BDLE
Traitement des jointures dans Spark

Décembre 2020

Plan

- Intro
  - Objectifs, Contexte général
- Equi-jointure
  - Clé simple, clé composée
- Traitement des jointures
- Produit cartésien

Objectifs

- Comprendre le traitement des jointures dans un cluster de machines
  - Répartition des données
  - Transfert de données
  - Traitement en parallèle
- Savoir traiter efficacement la jointure de larges quantités de données
  - Passage à l'échelle des algorithmes de jointures
- Application dans la plateforme Spark
  - compétences pratiques
  - contrôler l'exécution des jointures

Contexte Général

- Environnement d'exécution : un cluster de machines
  - Plusieurs machine interconnectées
    - Le réseau a une capacité *fixée*
  - Une machine = disque, mémoire, plusieurs cœurs de calcul
    - Une machine dispose d'une quantité *fixée* de mémoire
- Données fragmentées sur les machines
  - en mémoire et sur disque
  - car la taille des données > somme des tailles des mémoires
  - Un disque n'est jamais plein
    - Il y a suffisamment de machines donc de disques
- Traitement de requêtes
  - Découper un traitement (*job*) en une *séquence* d'étapes
  - Une étape (*stage*) = plusieurs tâches indépendantes en *parallèles*
  - Entrée**s** → tâche de calcul → sortie
  - Entrée distante implique un transfert à la demande

Equi-jointure sur **clé simple**

- Equi-jointure sur la clé K
  - Soient deux collections R et S
  - R est un ensemble de **paires (clé, valeur)** : RDD[**(K, V)**]
    - K est un type simple : nombre ou string
  - S est un ensemble de paires (clé, valeur) : RDD[(K, W)]
    - Les types V et W sont quelconques
- Le prédicat de jointure est l'égalité des clés K
- La jointure de R avec S est notée **J = R.join(S)**
  - Type du résultat J : ensemble de paires (clé, valeur)
    - RDD[(K, (V, W))]
    - Structure imbriquée: la valeur est elle-même une paire (V, W)

Exemple de jointure

- Les utilisateurs

  (Alice, Paris) (Bob, Londres) .... (Zoé, Paris)

- Les notes attribuées à des films

  (Alice, StarWars, 5) (Bob, Matrix, 3) (Alice, Matrix, 4)

- Jointure entre Utilisateurs et Notes

  (Alice, Paris, StarWars, 5) (Alice, Paris, Matrix, 3) (Bob, Londres, Matrix, 4)

Equi-jointure sur **clé composée**

- Données d'entrée
  - R(a, b, c, d) S(a, b, x, y, z)
- Equi-jointure en SQL

  SELECT * FROM R, S WHERE R.a = S.a **AND** R.b = S.b
- Equi-jointure en Spark
  - Structurer les données d'entrée pour avoir une clé composée
    - R = R.map({ case (a, b, c, d) => **((a, b)**, **(c, d))** })
    - R contient des paires **(K**, **V)** : K est **composée** de (a, b)
    - S = S.map({ case (a, b, x, y, z) => **((a, b)**, **(x, y, z))** })
  - Jointure
    - J = R.join(S)
    - Prédicat composé : R.a = S.a **ET** R.b = S.b
  - J contient les paires **(K**, W **)** telles que
    - K est (a, b)
    - W est ((c, d), (x, y, z))
- Structure générale d'une clé : nuplet avec imbrication possible

Exécuter la jointure

- J est définie mais n'est pas encore évaluée
  - Demander explicitement à évaluer J
- Invoquer une **action** qui **évalue** les transformations de la chaîne J

- Exemple d'actions :
  - J.count() compter le nombre d'éléments de J
  - J.take(3) lire 3 éléments de J

Algorithmes de jointure **parallèle**

2 algorithmes pour évaluer une jointure parallèle :

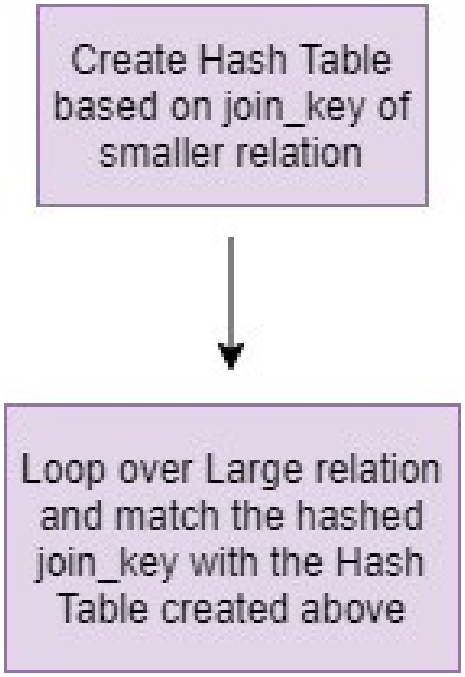
- Jointure par hachage et partitionnement (Pjoin)
  - Répartir les données par **hachage sur la clé de jointure**
  - Une donnée → une destination

- Jointure par diffusion : Broadcast join (Brjoin)
  - Diffuser une donnée vers **plusieurs** destinations

Jointure par hachage et partitionnement

Partitioned join

Principe du hash join



Principe du sort merge join

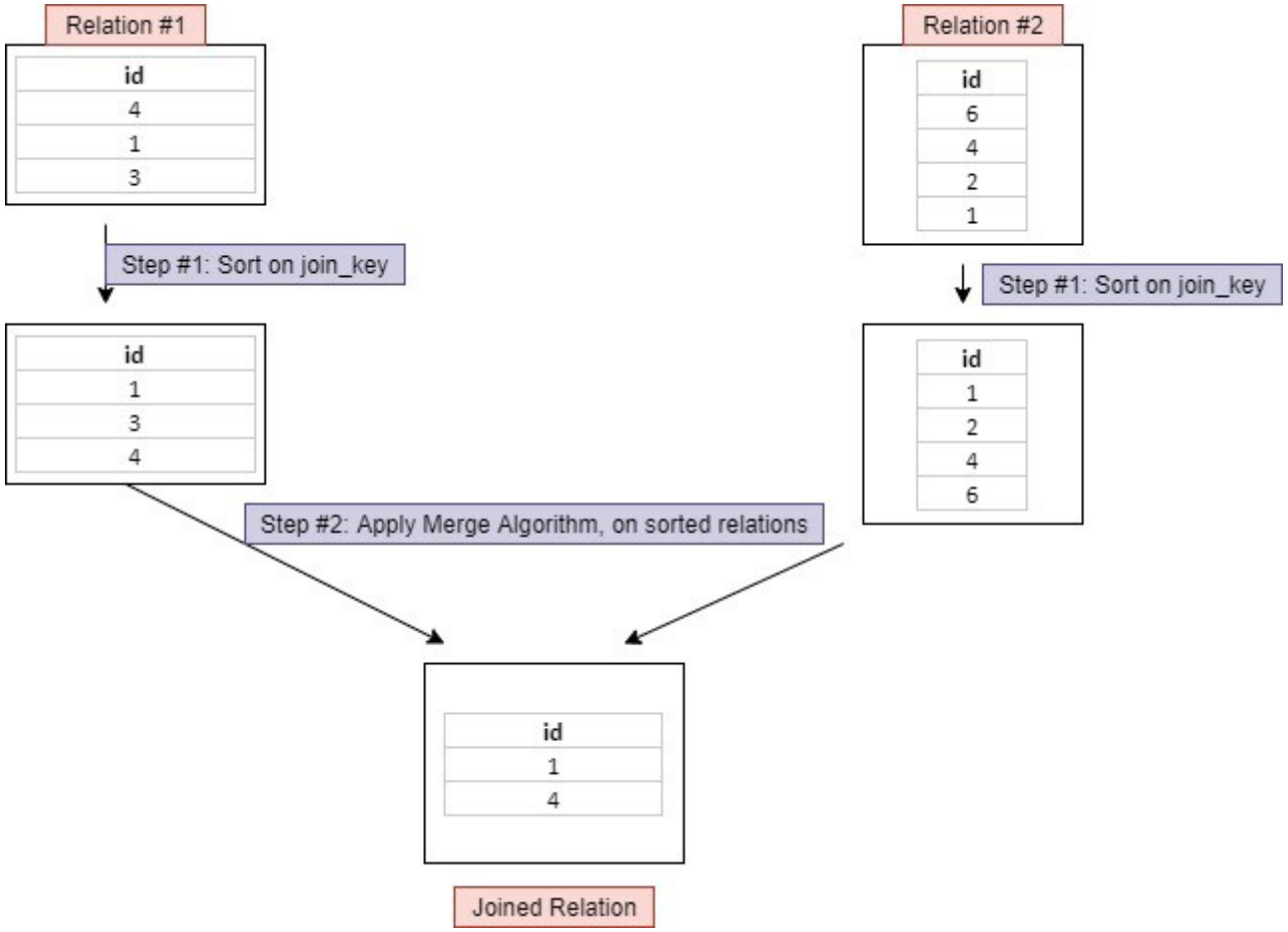
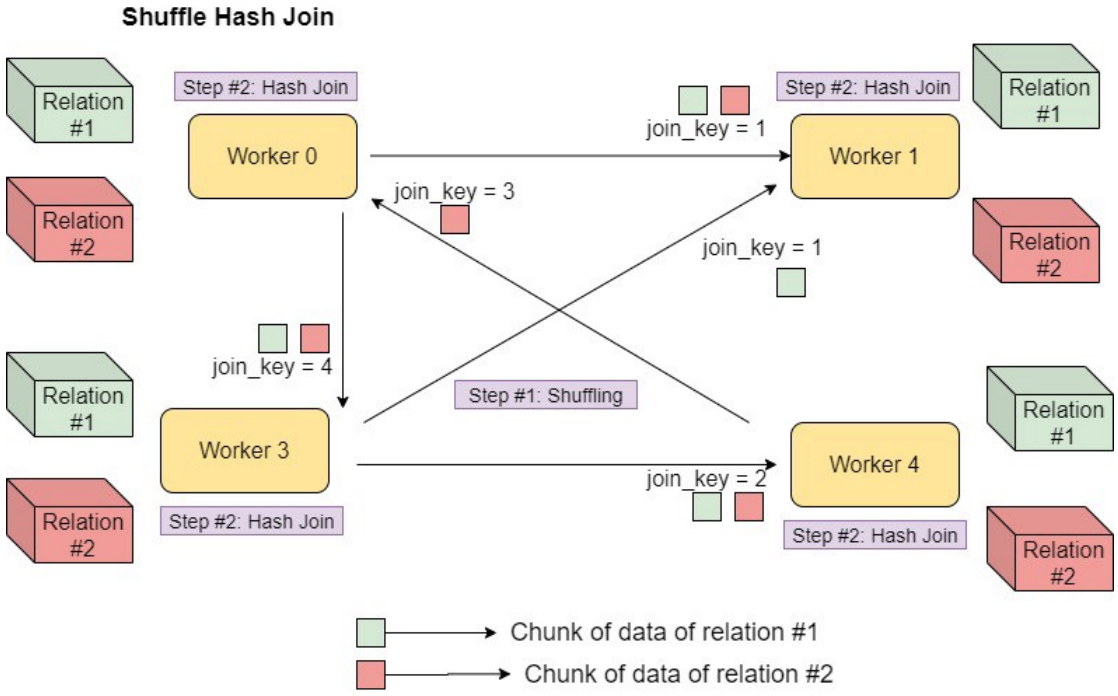


Illustration d'une jointure distribuée (ie, répartie)



Traitement réparti d'une équi-jointure

- Modèle "tout réparti"
  - Les données d'entrées **et de sortie** sont réparties
  - Les traitement sont répartis en plusieurs tâches
- Les données d'entrée sont **réparties**
  - Données déjà fragmentées en **N** fragments *a priori*
    - **Pas** forcément fragmentées selon la clé de jointure : plusieurs fragments peuvent contenir des données ayant la même clé K
  - Les fragments sont répartis sur plusieurs machines
    - Une machine peut contenir plusieurs fragments
- Les données du résultat de la jointure sont réparties
  - Car la génération du résultat est elle-même répartie
  - les **M** fragments sont générés indépendamment par M tâches
  - Les fragments J_1, ...J_i, ... J_M sont définis par **hachage** de la clé
    - le résultat concernant la clé K est calculé dans le fragment Ji tq. i = h(K)
  - M peut être différent de N
  - Choix de M ? Pour paralléliser au maximum : choisir M ≥ nb de cœurs

Traiter la jointure en deux étapes

- Etape locale
  - Evaluer tout ce qu'on peut localement, là où résident les données d'entrée
  - Correspond à un map()
- Etape répartie
  - Opération nécessitant de lire plusieurs fragments pouvant provenir de plusieurs machines
  - Aboutit à générer les M fragments J_1 à J_M
  - Correspond à un reduce() avec transfert
- Une étape est traitée par plusieurs tâches
  - Fixer M le nombre de tâches
  - Syntaxe RDD: R.join(S, **M**),
  - syntaxe Dataset: spark. sql.shuffle.partitions = M

Etape locale de la jointure

- Nb de tâches locales : une tâche par fragment de l'entrée
  - NR fragments de R → NR tâches locales
  - NS fragments de S → NS tâches locales
- Tâche locale
  - Lire et transformer un fragment d'entrée pour produire un *fragment intermédiaire* contenant les paires (K, V)
  - Réordonner un fragment intermédiaire en M plages notées P_1 à P_M
    - Plage Pi : toutes les paires (K, V) telles que h(K) = i
  - Tri *secondaire* à l'intérieur d'une plage P_i : trier les paires selon K
  - Si plusieurs fragments de R (resp. de S) sur une machine : les fusionner
    - préserve les plages
- A la fin de l'étape locale
  - Sur chaque machine contenant R (resp. S), il y a un fragment intermédiaire composé de M plages.

Etape répartie de la jointure

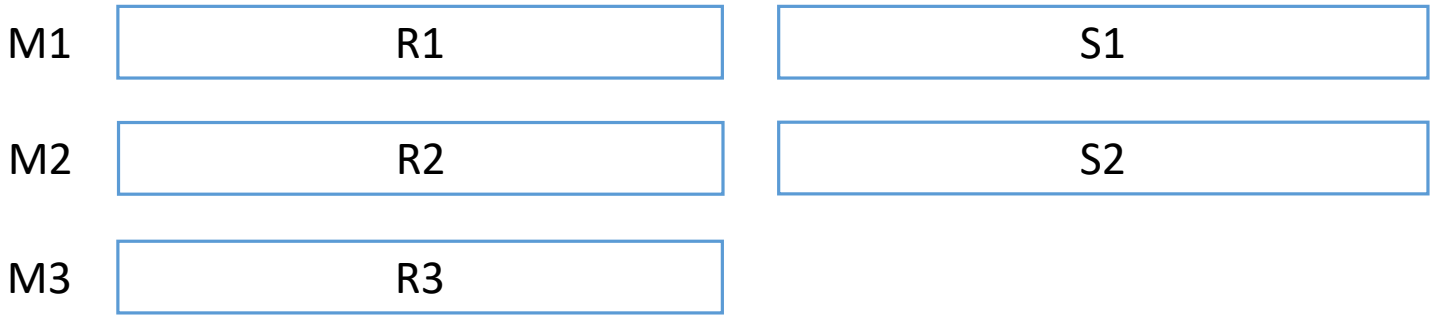
- M tâches indépendantes : T_1 à T_M
  - Peuvent être traitées en parallèle
- Tâche T_i
  - Lire la plage P_i de tous les fragments intermédiaires, pour R et S
    - Transfert à la demande (en cas de lecture distante)
  - Evaluer la jointure et générer le fragment J_i
    - Jointure par fusion car les fragments sont triés
    - Restructurer le résultat
- Avantage du transfert à la demande
  - Mode pull: le transfert est initié par le destinataire
    - Evite de transférer les données à l'avance, donc meilleure répartition des transferts dans le temps.
  - Si on choisit M > nombre de cœurs, alors meilleur parallélisme des tâches lorsque la jointure pour certaines plages est plus longues que pour d'autres.
    - Démarrer une tâche par cœur. Dès qu'un cœur termine sa tâche, lui attribuer une tâche suivante.

Illustration du traitement

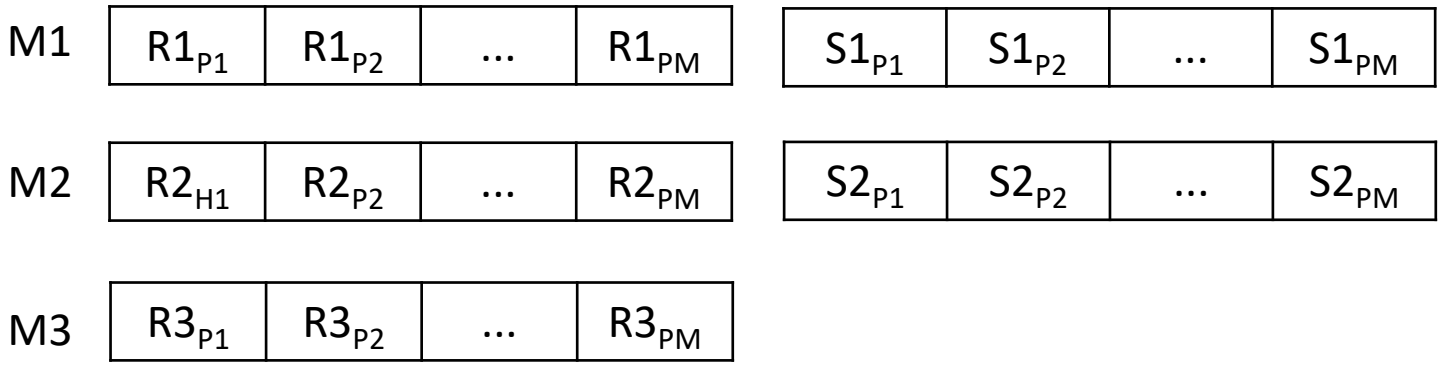
- Données : 2 collections R et S réparties sur 3 machines
  - Machine 1 : R_1 , S_1
  - Machine 2 : R_2 , S_2
  - Machine 3 : R_3
- Etape locale
  - Machine 1 : $\{R_{1,P1}$, ..., $R_{1,PM}\}$ et $\{S_{1,P1}$, ..., $S_{1,PM}\}$
  - Machine 2 : $\{R_{2,P1}$, ..., $R_{2,PM}\}$ et $\{S_{2,P1}$, ..., $S_{2,PM}\}$
  - Machine 3 : $\{R_{3,P1}$, ..., $R_{3,PM}\}$
- Etape répartie en utilisant 10 cœurs
  - Cœur 1 : traiter les données de la plage P_1
    - Récupérer toutes les plages $R_{1,P1}$ $R_{2,P1}$ et $R_{3,P1}$ en triant les paires (K, V) par K croissant.
    - Idem pour les plages $S_{1,P1}$ et $S_{2,P1}$
    - Jointure
  - Cœur 10 : idem avec les plages P_{10} de R et S
  - Traiter les plages P_{11} à P_M sur les cœurs qui se libèrent successivement

Détail de l'étape locale

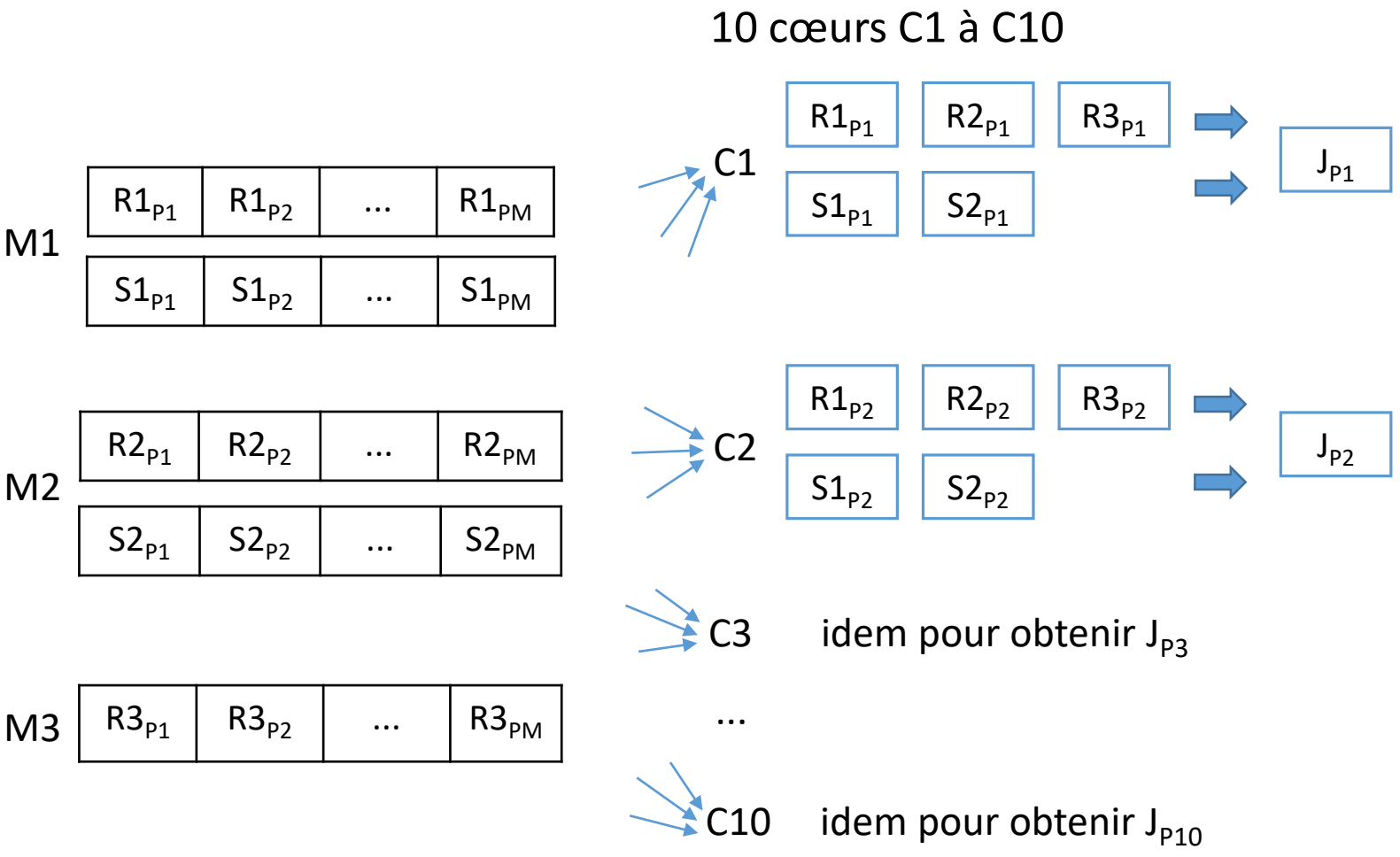
Les 2 collections R et S sont réparties sur les machines M1 à M3



Etape locale, tri selon h(K) = i pour former les plages



Détail de l'étape répartie



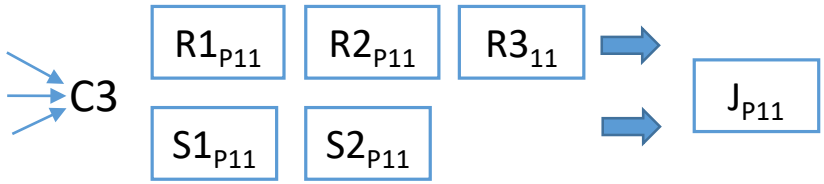
Suite de l'étape répartie

On a nombre de plages > nombre de cœurs

Calcul successif des fragments J_{Pi} suivants

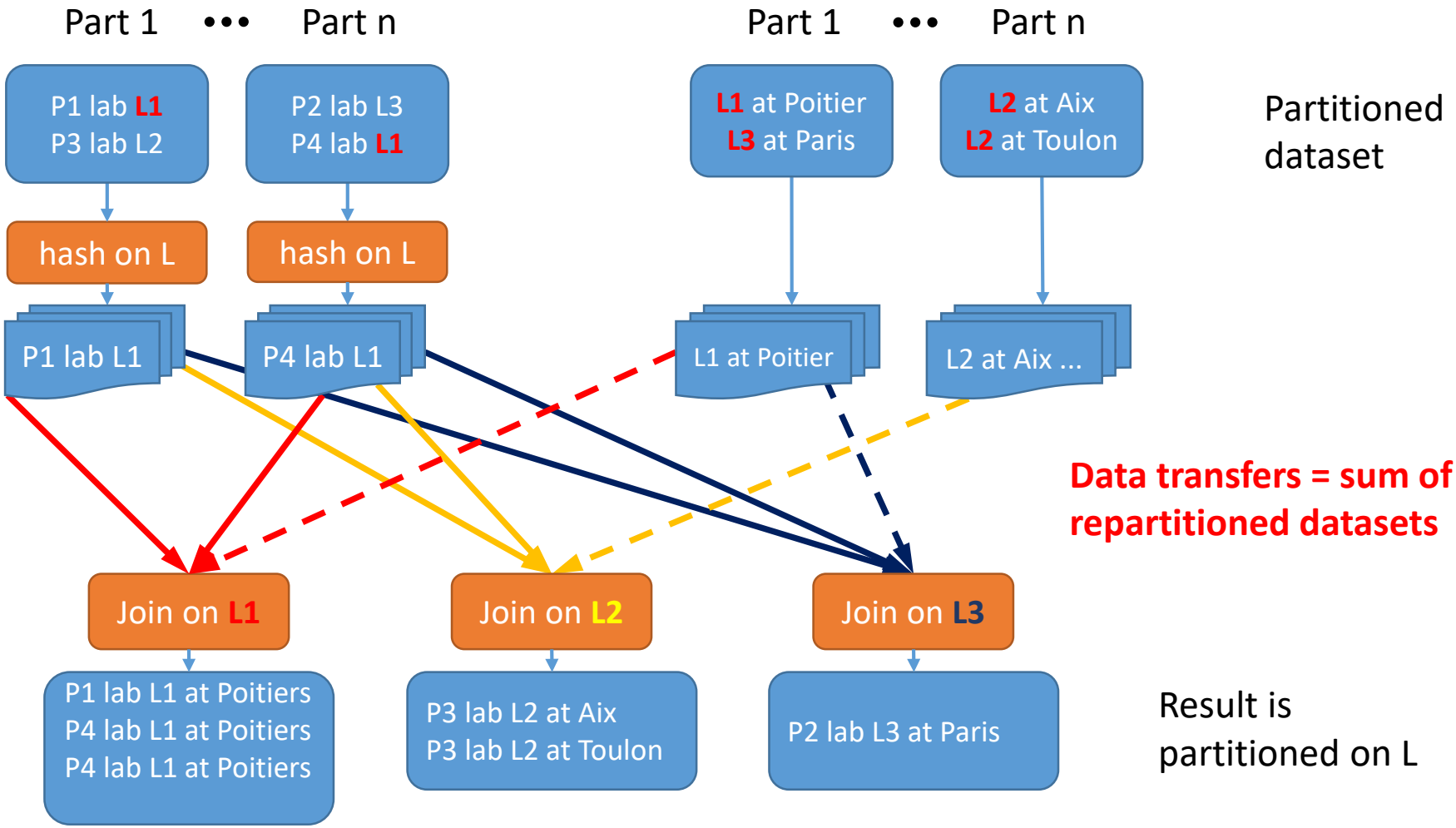
Le premier cœur C qui termine son calcul, continue avec J_{P11}

Exemple quand C3 termine :



Ainsi de suite pour calculer J_{P12}, ..., J_{PM}

Exemple de Jointure par hachage



Jointure : scalabilité

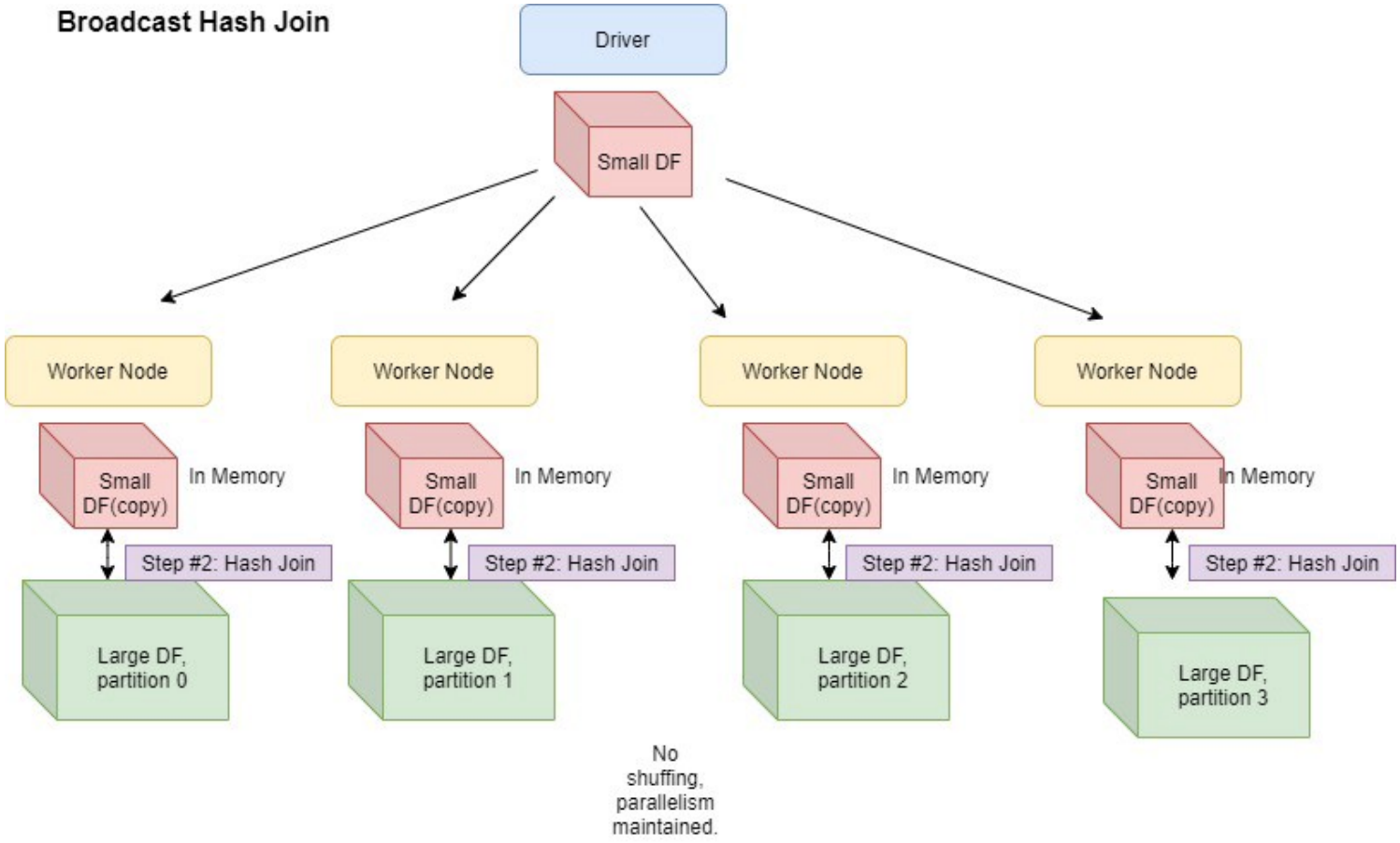
- Le traitement est conçu pour **passer à l'échelle**
  - Calculer une jointure quelle que soit la taille des données et le nombre de machines
- Exécution dans un cluster de machines
  - Contrainte : une machine dispose d'une quantité **fixée** de mémoire.
  - Hypothèse : disque jamais plein, car on peut (re)fragmenter les données.
  - Nombre de machine illimité
- Etape locale
  - Ecrire une plage *partielle* sur disque dès qu'elle ne tient plus en mémoire (spill)
  - Fusionner les plages partielles
    - Fusion en plusieurs passes si le nombre de plages est trop grand (cas rare)
- Etape répartie
  - Petite quantité de mémoire nécessaire
    - Pour toute clé K, seul l'ensemble des paires (K, V) venant de R (**ou** de S) doit tenir en mémoire.
    - Pas besoin qu'une plage entière tienne en mémoire
  - Si trop de paires pour un certain K : les écrire sur disque puis boucle imbriquée

Jointure par broadcast

Broadcast Join

- Cette méthode est généralement plus rapide dans le cas d'une jointure entre une *petite* collection et une *grande* collection.
- On considère une jointure entre T1 et T2. On suppose que la taille des données de T1 est petite par rapport à T2 (T1 « T2) et peut tenir entièrement en mémoire sur chaque machine qui évalue la jointure.

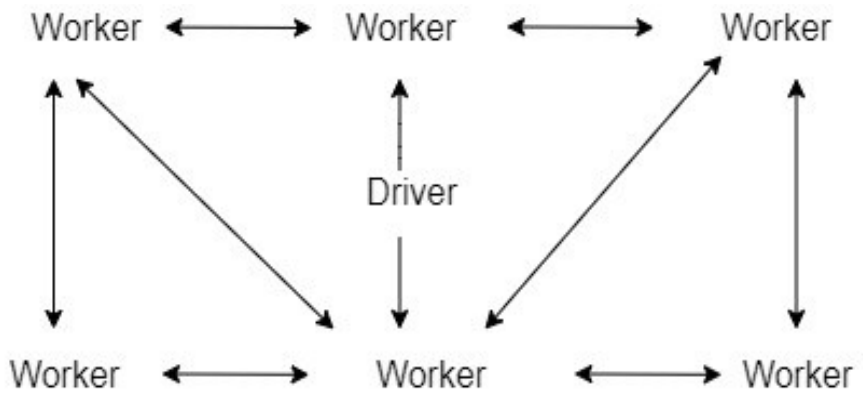
Illustration du broadcast join



https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

Broadcast join: diffusion de la petite relation

- R: large, S: Small
- $R \bowtie S$
- S doit tenir en mémoire dans le driver
- Diffusion aux worker nodes en cascade:
  - protocole similaire à bittorrent



Broadcast Join : exemple