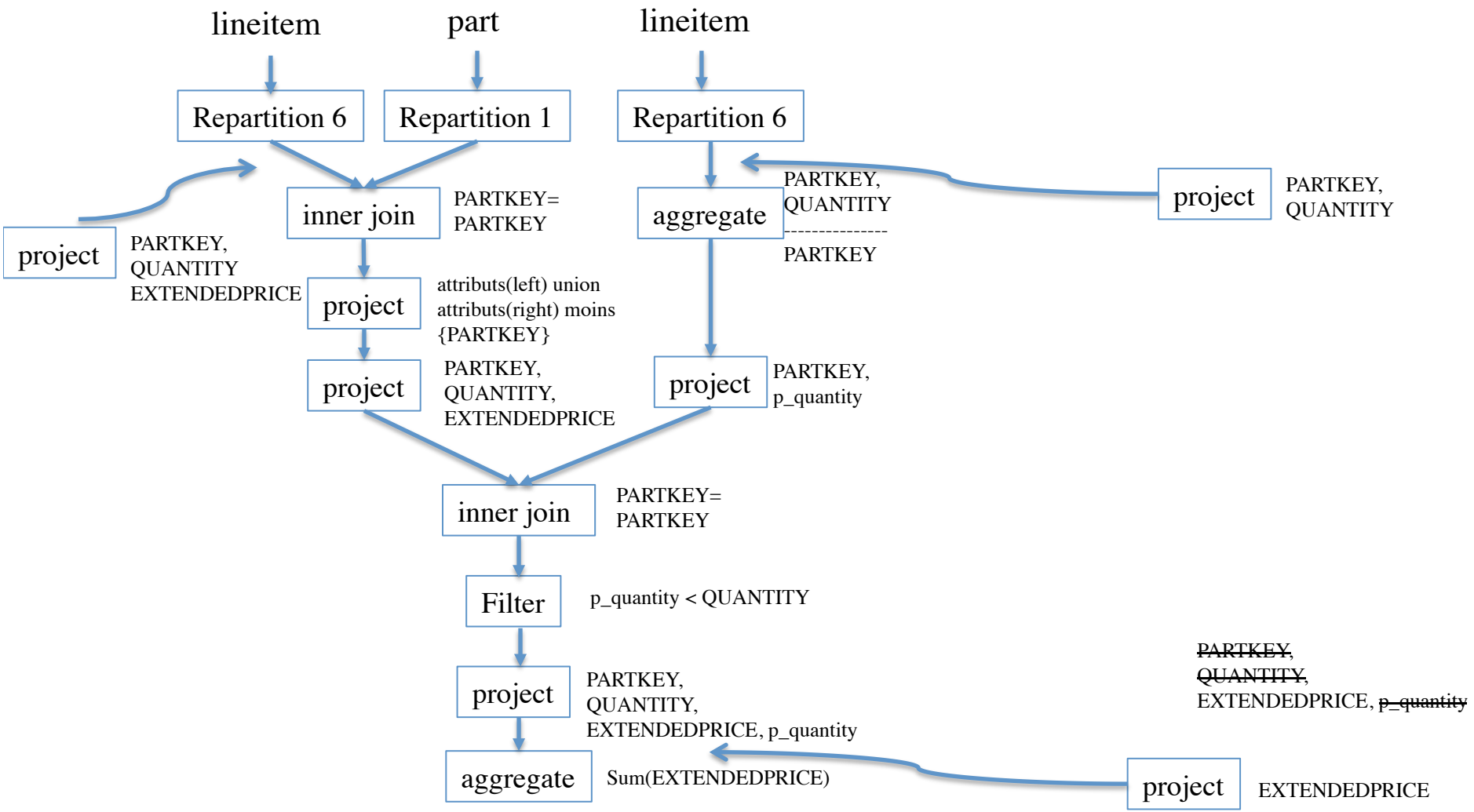
1. PushDownPredicate
2. ColumnPruning
3. CollapseProject
4. ConstantFolding
5. RemoveRedundantProject
6. PushPredicateThroughJoin
7. InferFiltersFromConstraints
8. PushPredicateThroughJoin
9. PushDownPredicate
10. PushPredicateThroughJoin
11. PushDownPredicate
12. ColumnPruning
13. CombineFilters
14. RemoveRedundantProject
15. PushDownPredicate

1-PushDownPredicate



Plan logique

2-ColumnPruning



Plan logique

3-CollapseProject