BDLE

Traitement parallèle et distribué des jointures

Décembre 2021

Plan

- Intro
  - Objectifs, Contexte général
- Plan d’une requête
- Exécution distribuée
- Algos de jointures
- Produit cartésien

Objectifs

- Comprendre le traitement des jointures dans un environnement distribué (cluster de machines)
  - Partitionnement des données
  - Transfert (ou shuffle) de données
  - Traitement en parallèle
- Contexte big data :
  - jointure de très grandes tables
  - algorithmes de jointures qui passent à l’échelle
- Mise en pratique sur la plateforme Spark
  - Contrôler/diagnostiquer 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
    - si la taille des données > somme des tailles des mémoires
    - un disque n'est jamais plein si suffisamment de disques

Plan d'une requête

Une requête est transformée en un plan d'opérations

- Un plan est composé de plusieurs étapes

  plan=*Job* étape=*Stage*

- Une étape
  - traitement **local** d'une sous-requête
- Découpage du plan en étapes
  - Frontière: transfert/échange de données

    = **changement de la clé de répartition**
  - --> Précédence entre les étapes : plan = DAG d'étapes

Exécution d'une requête

Exécution du plan

- Plan = Job = *séquence* d'étapes
- Traiter une étape (*stage*)
  - traiter plusieurs tâches indépendantes en *parallèle*
- Une tâche est l'exécution d'une étape
  - Par un processeur
  - Sur une partition. L'exécution est *locale* à une partition
- Transfert des données entre 2 étapes
  - Transfert à la demande
  - Début d'une étape
    - Les tâches recoivent les données préparées par l'étape précédente
  - Fin d'une étape
    - Les tâches préparent les données pour l'étape suivante

Définir une jointure
Exemple

- Les utilisateurs :
  - User (prénom,ville)
  - (Alice, Paris) (Bob, Londres) .... (Zoé, Paris)

- Les notes attribuées à des films :
  - Note (prenom,titre ,note)
  - (Alice, StarWars, 5) (Bob, Matrix, 3) (Alice, Matrix, 4)

- Jointure entre Utilisateurs et Notes
  - J = User.join(Note, « prenom »)
  - (Alice, Paris, StarWars, 5) (Alice, Paris, Matrix, 3) (Bob, Londres, Matrix, 4)

Exécuter la jointure

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

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

Algorithmes de jointure parallèle

2 algorithmes pour évaluer une jointure parallèle :

- Jointure par hachage et partitionnement
  - Répartir les données par **hachage sur la clé de jointure**

- Jointure par diffusion (*broadcast*)
  - Répliquer et diffuser des données

Voir diapos suivantes pour le détail de ces 2 algos

Jointure parallèle par hachage et partitionnement

Parallel Partitioned join

Rappel du hash join *centralisé*

- Jointure entre User et Notes sur la clé prénom
- Pour User : créer une table de hachage T sur la clé de jointure
  - on a une fonction H, on crée un tableau T[]
    - Chaque case contient une liste de prénoms
  - Pour chaque u dans User
    - Ajouter u dans la liste contenue dans T [H(user.prenom)]

- Itérer sur Notes pour les associer avec un utilisateur
  - Pour chaque *n* dans Notes:
    - ListeU = T[H(n.prenom)]
    - Pour chaque u dans ListeU
      - Si u.prénom = n.prenom :
        - Produire (u.prenom, u.age, n.titre, n.note)

Idée : **DISTRIBUER T** : une « case » par machine, une « case » devient une **partition**

Approche généralisable à une T de taille quelconque : taille(T) > nb de machines

Jointure distribuée par hachage **Partitionner** la première table

- Principe : la table de hachage permet de partitionner les données
  - n°case de T = H(clé de jointure) = n° de partition
- Exemple centralisé
  - (Carol, Nice) (Alice, Paris) (Bob, NY) (Will, Aix) (Zoé, Paris) (Tim, NY) devient
    - T[1] = (Alice, Paris) (Zoé, Paris) (Tim, NY)
    - T[2] = (Carol, Nice) (Bob, Londres) (Will, Aix)
- Exemple distribué : une table pour chaque partition initiale
  - Partition 1 : (Carol, Nice) (Alice, Paris) (Bob, NY) devient
    - T1[1] = (Alice, Paris)
    - T1[2] = = (Carol, Nice) (Bob, NY)
  - Partition 2 : (Will, Aix), (Zoé, Paris) (Tim, NY) devient
    - T2[1] = (Zoé, Paris) (Tim, NY)
    - T2[2] = (Will, Aix)

Prénom	**H(prénom)**
Carol	2
Alice	1
Bob	2
Zoe	1
Tim	1
Will	2

Jointure distribuée par hachage **Partitionner** la deuxième table

Traitement identique pour la **2ème table** à joindre

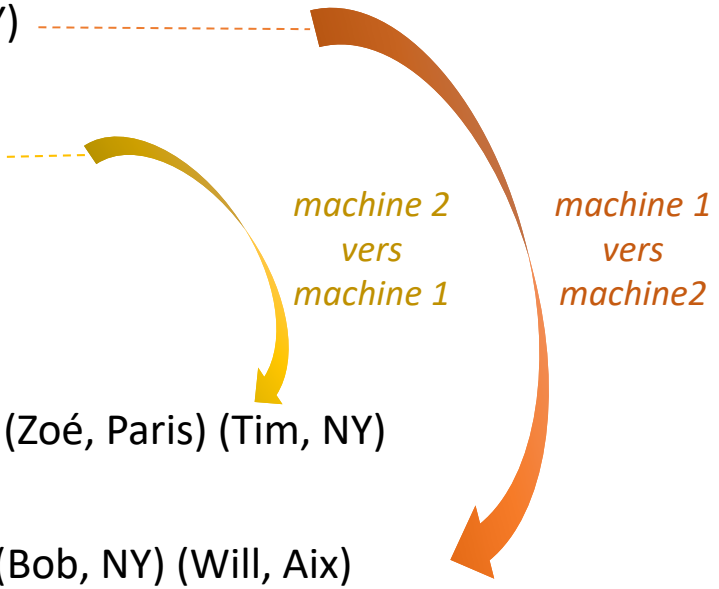
partitionner les Notes en fonction de H(prénom) = n° de partition

- Partition 1 : (Will, Dune, 5) (Alice, StarWars, 5) (Tim, Dune, 1) (Carol, Dune,3) (Alice, Dune, 3) organisée en
  - T1[1] = (Alice, StarWars, 5) (Tim, Dune, 1) (Alice, Dune, 3)
  - T1[2] = (Will, Dune, 5)
- Partition 2 : (Bob, StarWars, 5), (Alice, SpiderMan, 1) (Tim, SpiderMan, 2) organisée en
  - T2[1] = (Alice, SpiderMan, 1) (Tim, SpiderMan, 2)
  - T2[2] = (Bob, StarWars, 5)

Prénom	H(prénom)
Carol	2
Alice	1
Bob	2
Zoe	1
Tim	1
Will	2

Jointure distribuée par hachage **Shuffle** de la première table

- **SHUFFLE** = répartitioner les données à partir du n° de partition
- Avant le shuffle:
  - Partition 1 :
    - T1[1] = (Alice, Paris)
    - T1[2] = (Carol, Nice) (Bob, NY)
  - Partition 2 :
    - T2[1] = (Zoé, Paris) (Tim, NY)
    - T2[2] = (Will, Aix)
- Après le shuffle:
  - **Partition 1 :**
    - T1[1] U T2[1] = (Alice, Paris) (Zoé, Paris) (Tim, NY)
  - **Partition 2 :**
    - T1[2] U T2[2] = (Carol, Nice) (Bob, NY) (Will, Aix)



Jointure distribuée par hachage **Shuffle** de la 2ème table

Avant le shuffle

- Partition 1 :
  - T1[**1**] = (Alice, StarWars, 5) (Tim, Dune, 1) (Alice, Dune, 3)
  - T1[**2**] = (Will, Dune, 5)
- Partition 2
  - T2[**1**] = (Alice, SpiderMan, 1) (Tim, SpiderMan, 2)
  - T2[**2**] = (Bob, StarWars, 5)

Après le shuffle

- **Partition 1 :**
  - T1[**1**] U T2[**1**] = (Alice, StarWars, 5) (Tim, Dune, 1) (Alice, Dune, 3) (Alice, SpiderMan, 1) (Tim, SpiderMan, 2)
- **Partition 2 :**
  - T1[**2**] U T2[**2**] =(Will, Dune, 5) (Bob, StarWars, 5)

Jointure distribuée par hachage
Jointure parallèle

La machine n°k contient les partitions n°k de **toutes** les tables

- Machine **1**:
  - User1 :
    - (Alice, Paris) (Zoé, Paris) (Tim, NY)
  - Notes1:
    - (Alice, StarWars, 5) (Tim, Dune, 1) (Alice, Dune, 3) (Alice, SpiderMan, 1) (Tim, SpiderMan, 2)
- Machine 2
  - User2 :
    - (Carol, Nice) (Bob, NY) (Will, Aix)
  - Notes2 :
    - (Will, Dune, 5) (Bob, StarWars, 5)

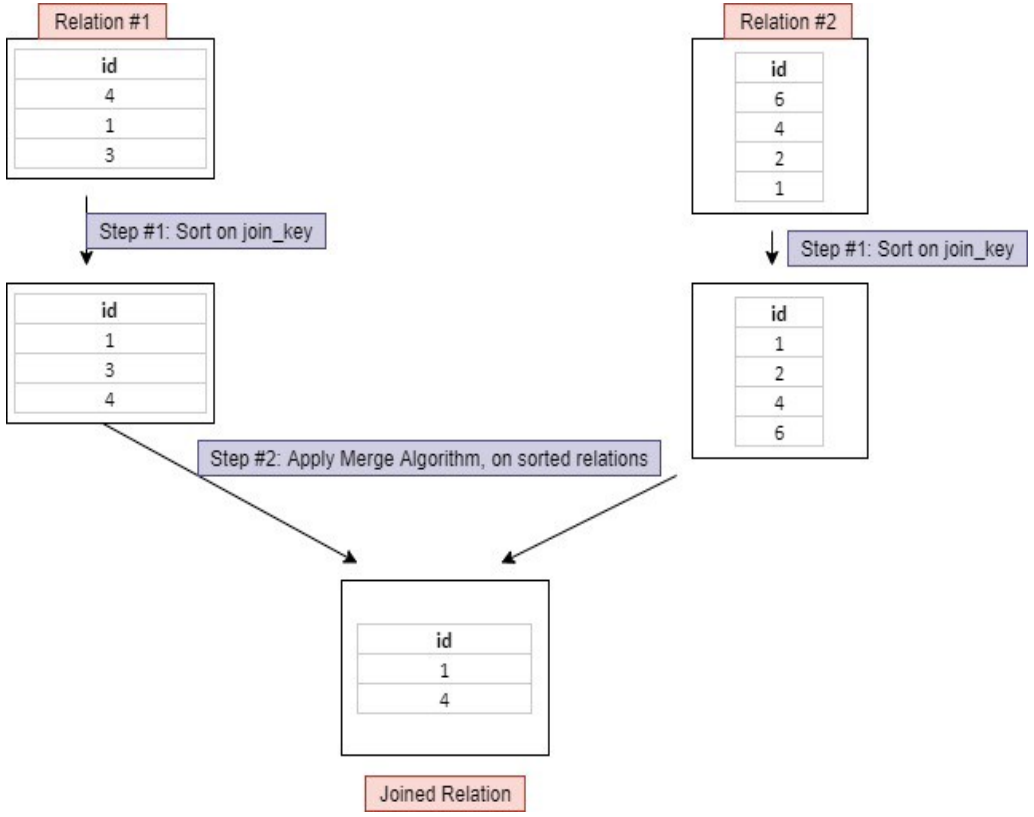
- Jointure **indépendante** sur chaque machine : parallélisme!

Quel algo de jointure peut-on utiliser **localement** sur une machine ?

Jointure distribuée par hachage
Zoom sur la jointure locale

- Tout algorithme de jointure peut être utilisé localement
  - Tri fusion, hachage, nested loops, …

Rappel du tri fusion



Jointure distribuée par hachage
TRI

Trier chaque partition

- Tri sur partition **1**:
  - User1
    - (Alice, Paris) (Tim, NY) (Zoé, Paris)
  - Notes1:
    - (Alice, StarWars, 5) (Alice, Dune, 3) (Alice, SpiderMan, 1) (Tim, Dune, 1) (Tim, SpiderMan, 2)
- Tri sur partition 2
  - User2 :
    - (Bob, NY) (Carol, Nice) (Will, Aix)
  - Notes2 :
    - (Bob, StarWars, 5) (Will, Dune, 5)

Jointure distribuée par hachage
FUSION

Fusion dans chaque partition

- Fusion sur partition **1**:

  (Alice, Paris, StarWars, 5) (Alice, Paris, Dune, 3) (Alice, Paris, SpiderMan, 1)

  (Tim, NY, Dune, 1) (Tim, NY, SpiderMan, 2)

- Fusion sur partition 2

  (Bob, NY, StarWars, 5) (Will, Aix, Dune, 5)

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 de répartition des données par hachage
  - Ecrire T sur disque si elle ne tient pas en mémoire (spill)
  - Tri les données de chaque partition avant de les transférer
    - Permet de fusionner les partitions créées sur une même machine (cas rare)
- Etape de jointure
  - Petite quantité de mémoire nécessaire
    - Pour toute clé de jointure K, seul l'ensemble des paires (K, V) venant des 2 relations doit tenir en mémoire.
    - Pas besoin qu'une partition 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.

Broadcast join: diffusion de la petite relation

- Taille de Notes > taille de User
- $J = Notes \bowtie User$
- Chaque machine K envoie sa partition de $User_K$ à **TOUTES** les autres machines
- Diffusion «indirecte» par le driver
  - Le driver récupère la table et la diffuse aux worker nodes en cascade
  - Inconvénient: User doit tenir en mémoire dans le driver
- Diffusion directe par shuffle ?
  - Initialement, chaque machine a une partition de Note
    - Partition n°K sur la machine K
  - Chaque machine réplique sa partition
    - Sur machine K on crée T[1] = T[2] = … = Partition n°K
    - La machine K récupère les T[K] des autres machines

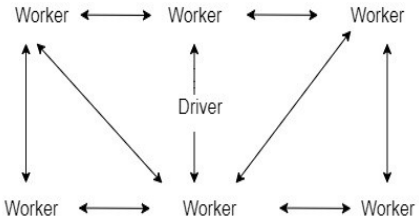
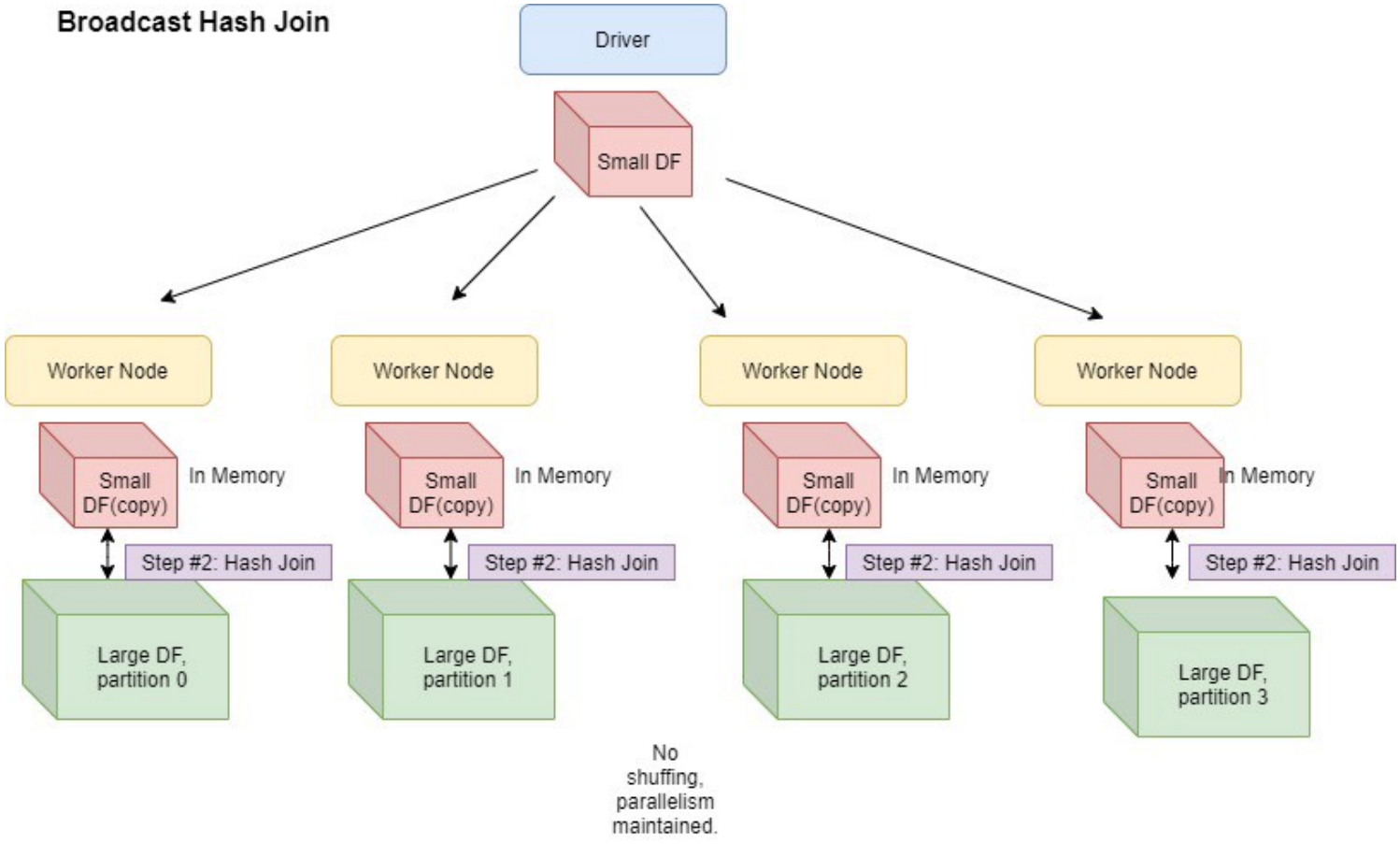
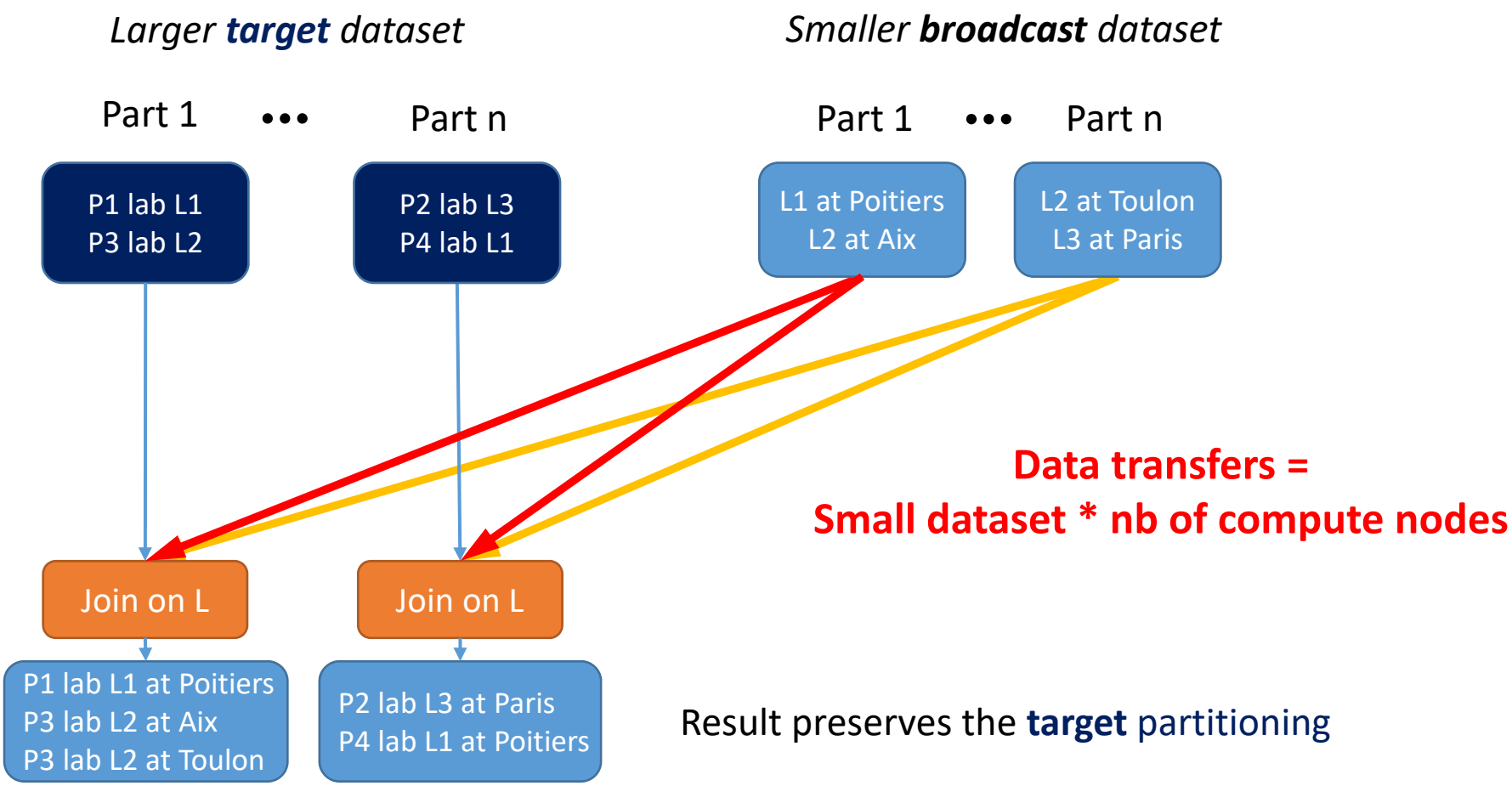


Illustration de la diffusion indirecte par de driver



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

Broadcast Join : exemple



A faire en TME

- Proposer une solution pour calculer la jointure
  - J(nF, titre, nU, note) = $\pi_{nF, titre, nU, note}$(Films⋈ Notes)
  - Implémenter vous-même la distribution des données à l'aide des méthodes mapPartitionsWithIndex et repartition.
  - L'invocation de la méthode join doit se limiter à associer des partitions ayant le même numéro *numP*.
- Exemple 1
  - Manipuler un extrait des notes pendant la mise au point de votre solution
    - Notes1K = Notes.sample(0.001).persist()
  - F1(numP, Liste(nF, titre)) = Films.mapPartitionsWithIndex(decouperFilm)
    - showPartitions(F1)
  - N1(numP, Liste(nU, nF, note)) = Notes1K.mapPartitionsWithIndex(decouperNotes)
    - showPartitions(N1)
  - F2 = F1.repartition(4, "numP")
    - showPartitions(F2)
  - N2 = N1.repartition(4, "numP")
    - showPartitions(N2)
  - J1 = F2.join(N2, "numP")
    - showPartitions(J1)
  - J2 = J1.mapPartitionsWithIndex(jointureLocale)
    - showPartitions(J2)
  - Vérifier que le contenu de J2 est identique à celui de J
- Exemple 2 : idem mais pour la jointure par diffusion

Bilan et Conclusion

- Jointure parallèle et distribuée
  - 2 algo détaillés
- Avantages/Limitations
  - Jointure parallèle par hachage
    - Passe à l'échelle pour joindre 2 grandes tables
    - Sensible au déséquilibre des données
  - Jointure par diffusion
    - Evite de distribuer la plus grande table
    - Robuste au déséquilibre des données
- Perspectives : optimisation des requêtes avec plusieurs jointures
  - Peut d'information sur les données au préalable
  - Optimisation dynamique par adapatation du plan d'exécution:
    - Traiter la requête sur 1% des données
    - Obtenir la sélectivité des opérations : modifier l'ordre des jointures ou les algos hachage/diffusion utilisés
    - Traiter sur 2% des données, puis 5%, 10%, 100% (taille de échantillons?, nombre d'échantillons ?)