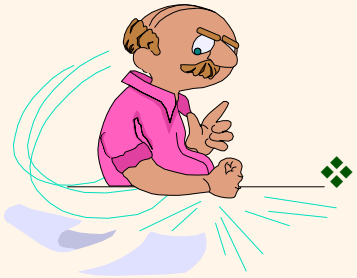
Bases de Données Avancées

Ioana Ileana

Université Paris Descartes

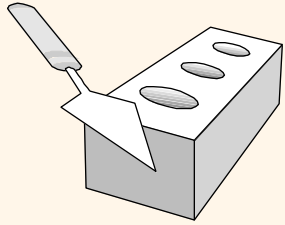
LIPADE
Laboratoire d'Informatique PAris DEscartes

Cours 8: Opérateurs relationnels 2ème partie



- Diapos traduites et adaptées du matériel fourni en complément du livre Database Management Systems 3ed, par Ramakrishnan et Gehrke ; un grand merci aux auteurs pour la réalisation et la disponibilité de ce matériel !
- Les diapos originales (en anglais) sont disponibles ici : http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- Plus particulièrement, ce cours touche aux éléments dans le Chapitre 14A du livre ci-dessus; lecture conseillée! ;)

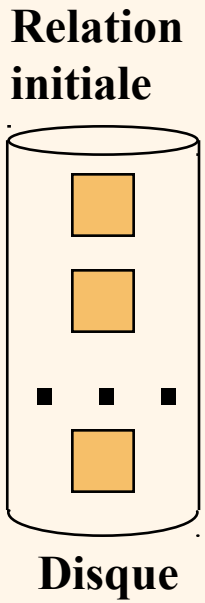
Jointure avec hachage (hash join)



- Partitionne les relations en utilisant une fonction de hachage **h**
  - Pour chaque relation, la fonction de hachage est appliquée aux valeurs dans la colonne de jointure !
  - Les tuples dans la relation sont ainsi distribués en k partitions
    - Avec en général k = B-1, B = nombre de buffers disponibles
- Pour trouver les tuples correspondants, on ne compare que les partitions correspondantes !
- Pour améliorer encore plus les performances, on peut utiliser du hachage « à la volée » pour chaque partition !

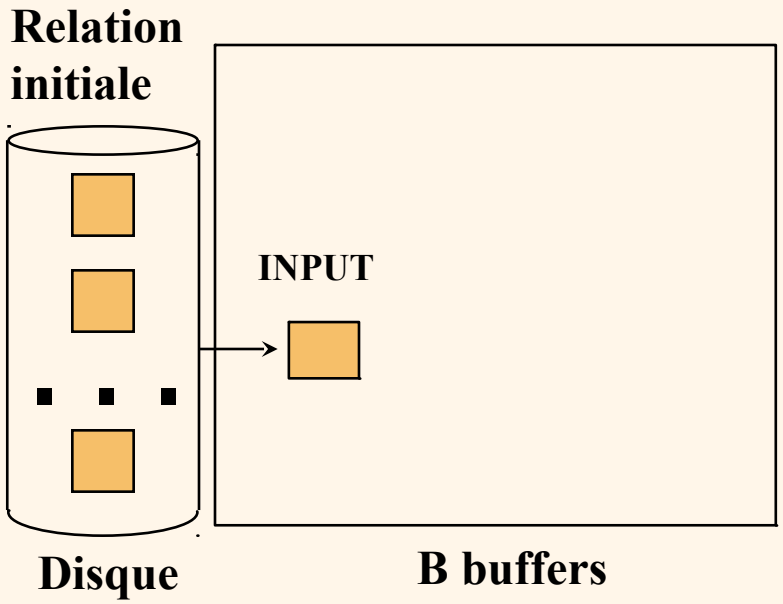
Hash join

- Partitionne chaque relation avec la fonction **h**



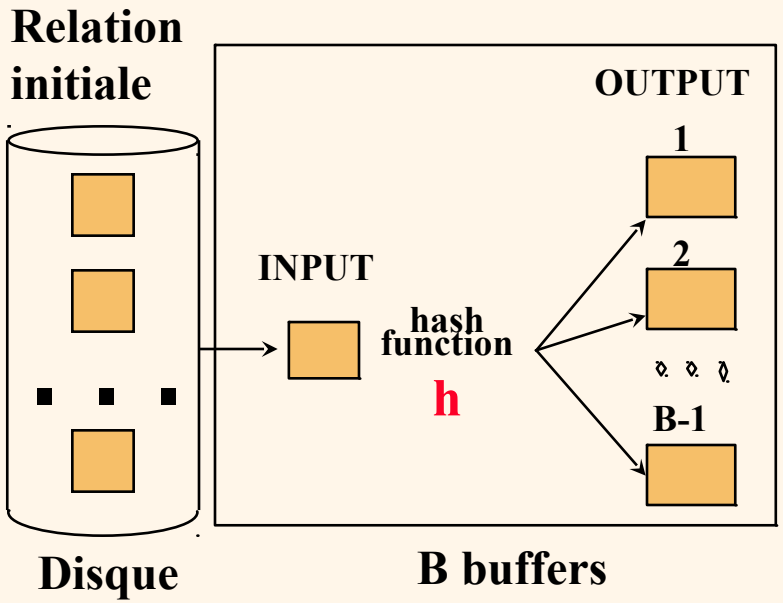
Hash join

- Partitionne chaque relation avec la fonction **h**



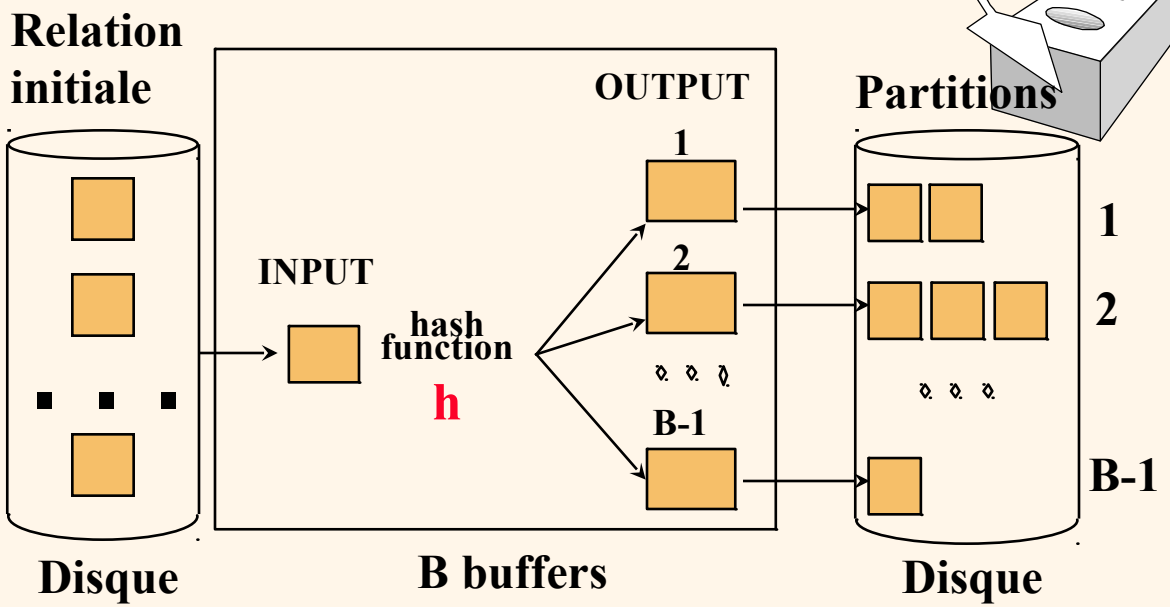
Hash join

- Partitionne chaque relation avec la fonction **h**



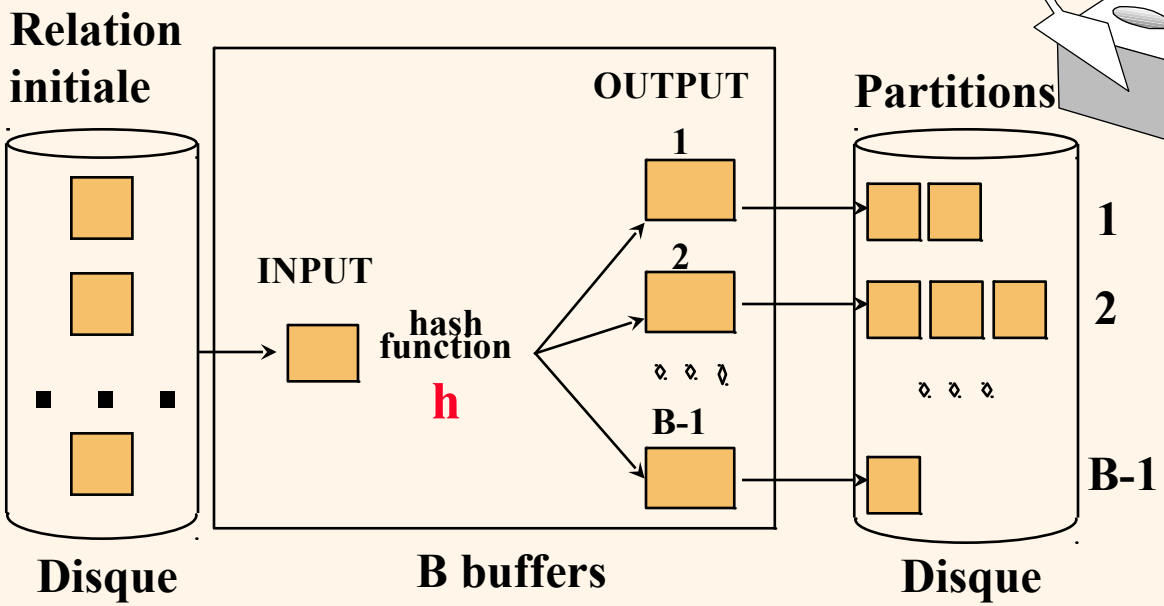
Hash join

- Partitionne chaque relation avec la fonction **h**



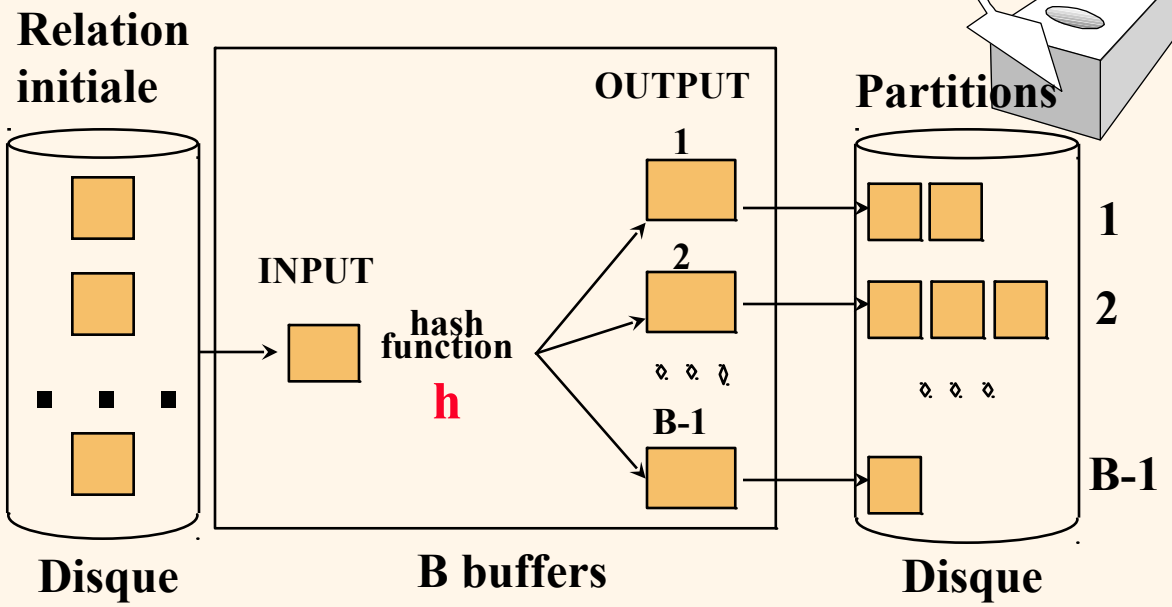
Hash join

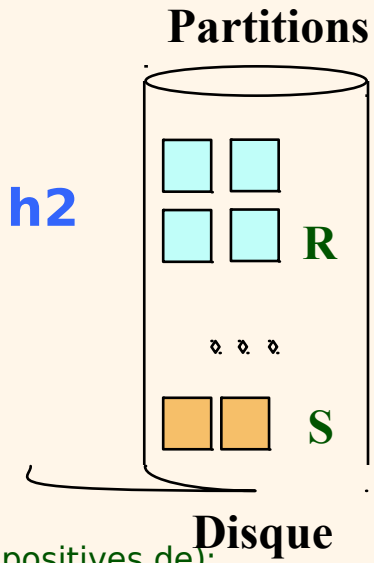
- Partitionne chaque relation avec la fonction **h**
- Les tuples dans la partition i de R ne correspondront qu'aux tuples de S dans la partition i de S !



Hash join

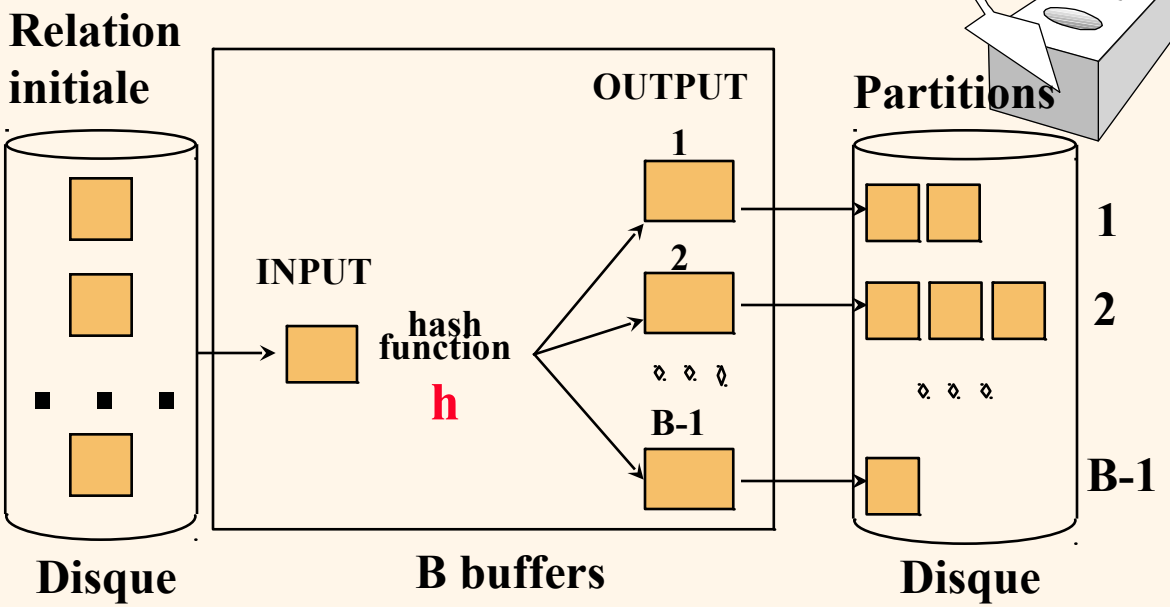
- Partitionne chaque relation avec la fonction **h**
- Les tuples dans la partition i de R ne correspondront qu'aux tuples de S dans la partition i de S !
- Lecture d'une des partitions de R, et construction d'une table de hash avec **h2 (<> h!)**.

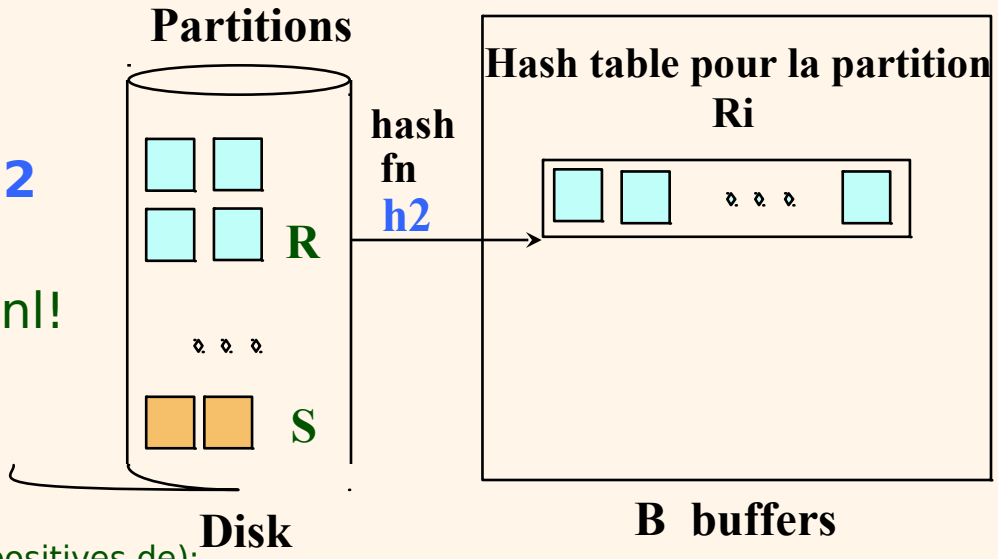




Hash join

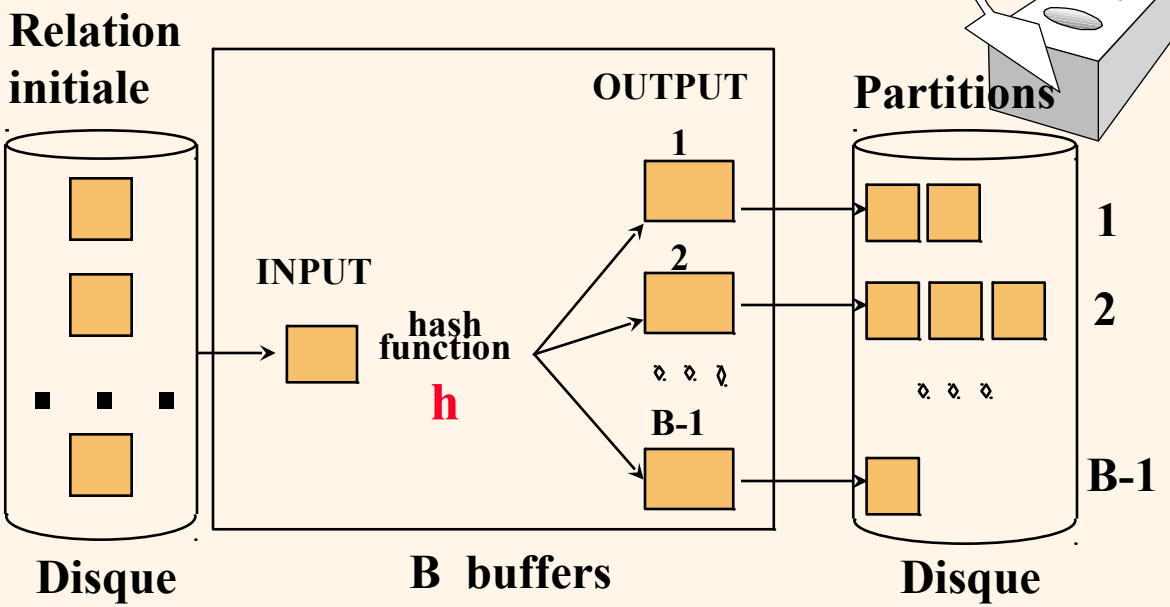
- Partitionne chaque relation avec la fonction **h**
- Les tuples dans la partition i de R ne correspondront qu'aux tuples de S dans la partition i de S !
- Lecture d'une des partitions de R, et construction d'une table de hash avec **h2 (<> h!)**.
  - Similaire au block nl!

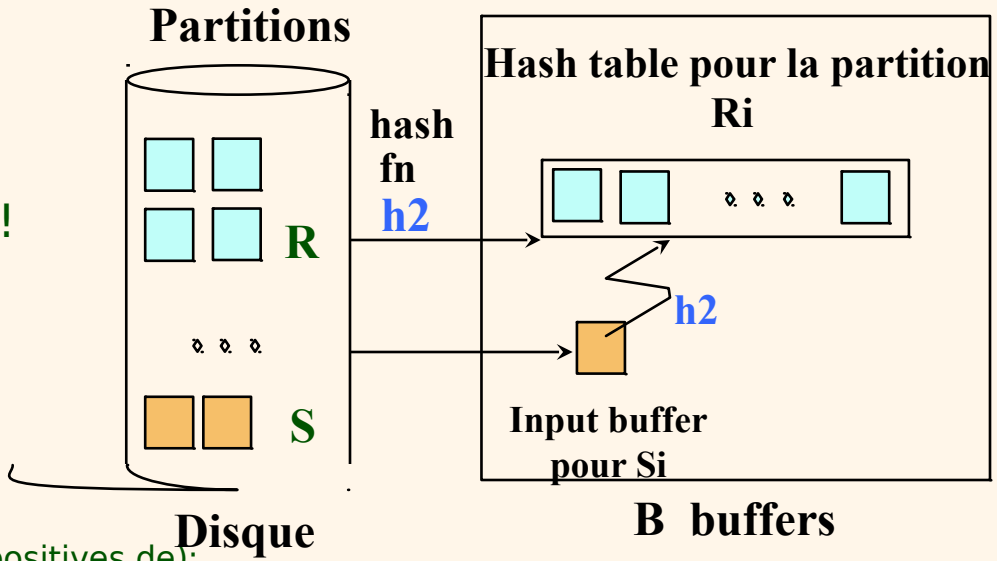




Hash join

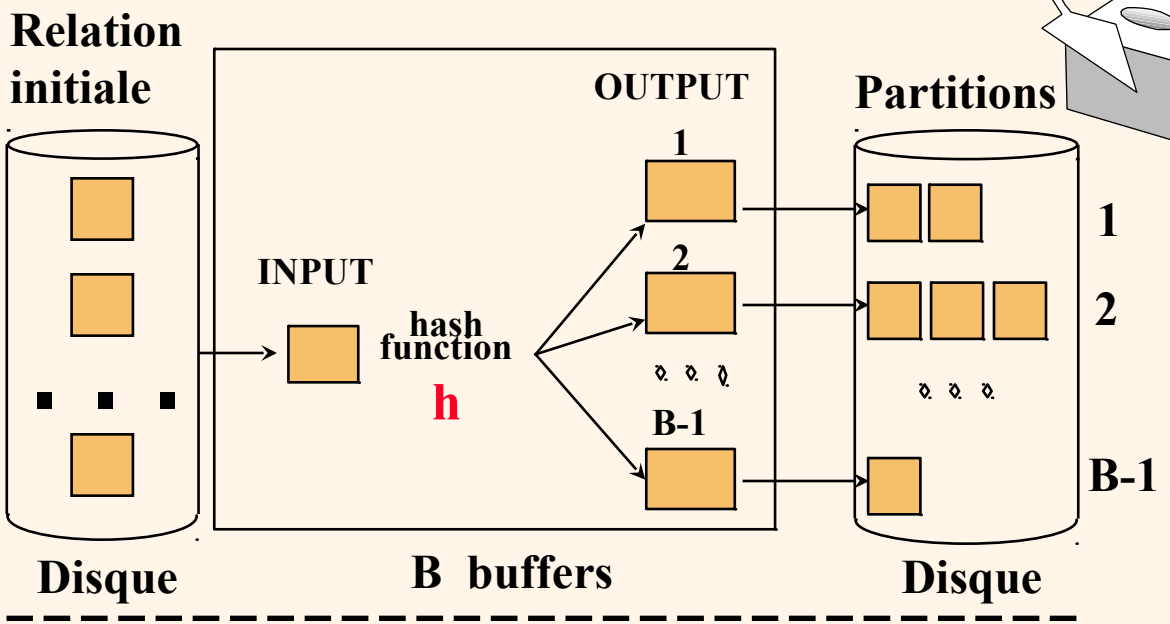
- Partitionne chaque relation avec la fonction **h**
- Les tuples dans la partition i de R ne correspondront qu'aux tuples de S dans la partition i de S !
- Lecture d'une des partitions de R, et construction d'une table de hash avec **h2 (<> h!)**.
  - Similaire au block nl!
- Puis scan de la partition correspondante de S, pour chercher des tuples correspondants!

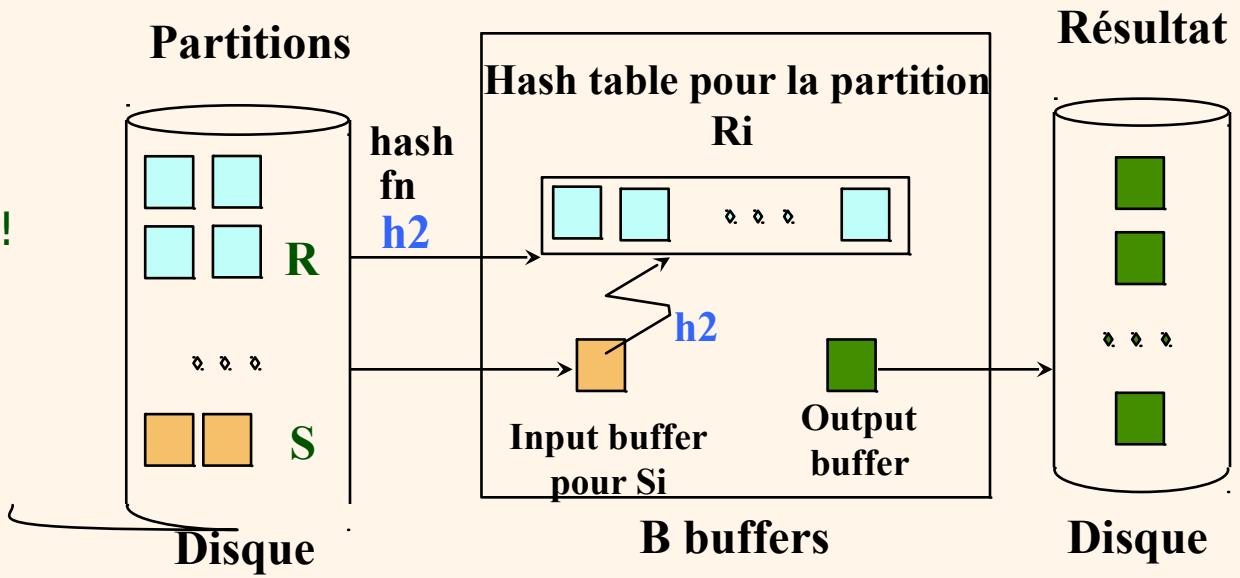


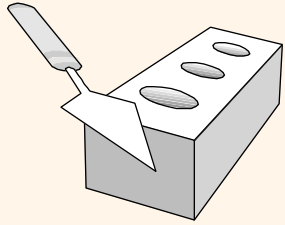


Hash join

- Partitionne chaque relation avec la fonction **h**
- Les tuples dans la partition i de R ne correspondront qu'aux tuples de S dans la partition i de S !
- Lecture d'une des partitions de R, et construction d'une table de hash avec **h2 (<> h!)**.
  - Similaire au block nl!
- Puis scan de la partition correspondante de S, pour chercher des tuples correspondants!



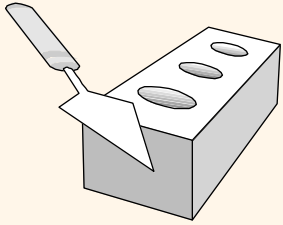




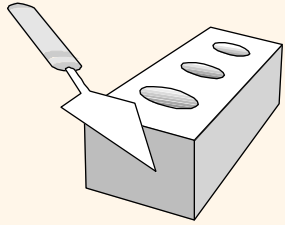
Remarques sur le hash join

- Nombre de partitions k <= B-1 et B-2 >= taille de la plus grande partition. Nous voulons maximiser k et donc choisissons k = B-1.
- Si les partitions font la même taille (uniformes), la taille d'une partition = M / k = M / (B-1).
  - Pour que chaque partition puisse "rentrer en mémoire", il faut que M/(B-1) <= B-2, donc on a besoin d'un nb min de buffers de l'ordre de $\sqrt{M}$
- Si les partitions ne sont pas uniformes (à cause de la distribution des valeurs + la fonction de hash utilisée), ou qu'il y a trop peu de buffers, il peut arriver qu'une partition "ne rentre pas en mémoire"
  - Nous pouvons dans ce cas simplement charger la partition « en plusieurs parties »!

Coût et comparaison avec le sort-merge join

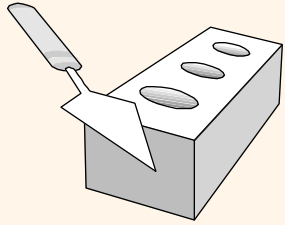


- En supposant assez de mémoire :
  - Partitionnement: 2(M+N) (pourquoi le facteur 2?).
  - Scan pour trouver les tuples correspondants: M+N I/Os.
  - Total : 3(M+N) I/Os
- Pour notre exemple: 4500 I/Os
- Comparé au sort-merge join:
  - Les deux ont le même coût *si assez de mémoire*
  - Avantages du hash join: moins de mémoire demandée si relations de taille sensiblement différente; peut être bien paralélisé
  - Avantages du sort-merge join: moins sensible à la distribution non uniforme des données; les résultats du sort-merge sont triés par construction.



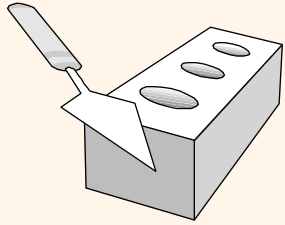
Jointures avec plusieurs égalités

- Exemple: *R.mid=S.mid* AND *R.mnom=S.rnom*
- Si un index existe pour S sur *<mid, rnom>*, il peut être utilisé avec un index nested loops où S est la relation interne ; commentaire symmétrique pour R.
- Si des index séparés existent pour *mid* et *rnom,* on peut adapter le index nested loops join pour faire deux recherches, intersecter les ensembles de rids résultants puis récupérer les tuples de S (voir aussi la suite du cours).
- Pour le sort-merge join et le hash join, il faut trier / partitioner sur la combinaison des deux colonnes de jointure.



Conditions d'inégalité

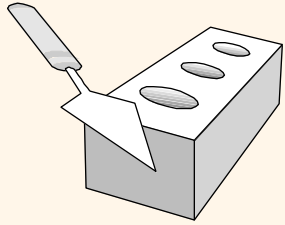
- Exemple: *R.mnom < S.rnom*
- Ces requêtes ont tendance a générer beaucoup plus de résultats que pour les conditions d'égalité!
  - Pour un tuple dans une des relations, on aura beaucoup plus de tuples correspondants dans l'autre relation !
- Si un B+ tree existe sur un des attributs, il est utilisable avec un index nested loops join et la bonne relation interne, mais s'il n'est pas clustered son intérêt peut s'avérer peu important
- Les hash index ne sont pas utilisables
- Le hash join ne s'applique pas
- Le sort-merge join peut être adapté mais ne présente pas en général de vrais avantages de coût...
- Le plus souvent, la meilleure méthode de jointure dans ce cas sera le block nested loops join.



Les sélections

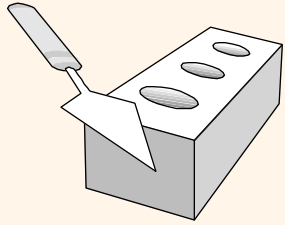
- “Plus faciles” que les jointures à conditions!
  - un “sous-ensemble des techniques” applicables pour les jointures
- Choix: scan avec vérification des conditions, utilisation d’un ou plusieurs index, mix des deux...
- Possibilité de prendre en compte des agrégations complexes de conditions avec des conjonctions (AND) et disjonctions (OR)
- Peuvent bénéficier d’un fichier trié (rare en pratique)

Sélections à une seule condition: usage des index

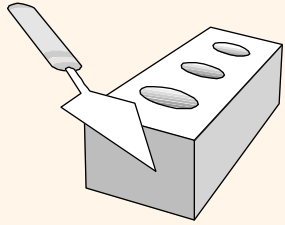


- B+ Tree: 2-4 I/Os pour accéder à “la bonne feuille de départ » + coût de lecture de toutes les entrées qui conviennent (potentiellement dans plusieurs feuilles!) + coût de lecture de tous les tuples qui conviennent
  - Si le B+ Tree n’est pas clustered, comme on a vu pour les jointures, on peut avoir un nombre très important de I/Os pour accéder aux tuples (1 I/O par tuple au pire!) ; cela devient problématique si beaucoup de résultats
  - *Optimisation pour les index non-clustered (alt 2, 3): trier d’abord les rids dans les entrées de données obtenues, pour les « regrouper » par leur partie PageId! L’accès ultérieur aux tuples sera plus efficace!*
- Hash index: 1.2 I/Os pour accéder à “la bonne bucket”; ne s’applique pas aux inégalités !
  - même commentaires clustered vs unclustered (et même optim)
  - Très efficace en général pour les égalités!
  - Mais n’est pas applicable pour les inégalités !

Sélections à conjonction de conditions



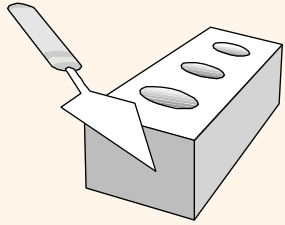
- Deux inégalités qui donnent un intervalle sur le même attribut peuvent être évaluées ensemble si B+Tree existant sur l'attribut en question
  - Comment? ;)
- Pour plusieurs conditions sur des attributs différents et plusieurs index unclustered (alternative 2 ou 3) existants sur ces attributs (et applicables aux conditions respectives → uniquement B+ Trees pour les inégalités):
  - Faire les recherches sur chaque index puis intersecter les listes de rids résultants!
  - Pour l'intersection: le fait d'avoir les rids triés peut être très utile (et servira aussi pour un accès plus efficace aux tuples eux mêmes!)
- Si pas d'index utilisable pour certaines conditions: « post-filtrage » des résultats!
  - Le post-filtrage / la vérification des conditions peut aussi se révéler plus rapide que l'usage de certains index / certaines combinaisons d'index !



Les projections

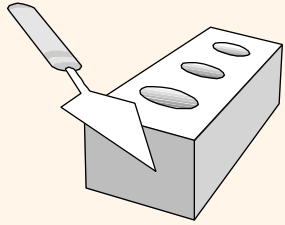
```
SELECT   DISTINCT R.mid, R.bid
FROM     Reservations R
```

- Enlever les attributs non nécessaires – facile
  - Si on dispose d'un index qui contient dans la clé tous les attributs demandés, on peut l'utiliser à la place du fichier original de records ! Les entrées d'index suffiront, pas besoin de regarder les tuples en entier !
- Enlever les doublons : l'étape la plus difficile, et la plus coûteuse
  - Deux approches: par tri et par hashing



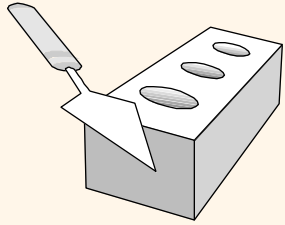
Projections avec tri

- Coût au pire: M log M; mais si assez de mémoire cela devient linéaire!
- La technique de tri la plus utilisée dans les SGBD (tri par fusion) permet d'enlever les doublons directement au cours du tri !
- Si un index de type arbre est disponible pour les attributs retenus, il nous permet de récupérer les entrées triées directement!
  - Si alternative 1 ou 2, on examine les entrées adjacentes et on "ignorer" toutes les entrées qui ont la même valeur pour les attributs!
- L'élimination des doublons par tri est l'approche standard pour la projection (marche bien pour les distributions "biaisées" et nous permet d'obtenir un résultat trié)



Projections avec hashing

- *Partitionnement*: Via une fonction de hash h1, comme pour le join, sauf que pour appliquer la fonction de hash ainsi que dans les partitions on ne retient que les tuples où on a déjà enlevé les attributs non nécessaires
  - Comme pour le join, nous pouvons obtenir B-1 partitions (B le nombre de buffers disponibles)
  - 2 tuples qui ne sont pas dans la même partition ne peuvent pas être des doublons!
- *Elimination des doublons* : Comme pour le join, lire chaque partition et construire une hash table en mémoire en utilisant la fonction de hash *h2* (<> *h1*)
  - L'elimination des doublons peut se faire en même temps que la construction de la table de hash – comment?
- Coût: linéaire si chaque partition « tient en mémoire »; autrement, on doit « re-partitionner » une partition ou la trier !
  - Moins il y a d'attributs retenus, plus on augmente les chances que les partitions « tiennent en mémoire » !



Intersection et union

- Intersection = « cas particulier » de jointure avec conditions d'égalité!
- Union:
  - Tri puis "fusion" = retenir « un seul exemplaire » des tuples égaux
  - Ou utilisation du hashing
    - Partitionnement de R and S avec la fonction de hash *h*.
    - Pour chaque partition de R, construire en mémoire une table de hachage utilisant la fonction *h2*, scanner la partition de S correspondante et rajouter les tuples dans la table en éliminant les doublons
  - Problème d'élimination de doublons similaire au cas de la projection, mais on considère tous les attributs!