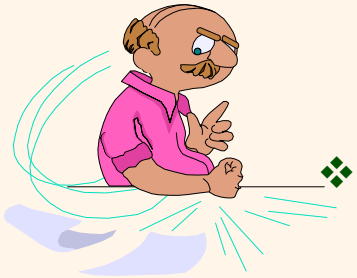
Bases de Données Avancées

Ioana Ileana

Université Paris Descartes

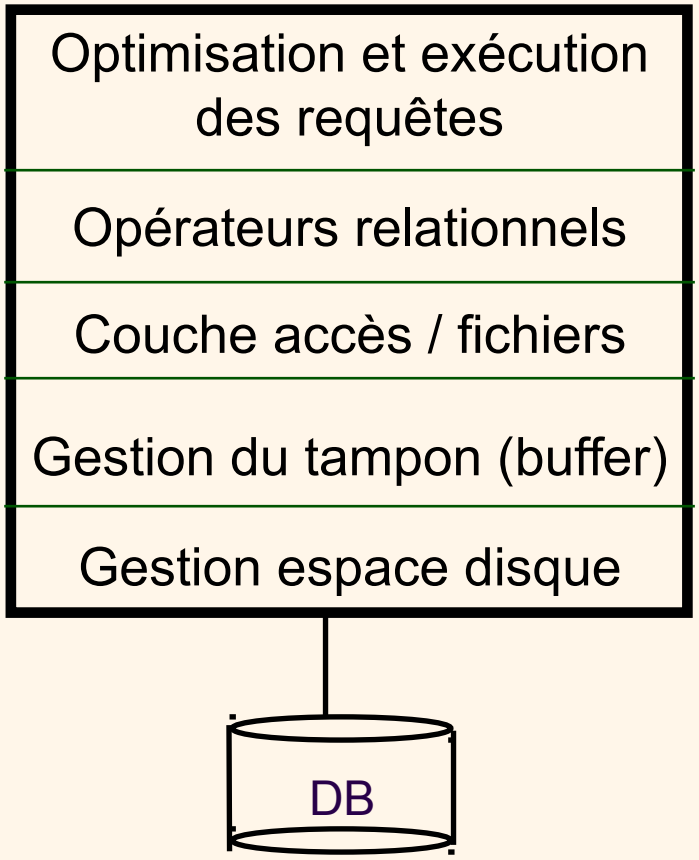
Cours 7b: Opérateurs relationnels 1ère partie



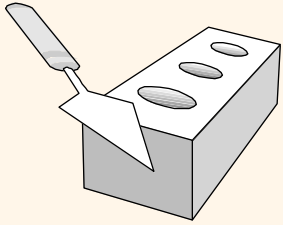
- Diapos traduites et adaptées du matériel fourni en complément du livre Database Management Systems 3ed, par Ramakrishnan et Gehrke ; un grand merci aux auteurs pour la réalisation et la disponibilité de ce matériel !
- Les diapos originales (en anglais) sont disponibles ici : http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html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

Opérateurs relationnels et évaluation / optimisation des requêtes

- Nous allons regarder les algorithmes disponibles pour les opérateurs relationnels, en commençant par les jointures
- Puis nous allons examiner comment optimiser et évaluer les requêtes

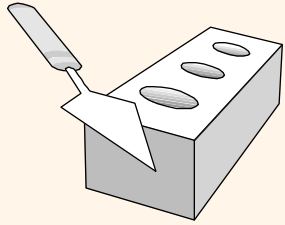


Jointures à conditions



- Opérations très communes, mais qui doivent être bien optimisées, car le produit cartésien est en général très grand, donc produit cartésien puis sélection à posteriori des tuples « composés » qui respectent les conditions = approche inefficace
- Dans ce qui suit, nous allons noter R et S les deux relations impliquées dans la jointure et:
  - p_R le nombre de tuples / page dans R et M le nombre de pages dans l'instance de R
  - p_S le nombre de tuples / page dans S, et N le nombre de pages dans l'instance de S

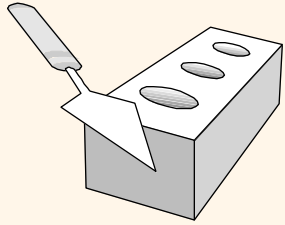
- *Nous allons utiliser come métrique de coût le nombre d'opérations I/O (→ pages lues / écrites)*



Jointures à conditions : exemple

Marins (*mid*: integer, *mnom*: string, *note*: integer, *age*: real)
Réservations (*bid*: integer, *mid*: integer, *jour*: date, *rnom*: string)

- Réservations :
  - Chaque tuple (record) de 40 bytes, 100 tuples / page
  - Instance : 1000 pages.
  - (bid, mid, jour) = clé primaire
- Marins :
  - Chaque tuple de 50 bytes, 80 tuples par page
  - Instance : 500 pages.
  - mid = clé primaire

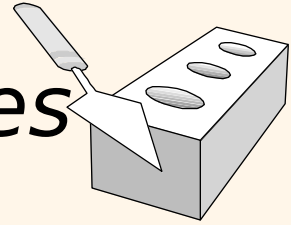


Equi-jointures avec une seule colonne de jointure par relation

```
SELECT  *
FROM    Réservations R1, Marins M1
WHERE   R1.mid=M1.mid
```

- A.R. : Réservations $><_{mid}$ Marins
  - Ou : Réservations $><_{1=2}$ Marins
- Dans le cas général : R $><_{i=j}$ S, avec i la colonne de jointure dans R et j la colonne de jointure dans S
- Quels algorithmes pour calculer le résultat ?

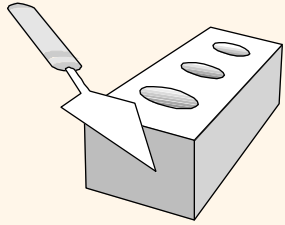
Algorithme simple de jointure à boucles imbriquées (simple nested loops join)



foreach tuple r in R do
    foreach tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

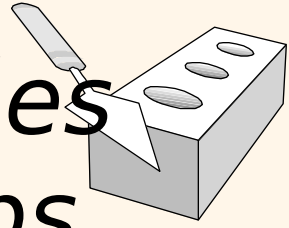
- Pour chaque tuple de la relation *externe* R, nous devons scanner l'intégralité de la relation *interne* S
  - Coût: $M + p_R * M * N$
  - Sur notre exemple, avec R = Réservations et S = Marins: 1000 + 100*1000*500 = 50001000 I/Os.
  - Peut-on faire mieux en inversant les relations interne et externe?

Mieux : algorithme « orienté pages » à boucles imbriquées (page-oriented nested loops join)



- Pour chaque *page* de R, itérer dans toutes les *pages* de S, et trouver les tuples <r, s> qui respectent la condition de jointure!
  - Coût: M + M*N
  - Sur notre exemple, avec R = Réservations et S = Marins: 1000 + 1000*500 = 501000
    - Avec Réservations comme relation externe, un peu mieux: 500500
  - Quelle est la taille (nombre de buffers) minimale dont on a besoin pour le buffer pool, supposant qu'on veut stocker le résultat sur disque ?

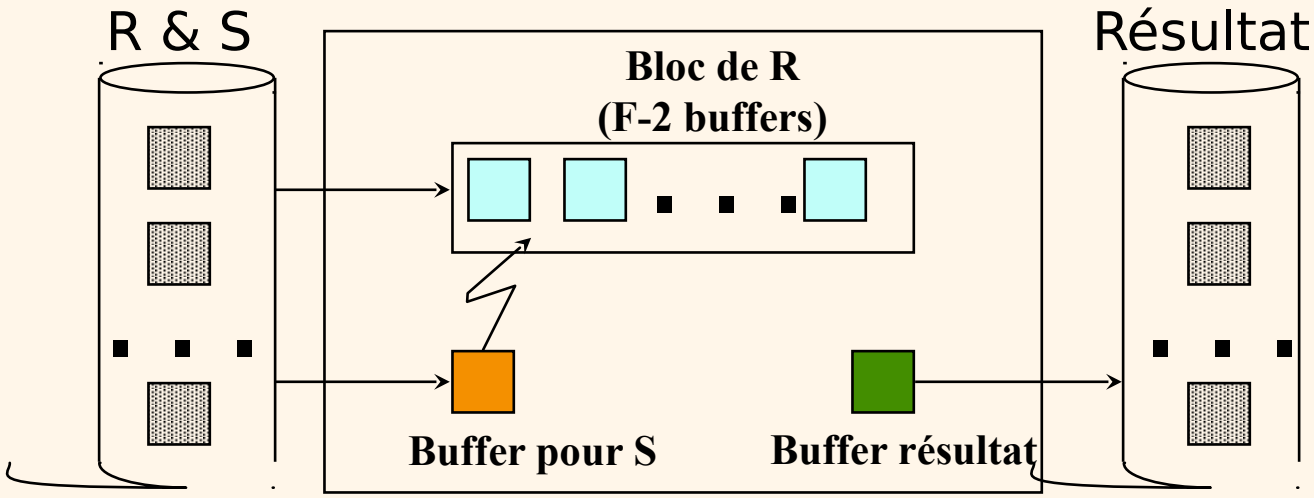
Encore mieux : boucles imbriquées « avec blocs » (block nested loops join)



- Si on dispose de F > 3 buffers dans le buffer pool, on peut « lire la relation externe » par blocs de F-2 pages
- On réserve un buffer pour S et un buffer pour le résultat
- L'algorithme devient : Pour chaque *bloc* de R, itérer dans toutes les *pages* de S, et trouver les tuples <r, s> qui respectent la condition de jointure!
  - Trouver les tuples correspondants : pour chaque page de S lue, on prend chaque S-tuple dans la page et on cherche les tuples correspondants dans le bloc courant chargé pour R
- Coût: M + (M/(F-2))*N

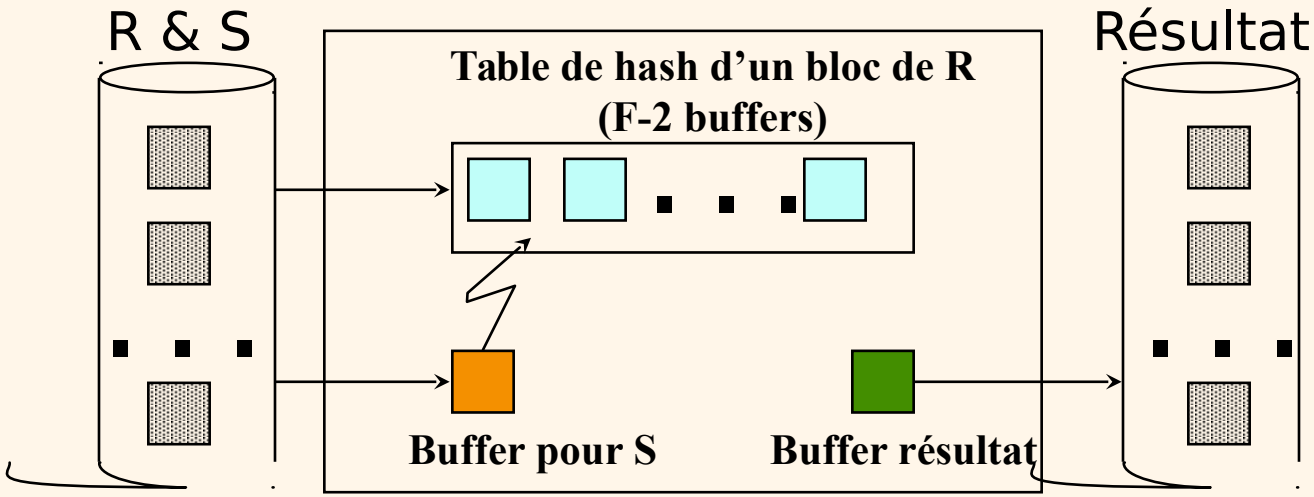
Block nested loops join

- Un buffer pour lire la relation interne S, un buffer pour le contenu courant du résultat, et le reste des buffers pour des “blocs” de la relation externe R.

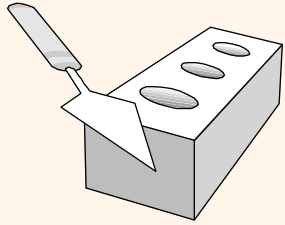


Block nested loops join avec hachage en mémoire « à la volée »

- Construire « à la volée » une table de hash (comme dans les index, mais en mémoire) pour le bloc de R couramment chargé; cela accélère la recherche des tuples correspondants!



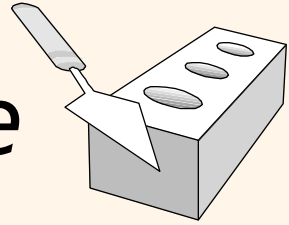
Boucles imbriquées avec index (index nested loops join)



foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

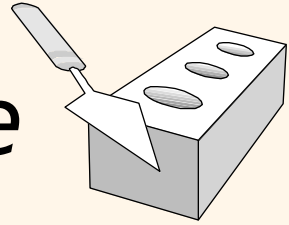
- S'il existe un index sur la colonne j de S, on peut l'exploiter !
  - Coût: M + (M*p_R) * coût pour trouver les tuples S qui respectent la condition de jointure, en utilisant l'index
- Le coût de recherche des entrées index avec la bonne valeur pour la clé est à peu près 1.2 I/Os pour un hash index, 2-4 I/Os pour un B+ Tree
- A cela on rajoute le coût d'accès aux tuples eux-mêmes (si alternative 2 ou 3), qui dépendra du caractère clustered ou pas de l'index
  - Index clustered: 1 I/O (pour tous les tuples de S qui correspondent)
  - Index unclustered: peut être de 1 I/O pour chaque tuple de S qui correspond (donc pb si beaucoup de doublons pour la clé)

Index nested loops join sur notre exemple

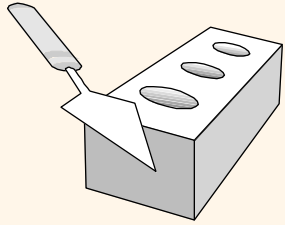


- Si on dispose d'un hash index avec alternative 2 sur la colonne *mid* de Marins, nous pouvons prendre Marins comme relation interne et obtenons:
  - Scan de Réservations: 1000 pages (I/Os), 100*1000 tuples.
  - Pour chaque tuple de Réservations: 1.2 I/Os (en moyenne) pour atteindre l'entrée correspondante (juste une! Pourquoi?) dans l'index, plus 1 I/O pour accéder au (seul ! Pourquoi ?) tuple de Marins pointé par l'entrée.
  - Coût total: 221,000 I/Os.

Index nested loops join sur notre exemple

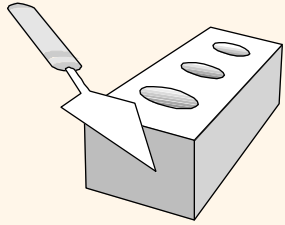


- Si on dispose d'un hash index avec alternative 2 sur la colonne *mid* de Réservations, nous pouvons prendre Réservations comme relation interne et obtenons:
  - Scan de Marins: 500 pages (I/Os), 80*500 tuples.
  - Pour chaque tuple de Marins: 1.2 I/Os (en moyenne) pour atteindre les entrées correspondantes, plus le coût d'accéder au tuples correspondants dans Réservations
  - Si on suppose une distribution uniforme, cela nous fait 2.5 réservations / marin
  - Le coût d'y accéder sera donc 1 si index clustered ou 2.5 si index unclustered
  - Nous obtenons donc un coût total qui varie de 88500 (index clustered) à 148500 (index unclustered)



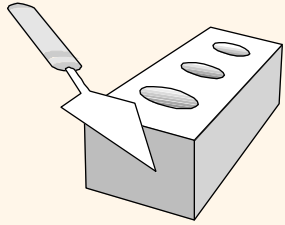
Jointure par tri puis fusion (sort-merge join)

- Soient i et j les colonnes impliquées respectivement dans l'égalité de la jointure : $R >\!\!<_{i=j} S$
- Pour la jointure:
  - Trier R sur la colonne i et S sur la colonne j
  - Scanner les relations pour les "**fusionner**" (merge) sur les colonnes de jointure, et prendre les tuples résultants!



Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Deux tableaux triés, trouver tous les couples d'éléments égaux:
  - « Curseur courant » dans chacun des tableaux, initialement sur le premier élément
  - Répéter tant que les deux curseurs n'ont pas dépassé les tableaux
    - Répéter tant que les éléments sous les deux curseurs ne sont pas égaux
      - Tant que l'élément sous le curseur du tableau 1 est inférieur à l'élément sous le curseur du tableau 2, avancer le curseur du tableau 1
      - Tant que l'élément sous le curseur du tableau 2 est inférieur à l'élément sous le curseur du tableau 1, avancer le curseur du tableau 2
    - Marquer la position courante du curseur dans le tableau 2 (pos)
    - Répéter tant que l'élément sous le curseur du tableau 1 est égal à l'élément sous pos
      - Mettre le curseur du tableau 2 à pos
      - Tant que l'élément sous le curseur du tableau 2 est égal à l'élément sous le curseur du tableau 1, afficher les deux éléments sous le curseur et avancer le curseur du tableau 2
      - Avancer le curseur du tableau 1

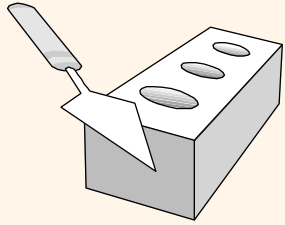


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

- Résultat :

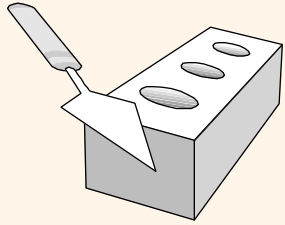


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

- Résultat :

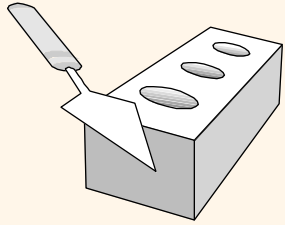


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

- Résultat :



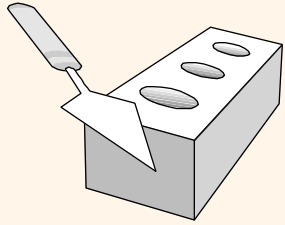
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat :

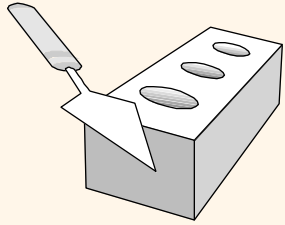


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10
- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7)

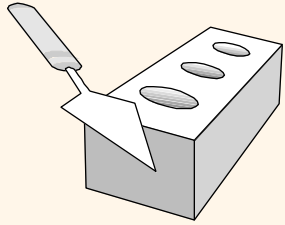


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10
- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7)

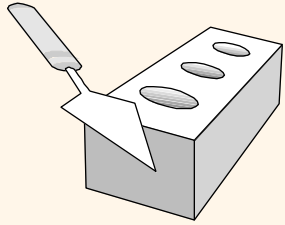


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10
- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7)



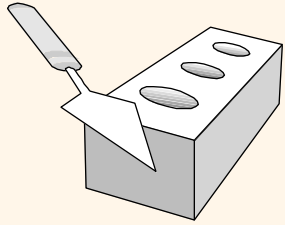
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7)



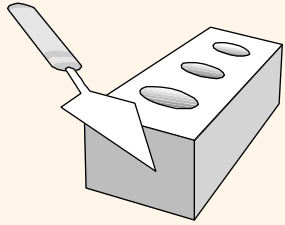
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8)



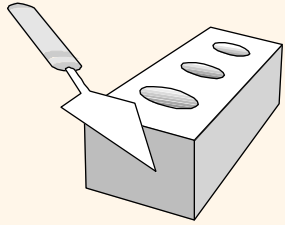
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8)



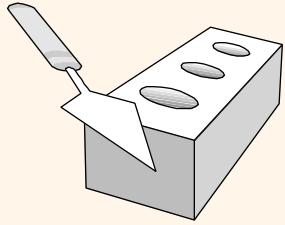
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

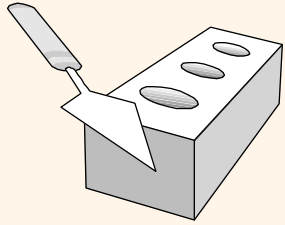
pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8),(8,8)



Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10
- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10
- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8),(8,8)



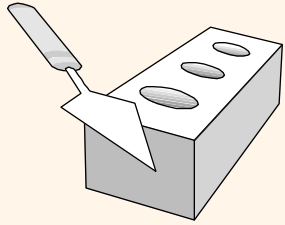
Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10

- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8),(8,8)

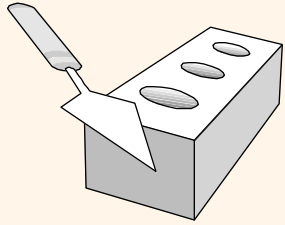


Rappels : algorithme type fusion (merge) pour trouver les éléments égaux dans deux tableaux triés

- Tableau 1 : 1, 2, 7, 7, 8, 8, 9, 10
- Tableau 2 : 3, 3, 7, 7, 7, 8, 10, 10

pos

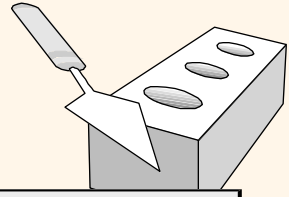
- Résultat : (7,7), (7,7), (7,7),(7,7),(7,7),(7,7),(8,8),(8,8),(10,10), (10,10)



Jointure par tri puis fusion (sort-merge join)

- Soient i et j les colonnes impliquées respectivement dans l'égalité de la jointure : $R ><_{i=j} S$
- Pour la jointure:
  - Trier R sur la colonne I et S sur la colonne j
  - Scanner les relations pour les "fusionner" (merge) sur les colonnes de jointure, et prendre les tuples résultants
- Le scan se déroule de manière similaire au parcours des tableaux ; plutôt que de chercher des éléments égaux dans des tableaux on cherche des tuples dont les valeurs sur les colonnes i et j sont égales!
- R est scannée une seule fois; pour S on a des scans multiples potentiels (quand on « remet le curseur à pos »)

Exemple de sort-merge join



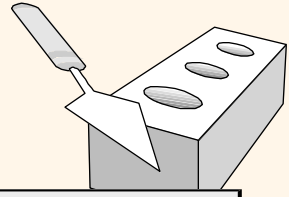
mid	mnom	note	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom

Exemple de sort-merge join



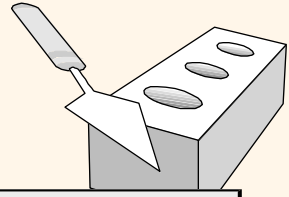
mid	mnom	note	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom

Example of Sort-Merge Join



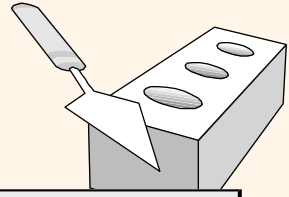
mid	mnom	note	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom

Exemple de sort-merge join



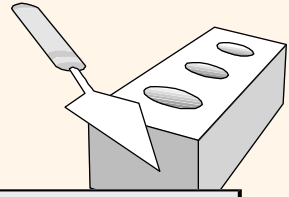
mid	mnom	note	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom

Exemple de sort-merge join



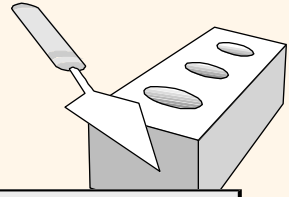
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy

Exemple de sort-merge join



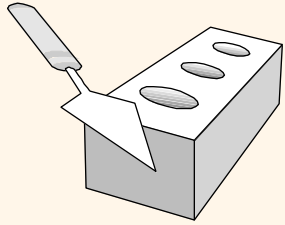
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

mid	bid	jour	rnom
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

résultat

mid	mnom	note	age	bid	jour	rnom
28	yuppy	9	35.0	103	12/4/96	guppy
28	yuppy	9	35.0	103	11/3/96	yuppy

Coût du sort-merge join



- Coût: M log M + N log N + (M+N)
  - Sur notre exemple: 10000 + 4500+ 1500 = 16000 I/Os
- Le coût du scan est en général M+N; peut dans de très rares cas monter jusqu'à M*N (à cause des "re-parcours de certaines parties de la relation interne" en cas de doublons de valeurs).
- D'un autre côté, le tri peut être fait de manière linéaire (en I/Os: 2*M respectivement 2*N) si assez de pages buffer disponibles
  - on peut donc obtenir un coût total du join de 2000+1000 + 1500 = 4500 I/Os