Master d'Informatique

SAM - 4I803 - Cours 10

Bases de données parallèles

Plan

Introduction

Architectures

Placement des données

Parallélisme dans les requêtes

Optimisation de requêtes

Introduction

- Une machine parallèle contient
  - plusieurs CPU (multicores), mémoire, disque.
- Taille croissante des données à gérer
  - ex. IOT (Linky), index du web (Qwant)

- Principe des SGBD parallèles :
  - interconnecter plusieurs machines pour exploiter le parallélisme dans la gestion de données
  - notions de systèmes distribués

Exemples de cas d'usage

- Parallélisme pour les transactions
  - benchmark TPC-C
- Parallélisme pour les requêtes
  - Tri : SortBenchmark
    - Lire des rapports d'expérimentation
      - 2016: http://sortbenchmark.org/TencentSort2016.pdf
      - 2014: http://sortbenchmark.org/ApacheSpark2014.pdf
  - Jointures et SQL : Spark
  - Group By : solution Indexima
- Parallélisme pour les données
  - Stockage massif (sky server)

Parallélisme dans les BD

- Parallélisme des données
  - Partitionner et stocker les données sur plusieurs disques
    - Lecture/écriture en parallèle
- Parallélisme des traitements
  - À tout instant : n processeurs accèdent à n partitions
  - Traiter des opérations relationnelles en parallèle
  - Traiter des requêtes en parallèle
- Parallélisme transparent pour l'utilisateur
  - Requêtes exprimées en SQL (haut niveau)
  - Traduction en opérateurs algébriques
  - Génération d'un plan d'exécution parallèle

Objectifs des BD parallèles

Contrairement aux BD réparties, on ne s'intéresse pas à la localité par rapport à l'émetteur de la requête

- Améliorer les performances (temps de réponse)
- Améliorer la disponibilité (réplication)
- Approche cluster : réduire les coûts
  - À capacité égale, plusieurs petites machines sont moins chères qu'un seul super calculateur.
- Approche cloud : datacenter = n clusters « pay on use »

Architectures de BD parallèles

- On a 3 types de ressources
  - P = Processeur
  - M = Mémoire
  - D = Disque
- Plusieurs unités de chaque type
- Comment organiser les accès entre ces unités ?
  - Un processeur peut-il accéder à toutes les unités de mémoires et/ou à tous les disques ?

Architectures de BD parallèles

3 façons d'associer les ressources P, M, D:

- mémoire partagée (*shared nothing)*

  P ← n → M ← n → D

- disque partagé *(shared disk)*

  P ← 1 → M ← n → D

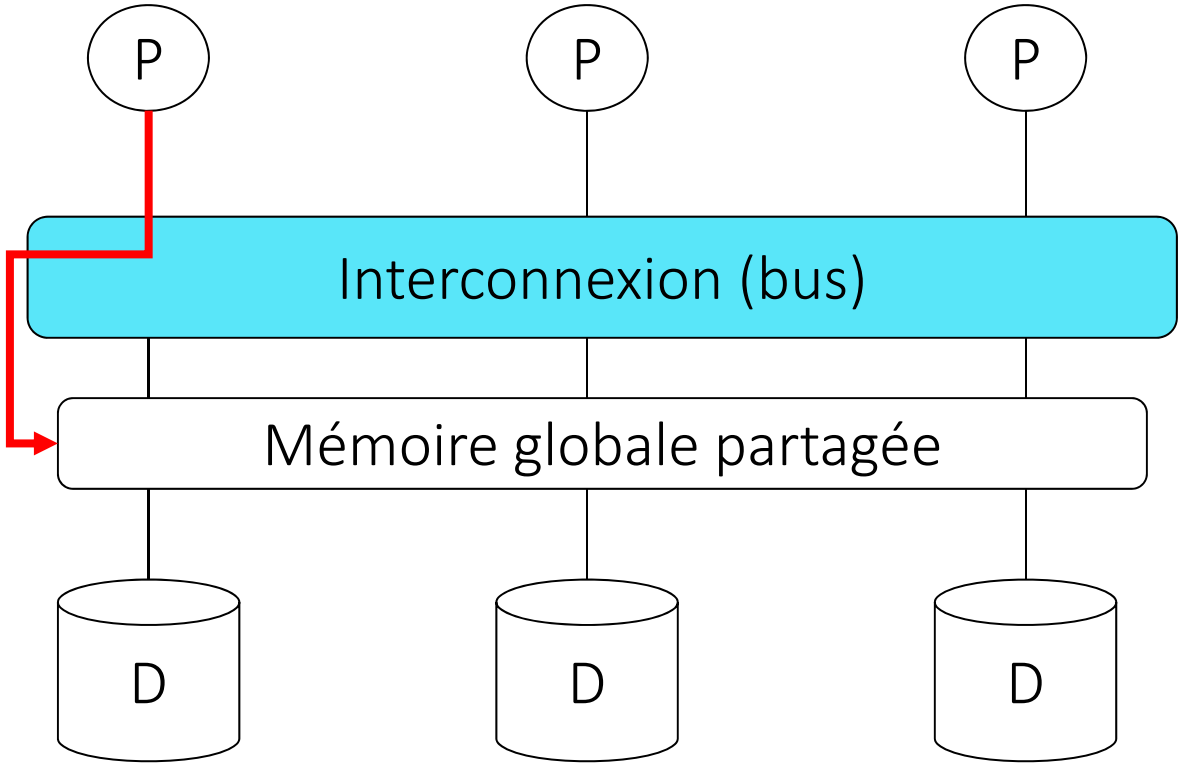
- aucun partage *(shared nothing)*

  P ← 1 → M ← 1 → D

Moins de partage améliore l'extensibilité...

Mémoires partagées

- Tous les processeurs partagent le disque et la mémoire



Mémoires partagées

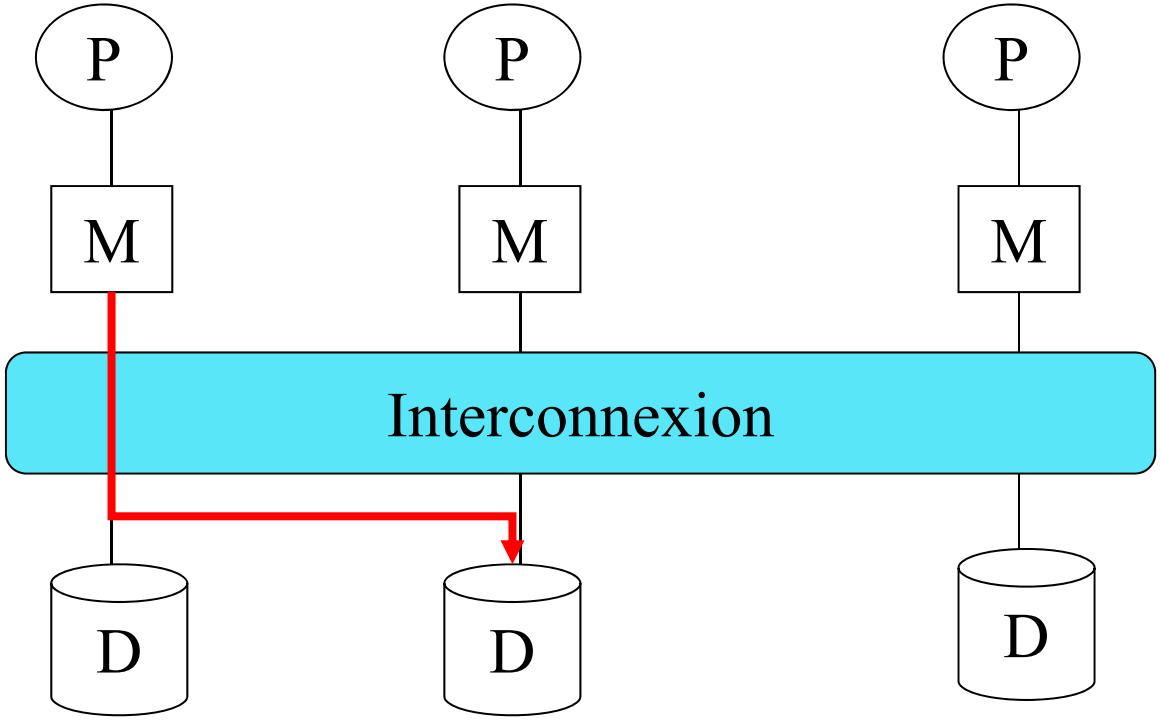
+ simple : une seule mémoire composée de plusieurs M
+ équilibrage de charge : tout P peut traiter toute requête
+ parallélisme inter-requête (ajout de processeur) direct, intra-requête possible (assez simple)
- Peu extensible : moins de 20 P à cause des accès concurrents à M et des contraintes matérielles
- Lent si conflits d'accès aux M
- Cher : chaque processeur doit être lié à tous les M
- Indisponibilité des données si M défaillante (affecte plusieurs P)

Exemple: Serveur HPE Superdome Flex

Utilisé dans des SGBD commerciaux : Oracle, IBM DB2

Disques partagés

- Chaque noeud (P,M) accède à tous les disques D
  - les M ne sont pas partagées



Disques partagés

+ interconnexion moins complexe qu'en mémoires partagées
+ Bonne disponibilité (défaut de mémoire local à un processeur)
+ Migration facile de BD existantes

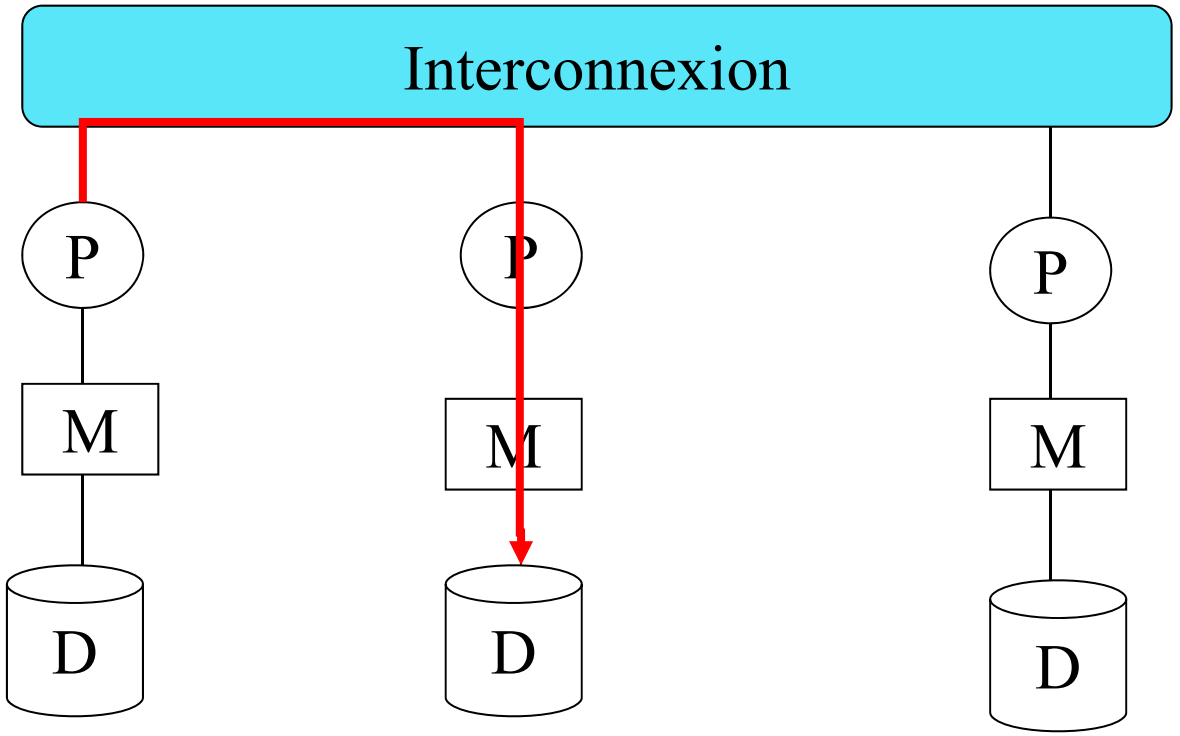
  Pas de réorganisation des données sur le disque

- Assez complexe
- Pbs de performance
  - Eviter les accès conflictuels aux mêmes pages
    - Verrouillage peut être bloquant
  - Maintenir la cohérence des copies
  - Accès aux D en parallèle mais requête traitée localement par 1 seul P

- Utilisé dans IMS/VS(IBM), VAX (DEC), Microsoft **Socrate** newSQL

Aucun partage

- Les différents noeuds sont reliés par un réseau d'interconnexion. Chaque processeur a un accès exclusif à la mémoire et au disque (idem BD réparties).



Aucun partage

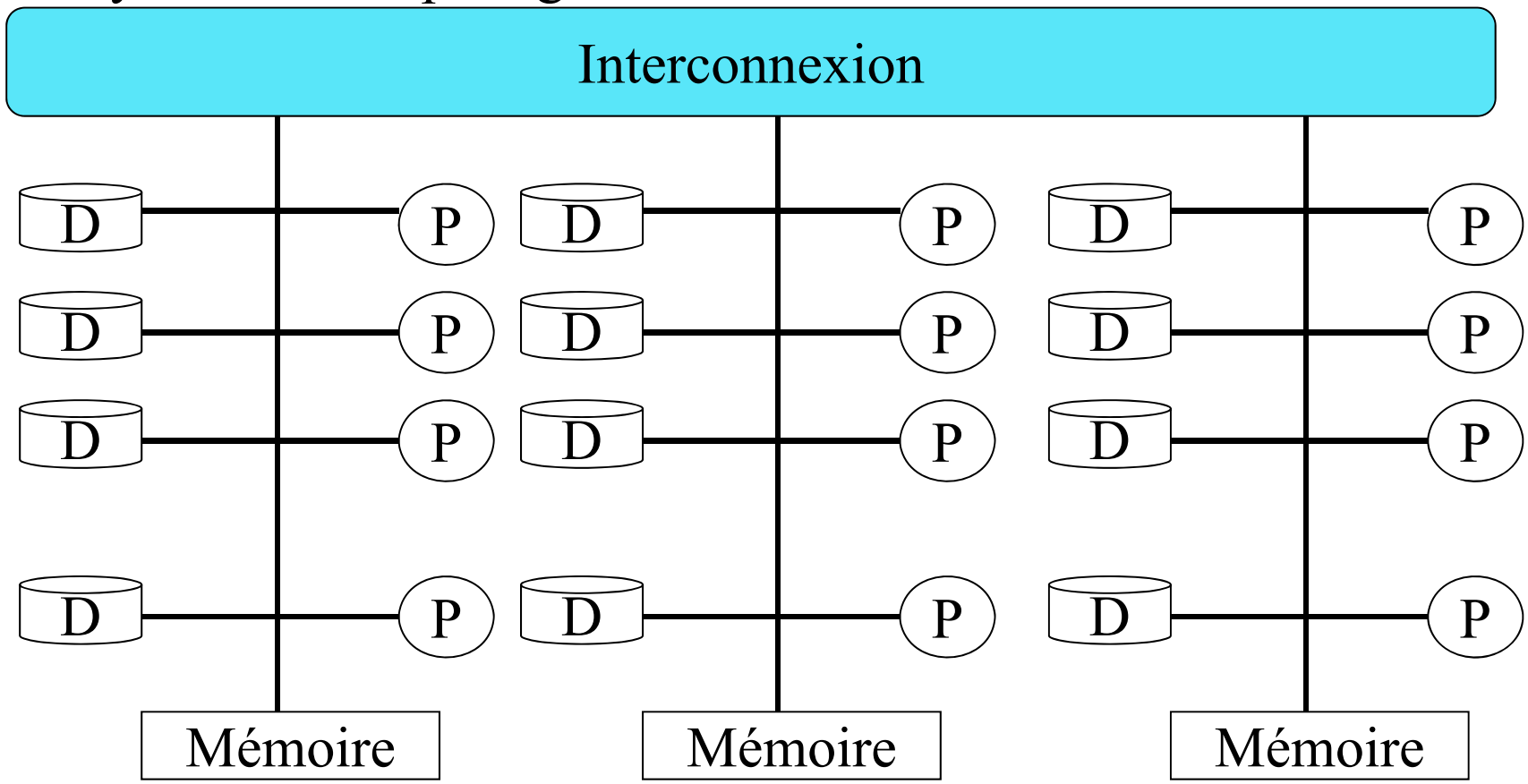
+ Bonne extensibilité (milliers de noeuds)
+ Disponibilité par réplication des données
+ Réutilisation des technique de **BD réparties**
+ Bien adapté au parallélisme intra-requête

- Plus complexe que les systèmes à mémoire partagée (mêmes fct. qu'en BD réparties)
- Répartition de charge difficile à mettre en oeuvre, à cause de la répartition statique des données (le proc. n'accède qu'à son disque)
- Ajouter des noeuds entraîne une réorganisation des données pour permettre la répartition de charge

• Utilisé dans Teradata DBC, EDS, **Clustrix/MariaDB**, etc.

Systèmes hybrides

- Combinaison de systèmes à mémoire partagée et de systèmes sans partage.



Systèmes hybrides

- Chaque noeud individuel est un système à mémoire partagée.

+ Combine la souplesse et les performances des systèmes à mémoire partagée avec les capacités d'extensibilité des systèmes sans partage.

+ La communication au sein de chaque noeud est efficace (mémoire partagée).

+ Bon compromis entre répartition de charge et passage à l'échelle.

Traitement en parallèle

Quel traitement **attribuer** à chaque processeur ?

Plusieurs méthodes possibles :

- Inter requêtes : 1 requête → 1 processeur
- Intra requête : 1 requête → N processeurs
  - Inter opérateurs : 1 opérateur → 1 processeur
    - Pipeline
    - Matérialisation
  - Intra opérateur : 1 opérateur → N processeurs

Parallélisme inter-requêtes

- Forme la plus simple du parallélisme
- Les requêtes (et les transactions) s'exécutent en parallèle
- Augmente le débit de transactions (trans par minute TPM)
- Plus difficile à implémenter sur les architectures disque partagé et sans partage
  - La coordination des verrouillages et des journaux s'effectue par envois de messages entre processeurs
  - Les données d'un buffer local peuvent avoir été mises à jour sur un autre processeur
  - Nécessité de maintenir la cohérence des caches (les lectures et écritures dans le buffer doivent concerner la version la plus récente.)

Parallélisme intra-requêtes

- Exécuter une seule requête en parallèle sur plusieurs processeurs

  important pour les requêtes « longues »
- Une requête = plusieurs opérateurs
  - **Inter-opérateurs** : exécuter plusieurs opérations en parallèle.
  - **Intra-opérateurs** : paralléliser l'exécution de chaque opération dans la requête. Intéressant si beaucoup de données pour une seule opération.

Parallélisme inter-opérateurs

- Deux formes :
  - **Pipeline** : plusieurs opérateurs successifs sont exécutés en parallèle, le résultat intermédiaire n'est pas matérialisé. Gain de place mémoire et minimise les accès disque. Risque d'attente des résultats de l'opé. précédent.
  - **Parallélisme indépendant** : les opérateurs sont indépendants, pas d'interférence entre les processeurs. Le résultat est matérialisé = coût

Parallélisme intra-opérateurs

- Décomposer un opérateur en un ensemble de sous-opérateurs, chacun s'exécutant sur une partition de la relation.

Exemple pour la sélection :

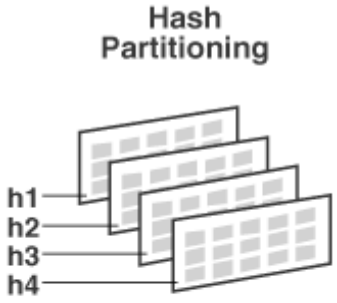
$$\sigma_S (R) \equiv \sigma_{S1} (R_1) \cup \sigma_{S2}(R_2) \cup \ldots \cup \sigma_{Sn}(R_n)$$

Si S porte sur l'attribut de partitionnement, on peut éliminer certains Si

- Dépend de la manière dont on **partitionne** les données
  - Cf diapos suivantes

Partitionnement des données (1/2)

- Répartir les données sur les disques permet de réduire le temps d'accès aux données par parallélisme (pas par localité, car utilisateur pas affecté à nœud)
- Partitionnement horizontal : les n-uplets d'une relation sont répartis sur les différents disques (pas de partitionnement de n-uplet).
- Différentes techniques de partitionnement (nb de disques = n) :
  - **Round-robin** : le $i^{ème}$ n-uplet inséré dans la relation est stocké sur le disque *i mod n*
  - **Partitionnement par hachage (hash partitioning) :**
    - Choisir un ou plusieurs attributs comme attributs de partitionnement
    - Appliquer une fonction de hachage (renvoyant un entier de 0 à n-1) à ces attributs
    - Stocker le n-uplet sur le disque correspondant au résultat de la fonction de hachage

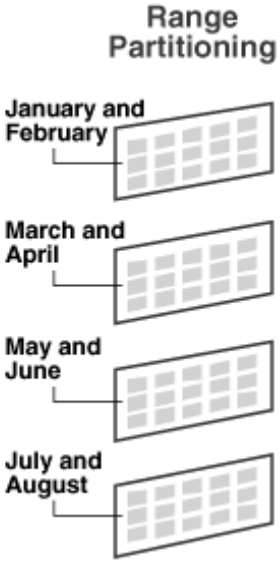


Partitionnement des données (2/2)

- **Partitionnement par intervalles (range partitionning)** : répartit les n-uplets en fonction des intervalles de valeurs d'un attribut
  - Choisir un attribut pour le partitionnement

    Choisir un vecteur de partitionnement $[v_0, v_1, \ldots, v_{n-2}]$

    Soit v la valeur de l'attribut de partitionnement.
    - Les n-uplets tq $v<v_0$ vont sur le disque 0
    - Les n-uplets tq $v \geq v_{n-2}$ vont sur le disque n-1
    - Les n-uplets tq $v_i < v \leq v_{i+1}$ vont sur le disque i+1



Sélection de données partitionnées (1/3)

- Sélection
  - Parcours de la relation (scan)
  - Sélection sur égalité (ex: R.A=15)
  - Sélection sur intervalles (ex: $10 \leq R.A < 25$)
- **Partitionnement Round-robin** :
  - Convient bien au parcours séquentiel de relations
  - Bonne répartition des n-uplets sur les disques
    - bonne répartition de charge
  - Mal adapté aux requêtes avec sélection
    - pas de regroupement, les données sont dispersées

Sélection de données partitionnées (2/3)

Partitionnement par hachage

- Efficace pour les accès séquentiels
  - Si on a une bonne fonction de hachage, et si le(s) attribut(s) de partitionnement est (sont) clé, il y a bonne répartition des n-uplets sur les disques, et une bonne répartition de la charge pour rechercher les données
- Efficace pour les sélections par égalité sur l'attribut de partitionnement
  - La recherche peut se limiter à un seul disque
  - Possibilité d'avoir un index local sur l'attribut de partitionnement
- Mal adapté aux sélections sur intervalles, car il n'y a pas de regroupement.

Sélection de données partitionnées (3/3)

Partitionnement par intervalle

- Les données sont regroupées par valeurs de l'attribut de partitionnement
- Efficace pour les accès séquentiels
- Efficace pour les sélections par égalité sur l'attribut de partitionnement
- Bien adapté aux sélections sur intervalles (seul un nombre limité de disques est concerné)
  - Efficace si les n-uplets du résultat se trouvent sur peu de disques
- Risque de répartition inégale sur les disques
  - biais sur les valeurs ou *data skew*

Partitionnement d'une relation

Compromis entre 2 objectifs de performance:

- Minimiser les transferts
  - Réduire de nombre de disques
    - Nb partitions = taille relation /taille disque
- Maximiser le parallélisme
    - Nb partitions = Nbre disques

Rmq : La taille d'une partition est au moins une page

- avec page(relation) = p et Nbre disques = n
- Nb partitions = *min(n, p)*

Gestion des répartitions non uniformes

- Certains algorithmes de répartition risquent de donner lieu à une répartition inégale des données (bcp sur un disque, peu sur un autre), ce qui peut dégrader les performances:
  - Mauvais choix de vecteur de partition
  - Mauvais choix de fonction de hachage
  - Répartition inégale des valeurs d'un attribut de partitionnement
- Solutions (partitionnement par intervalle) :
  - Créer un vecteur de partitionnement équilibré (trier la relation sur l'attribut de partitionnement et diviser en sous-ensembles de même taille)
  - Utiliser des histogrammes
  - Mais maintenance coûteuse si le biais des données change

Traitement parallèle des opérateurs relationnels

- Suppositions :
  - Requêtes en lecture seule
  - Architecture sans partage
  - n processeurs, et n disques (D_1 est associé à P_1)
- Les architectures sans partage peuvent être simulées efficacement sur des architectures à mémoire partagées ou à disques partagés. Les algorithmes peuvent être appliqués, avec différentes optimisations, sur ces architectures.

Tri parallèle par intervalle

Appelé *Range-partitioning sort*

- **On impose d'allouer m processeurs pour faire le tri (**$m \leq n$)
- Choisir les processeurs $P_0 \ldots P_m$, avec $m \leq n$,
- Définir un partitionnement du domaine de l'attribut de tri
  - vecteur avec m entrées
- Redistribuer la relation selon ce partitionnement
  - tous les n-uplets du $i^{ème}$ intervalle sont envoyés au processeur P_i
  - P_i stocke temporairement les n-uplets sur son disque D_i
- Chaque processeur trie sa partition localement
  - en parallèle, indépendamment des autres processeurs
- Concaténer les différents résultats
  - pas besoin de fusion car les partitions sont déjà ordonnées).

Nom	Points
Bob	80
Alice	100
Carl	9
Jim	300
Zoe	7
Noe	40

Redistribution

[0, 10[

Nom	Points
Carl	9
Zoe	7

P1

Zoe	7
Carl	9

[10, 100[

Nom	Points
Bob	80
Noe	40

P2

Noe	40
Bob	80

[100, 1000[

Nom	Points
Alice	100
Jim	300

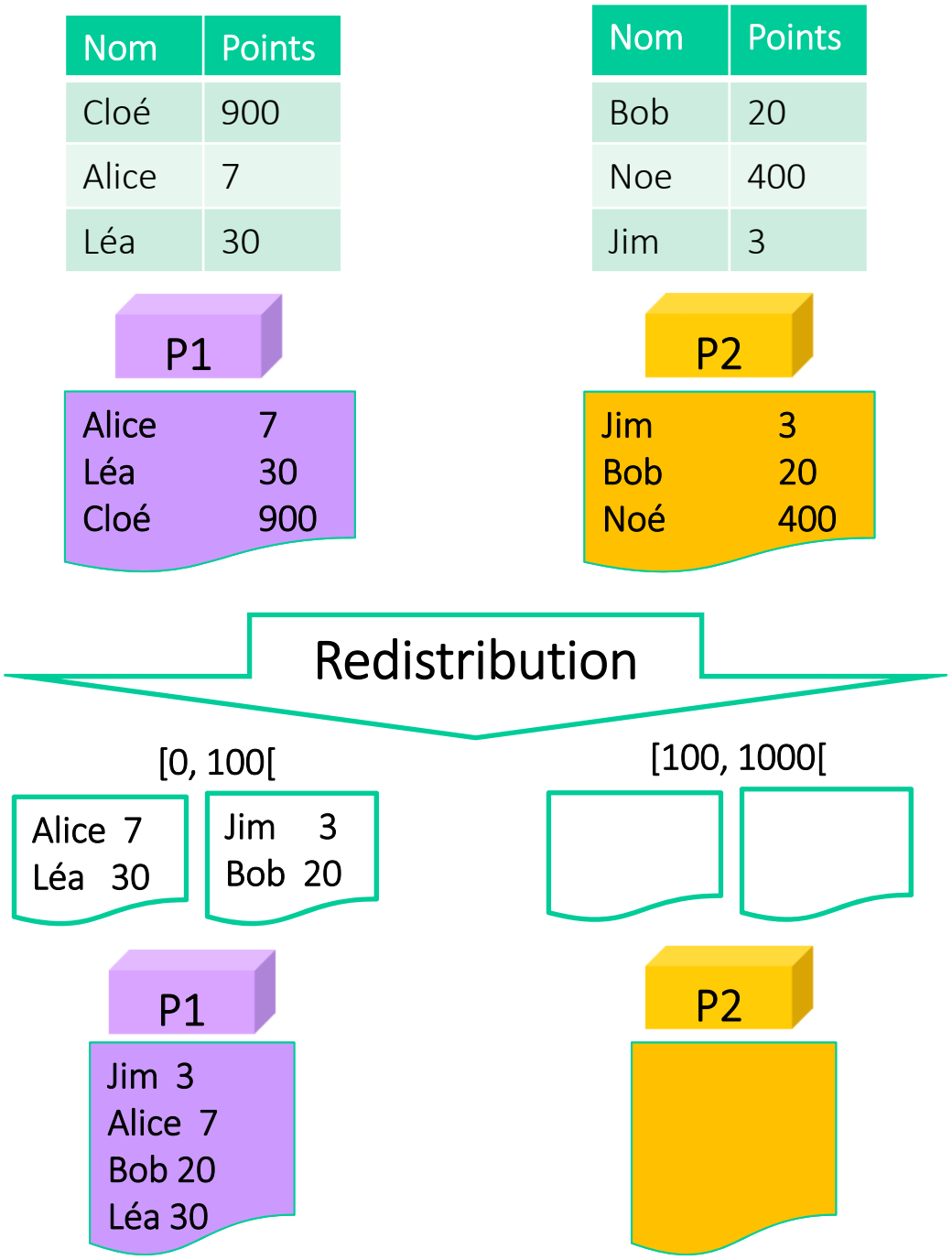
P3

Alice	100
Jim	300

Tri parallèle avec round robin (RR)

Trifusion parallèle externe :

- Tous les processeurs contenant une partition de données font le tri
- La relation est partitionnée sur les *n* disques (RR).
- Chaque processeur trie localement ses données.
- La fusion est parallélisée :
  - Les partitions triées sur chaque processeur sont partitionnées par intervalle puis distribuées sur les différents processeurs
  - Chaque processeur fusionne les données qu'il reçoit à la volée.
  - Les différents résultats sont concaténés


Nom
Points
Cloé
900
Alice
7
Léa
30
Nom
Points
Bob
20
Noe
400
Jim
3
P1
Alice 7
Léa 30
Cloé 900
P2
Jim 3
Bob 20
Noé 400
Redistribution
[0, 100[
[100, 1000[
Alice 7
Léa 30
Jim 3
Bob 20
P1
Jim 3
Alice 7
Bob 20
Léa 30
P2

Jointure parallèle

- Principes :
  - La jointure consiste à tester les n-uplets deux par deux et à comparer la valeur des attributs de jointure.
  - Le parallélisme consiste à répartir ces tests sur des processeurs différents, chacun calculant une partie de la jointure localement.
  - Les résultats sont ensuite rassemblés.

Boucles imbriquées en parallèle

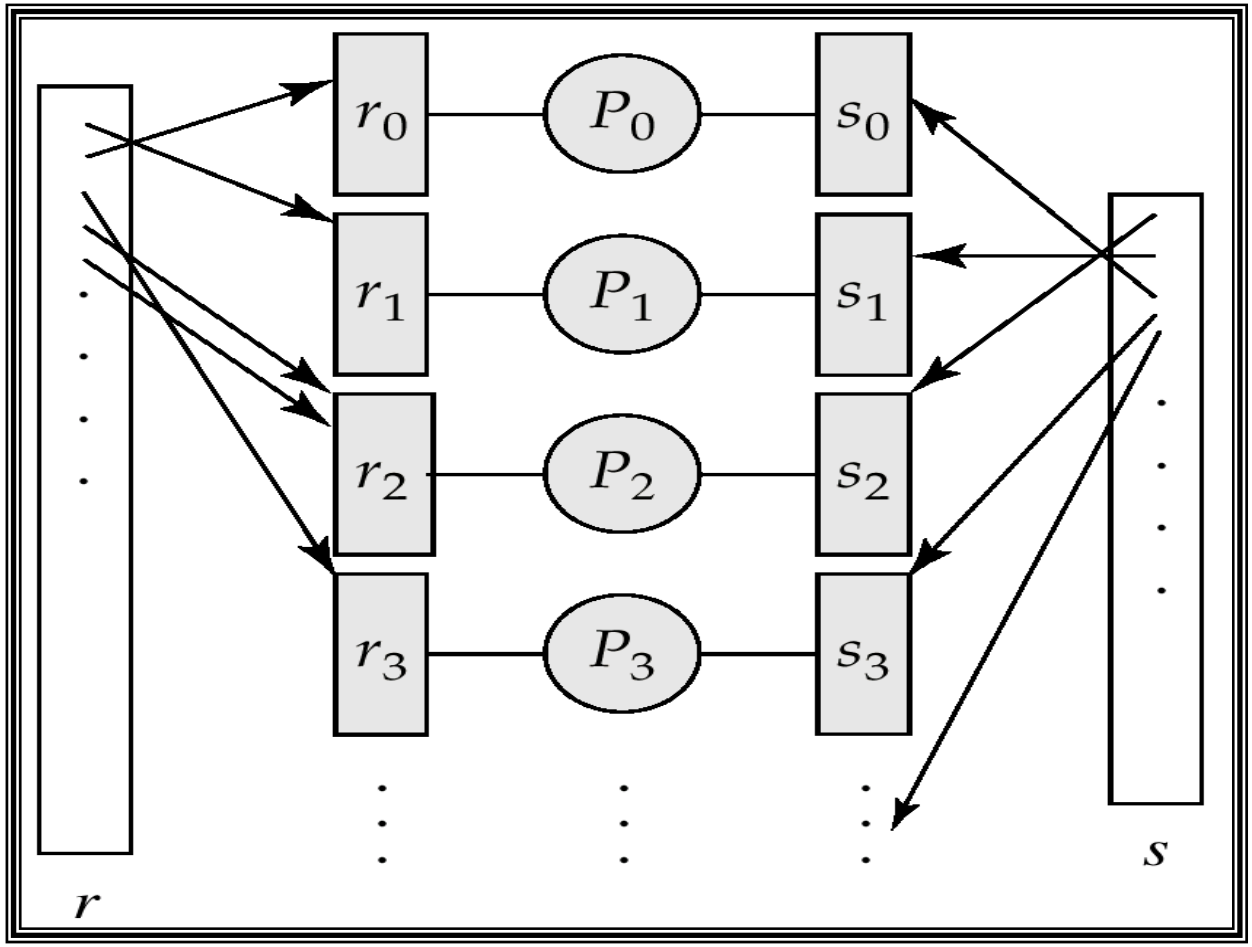
- Les relations R et S sont fragmentées en m et n fragments respectivement.
- Envoyer (en parallèle) les m fragments de R sur les n nœuds où se trouvent des fragments de S.
  - Chaque nœud reçoit R en entier
- Faire la jointure sur ces nœuds (en parallèle).
- Concaténer les résultats.

Jointure par partitionnement

- Equi-jointure, on peut partitionner la jointure en jointures indépendantes.
  - Cf cours requêtes réparties et propriétés du partitionnement.

- Partitionner (attention, coûteux) les deux relations (par hachage ou par intervalle) en n partitions, sur l'attribut de jointure, en utilisant la même fonction de hachage ou le même vecteur de partitionnement.
- Les partitions r_i et s_i sont envoyées au processeur P_i
- Chaque processeur calcule la jointure entre r_i et s_i (avec n'importe quel algorithme)
- Les résultats sont concaténés.
- Attention : difficile de contrôler la taille de r_i et s_i. Equilibrage de charge difficile

Jointure par partionnement



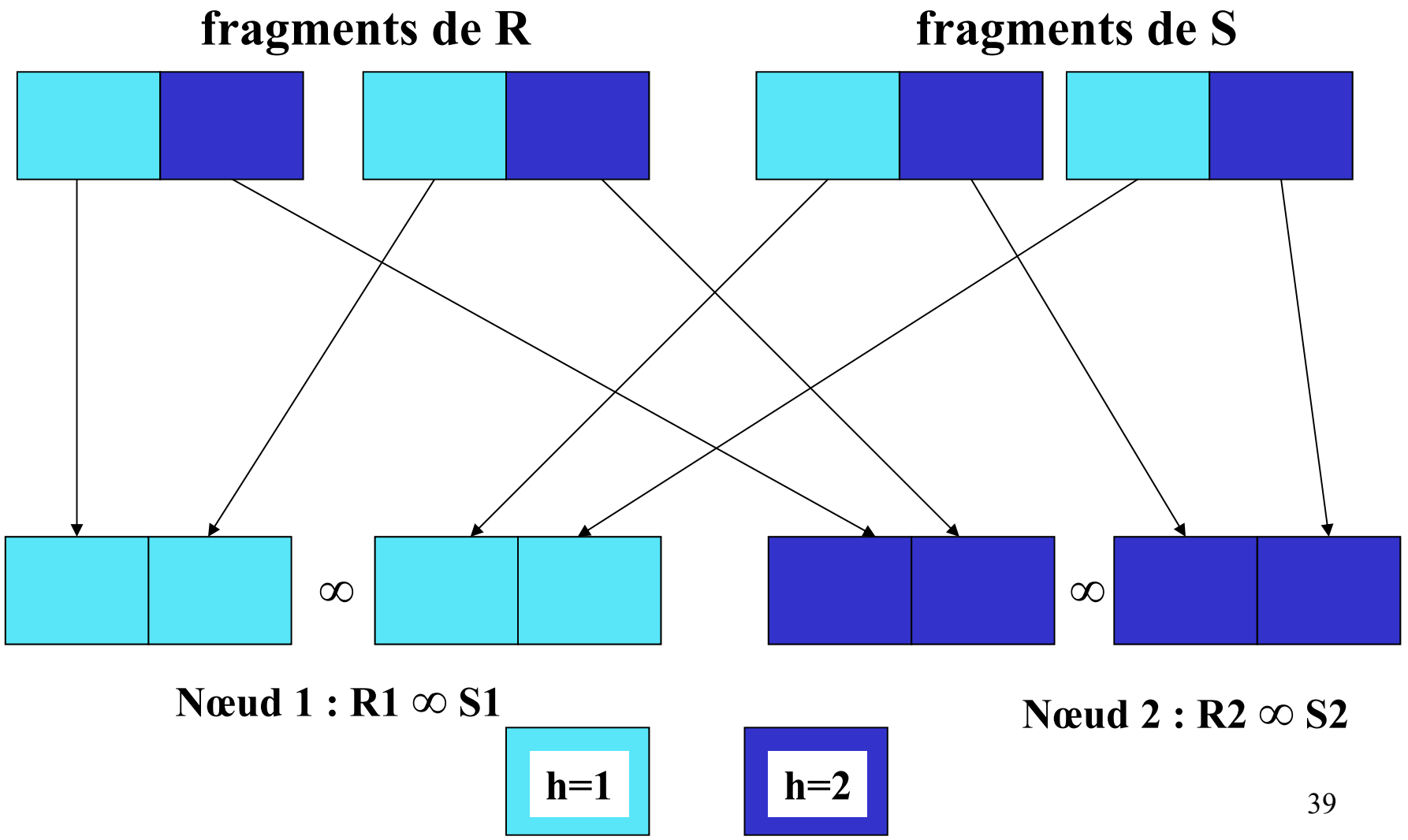
Jointure parallèle par hachage

- Partitionner (en parallèle) les relations R et S en k partitions à l'aide d'une fonction de hachage h appliquée à l'attribut de jointure tq

$$R \infty S = U_{(i=1..k)} (Ri \infty Si)$$

- Les partitions de R et de S ayant même valeur de hachage sont envoyées sur le même processeur.
- Faire les jointures sur chaque processeur en parallèle (k jointures en parallèle)
- Concaténer les résultats

Jointure parallèle par hachage



Optimisation de requêtes

- Semblable à l'optimisation de requêtes dans les BD réparties.
- Espace de recherche
  - Les arbres algébriques sont annotés pour permettre de déterminer les opérations pouvant être exécutées à la volée (pipeline). Génération d'arbres linéaires gauche, droite, en zig-zag, touffus.
- Modèle de coût
  - Dépend en partie de l'architecture
- Stratégie de recherche
  - Les mêmes qu'en environnement centralisé (les stratégies aléatoires sont mieux adaptées en raison de la taille de l'espace de recherche, plus grand car on considère plus de possiblité de paralléliser)

Exemple Oracle

Séquentiel

```
-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 58808 |  2928K|    68   (0)|
|   1 |  TABLE ACCESS FULL| J    | 58808 |  2928K|    68   (0)|
-------------------------------------------------------------------
```

Parallèle

```
explain plan for
select /*+ parallel(10) */ j.cnum
from J;
@p4
```

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          | 58808 |  2928K|     8   (0)|
|   1 |  PX COORDINATOR      |          |       |       |            |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 | 58808 |  2928K|     8   (0)|
|   3 |    PX BLOCK ITERATOR |          | 58808 |  2928K|     8   (0)|
|   4 |     TABLE ACCESS FULL| J        | 58808 |  2928K|     8   (0)|
-------------------------------------------------------------------------------
```

Tri

Séquentiel

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 58808 |  2928K|       |   818   (1)|
|   1 |  SORT ORDER BY     |      | 58808 |  2928K|  3720K|   818   (1)|
|   2 |   TABLE ACCESS FULL| J    | 58808 |  2928K|       |    68   (0)|
-------------------------------------------------------------------------------
```

Parallèle

```
explain plan for
select /*+ parallel(10) */ j.cnum
from J
sort by j.cnum;
@p4
```

```
-----------------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          | 58808 |  2928K|       |     9  (12)|
|   1 |  PX COORDINATOR         |          |       |       |       |            |
|   2 |   PX SEND QC (ORDER)    | :TQ10001 | 58808 |  2928K|       |     9  (12)|
|   3 |    SORT ORDER BY        |          | 58808 |  2928K|  3720K|     9  (12)|
|   4 |     PX RECEIVE          |          | 58808 |  2928K|       |     8   (0)|
|   5 |      PX SEND RANGE      | :TQ10000 | 58808 |  2928K|       |     8   (0)|
|   6 |       PX BLOCK ITERATOR |          | 58808 |  2928K|       |     8   (0)|
|   7 |        TABLE ACCESS FULL| J        | 58808 |  2928K|       |     8   (0)|
-----------------------------------------------------------------------------------
```

Group by

Séquentiel

```
explain plan for
select j.cnum, count(*)
from J
group by j.cnum;
@p4
```

```
-----------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  5000 | 20000 |    70   (3)|
|   1 |  HASH GROUP BY     |      |  5000 | 20000 |    70   (3)|
|   2 |   TABLE ACCESS FULL| J    | 50000 |   195K|    68   (0)|
-----------------------------------------------------------------------
```

Parallèle

```
explain plan for
select /*+ parallel(10) */ j.cnum, count(*)
from J
group by j.cnum;
@p4
```

```
-----------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |  5000 | 20000 |     9  (12)|
|   1 |  PX COORDINATOR         |          |       |       |            |
|   2 |   PX SEND QC (RANDOM)   | :TQ10001 |  5000 | 20000 |     9  (12)|
|   3 |    HASH GROUP BY        |          |  5000 | 20000 |     9  (12)|
|   4 |     PX RECEIVE          |          | 50000 |   195K|     8   (0)|
|   5 |      PX SEND HASH       | :TQ10000 | 50000 |   195K|     8   (0)|
|   6 |       PX BLOCK ITERATOR |          | 50000 |   195K|     8   (0)|
|   7 |        TABLE ACCESS FULL| J        | 50000 |   195K|     8   (0)|
-----------------------------------------------------------------------------
```

Jointure

Séquentiel

```
explain plan for
select *
from J, C
where j.cnum = c.cnum;
@p4
```

```
-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      | 50000 |  2099K|    76   (2)|
|*  1 |  HASH JOIN         |      | 50000 |  2099K|    76   (2)|
|   2 |   TABLE ACCESS FULL| C    |  5000 |   102K|     7   (0)|
|   3 |   TABLE ACCESS FULL| J    | 50000 |  1074K|    68   (0)|
-------------------------------------------------------------------
```

Parallèle

```
explain plan for
select /*+ parallel(10) */ *
from J, C
where j.cnum = c.cnum;
@p4
```

```
-----------------------------------------------------------------------------
| Id  | Operation                | Name     | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          | 58808 |  5800K|    10   (0)|
|   1 |  PX COORDINATOR          |          |       |       |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10001 | 58808 |  5800K|    10   (0)|
|*  3 |    HASH JOIN             |          | 58808 |  5800K|    10   (0)|
|   4 |     PX RECEIVE           |          |  5000 |   244K|     2   (0)|
|   5 |      PX SEND BROADCAST   | :TQ10000 |  5000 |   244K|     2   (0)|
|   6 |       PX BLOCK ITERATOR  |          |  5000 |   244K|     2   (0)|
|   7 |        TABLE ACCESS FULL | C        |  5000 |   244K|     2   (0)|
|   8 |     PX BLOCK ITERATOR    |          | 58808 |  2928K|     8   (0)|
|   9 |      TABLE ACCESS FULL   | J        | 58808 |  2928K|     8   (0)|
-----------------------------------------------------------------------------
```

Exemple

- Requête parallèle
  - Lire les données d'un fichier en parallèle
    - Select * from Table
  - Accès en parallèle aux pages de données
- Expérience
  - Faire varier le degré de parallélisme
  - Observer l'usage des ressources
  - Mesurer le temps de réponse

Mise en œuvre
outil spark-shell

- Outil d'exécution de requêtes SQL en parallèle

  spark-shell --conf spark.sql.catalogImplementation=in-memory

- Déclarer les tables Film et Note à partir des fichiers de données

  spark.read.option("header", true).csv("movies.csv")

  spark.read.option("header", true).csv("ratings.csv")

- Choisir le parallélisme: N

  spark.read.option("header", true).csv("ratings.csv").coalesce(**N**).createOrReplaceTempView("Note")

  sqlContext.sql("SET spark.sql.shuffle.partitions = **N**")

Rmq: données disponibles sur

https://grouplens.org/datasets/movielens/

Exemple Spark

- Requête

  spark.sql("**select * from Note**").count()

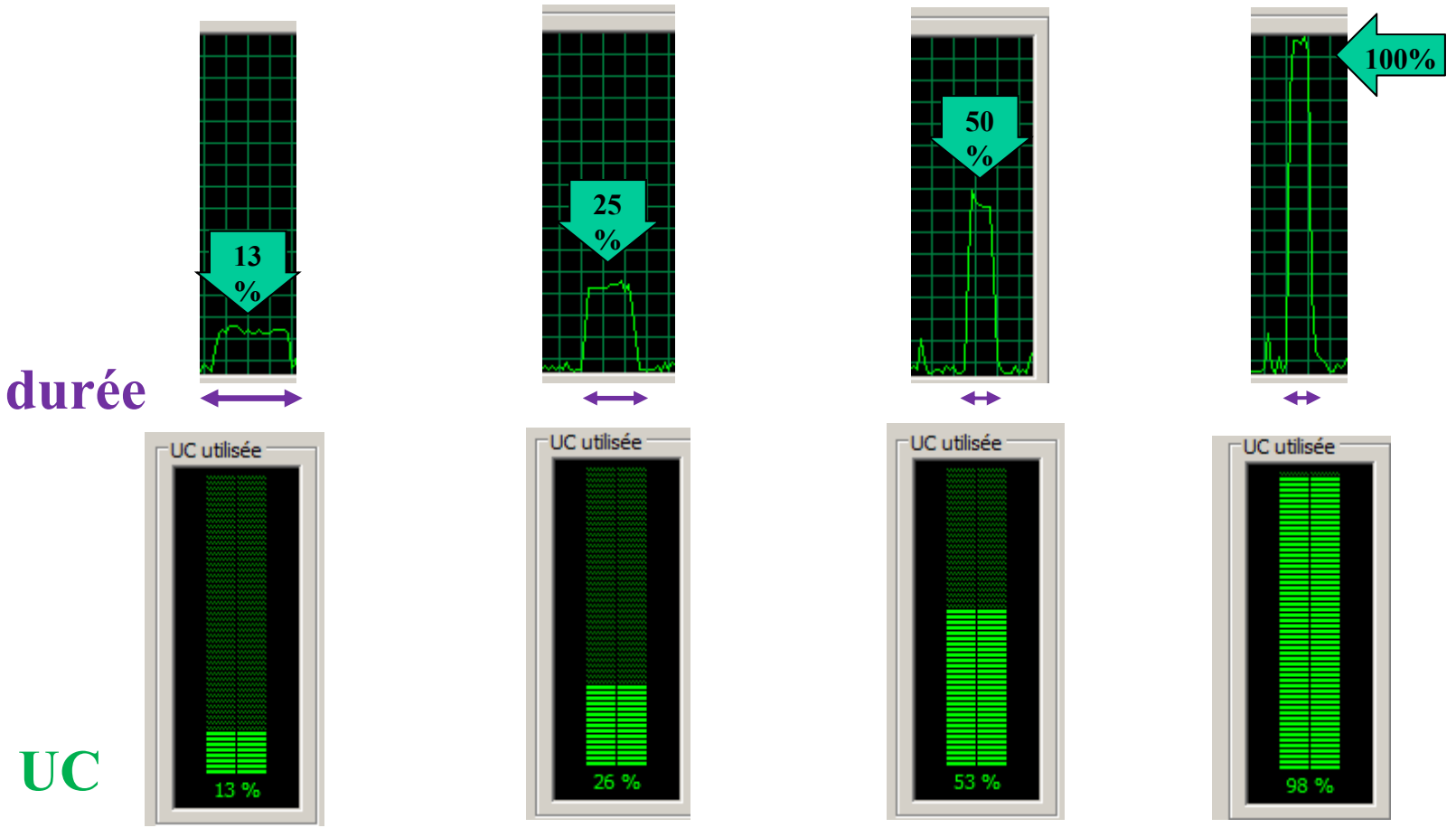
  etc… toute requête SQL (jointure, aggrégation, …)

- Mesurer la durée :

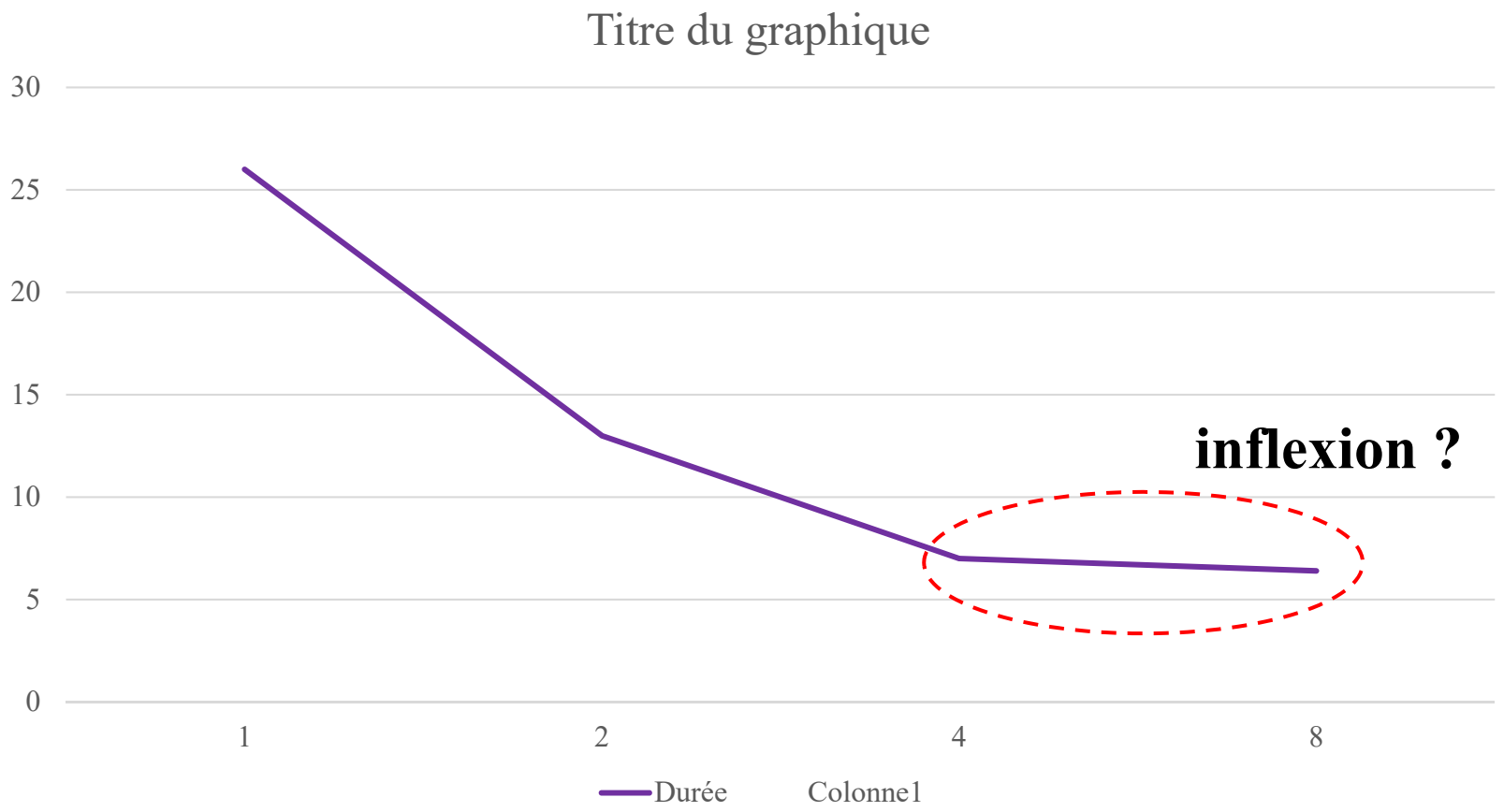
```
val debut = System.currentTimeMillis()
 requête
println("duree : " + (System.currentTimeMillis() - debut))
```

- Observer les ressources utilisées
  - Win : Gestionnaire de tâches
  - Linux : commande: top

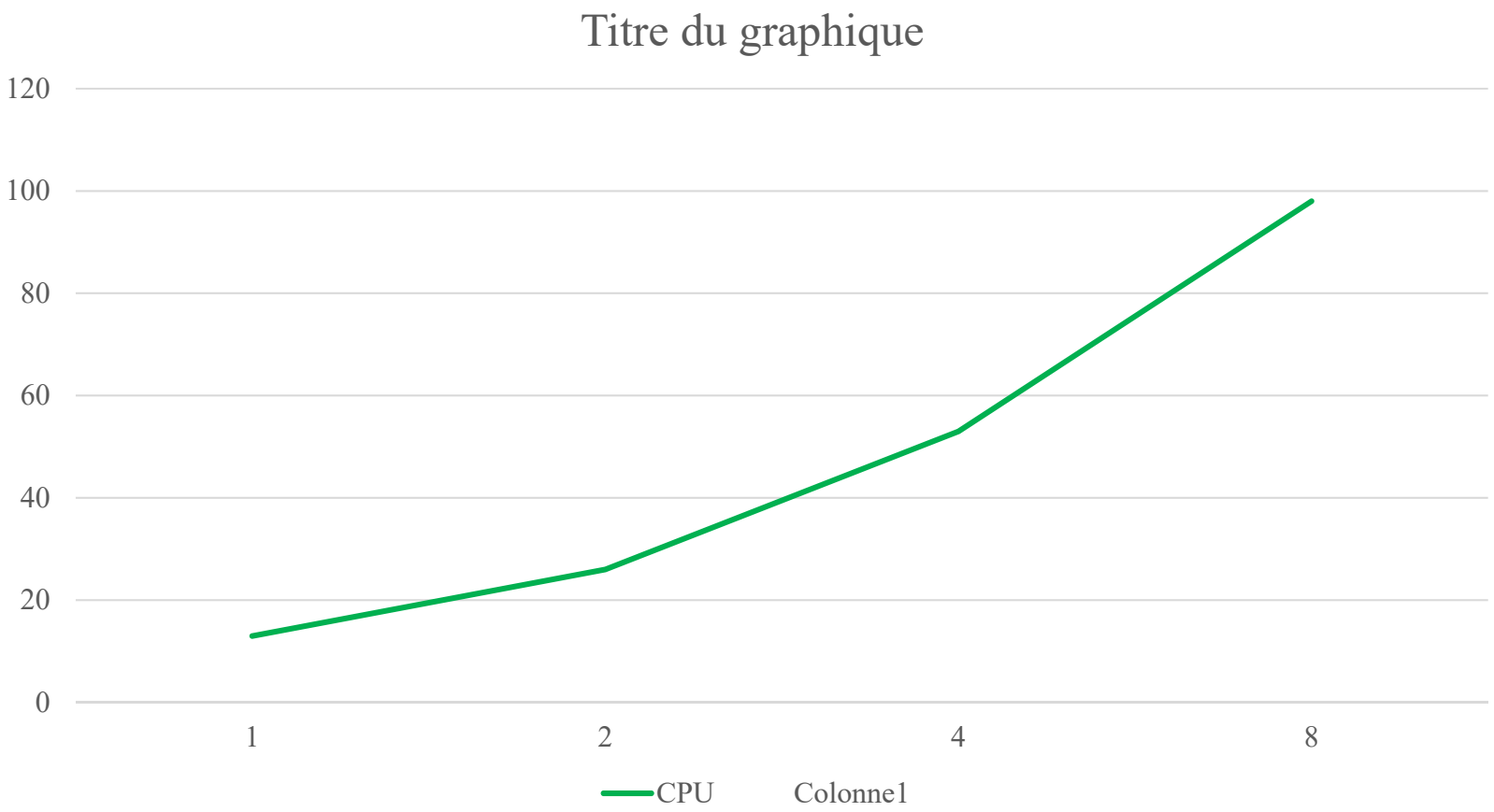
Durée d'une requête



Durée


Titre du graphique
30
25
20
15
10
5
0
1
2
4
8
inflexion ?
Durée
Colonne1

Utilisation CPU



Conclusion

- Les BD parallèles améliorent
  - Les performances,
  - La disponibilité
  - L'extensibilité

Tout en réduisant le ratio prix/performance.

Plusieurs systèmes commercialisés.

Outils de requête parallèle (Spark, Flink)

inspiré de Hadoop Map Reduce: forte utilisation

Pbs :

choisir la meilleure architecture

améliorer les techniques de placement de données (éviter les répartitions inégales)