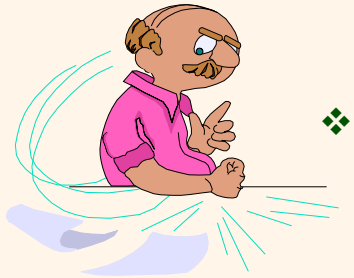
Bases de Données Avancées

Ioana Ileana

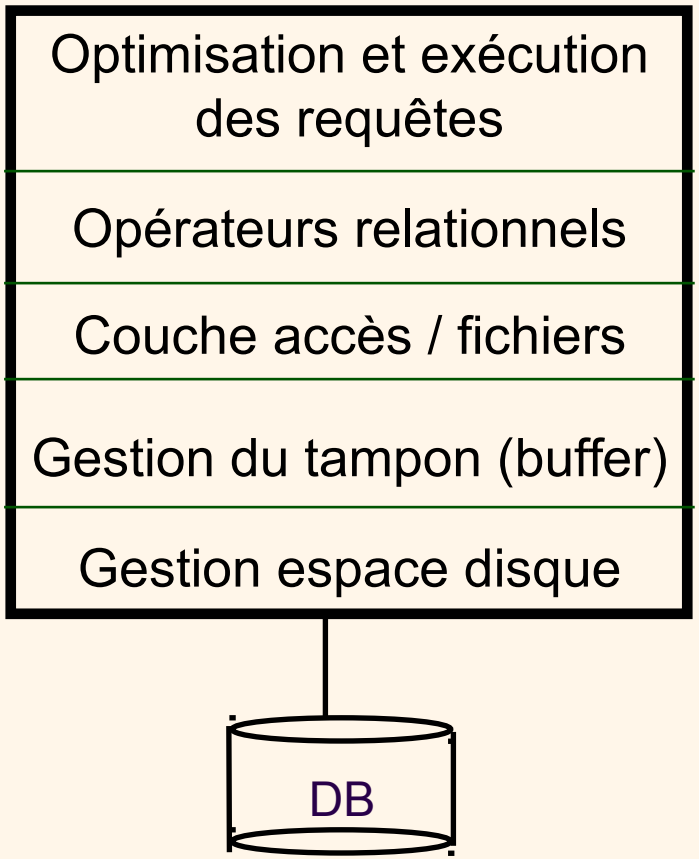
Université Paris Descartes

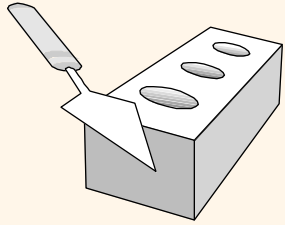
Cours 11: Optimisation et exécution des requêtes



- 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 15 du livre ci-dessus; lecture conseillée! ;)
- Merci également à Themis Palpanas pous les diapositives complémentaires sur les requêtes, qui sont en partie traduites et adaptées dans ce cours!

Optimisation et exécution des requêtes





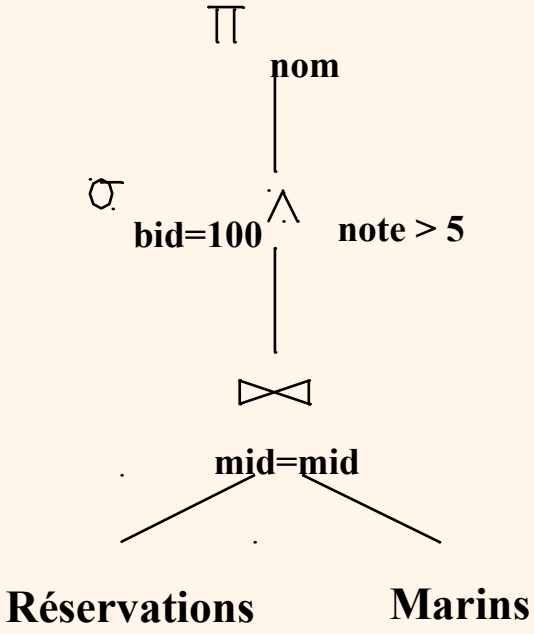
Exemple (rappel)

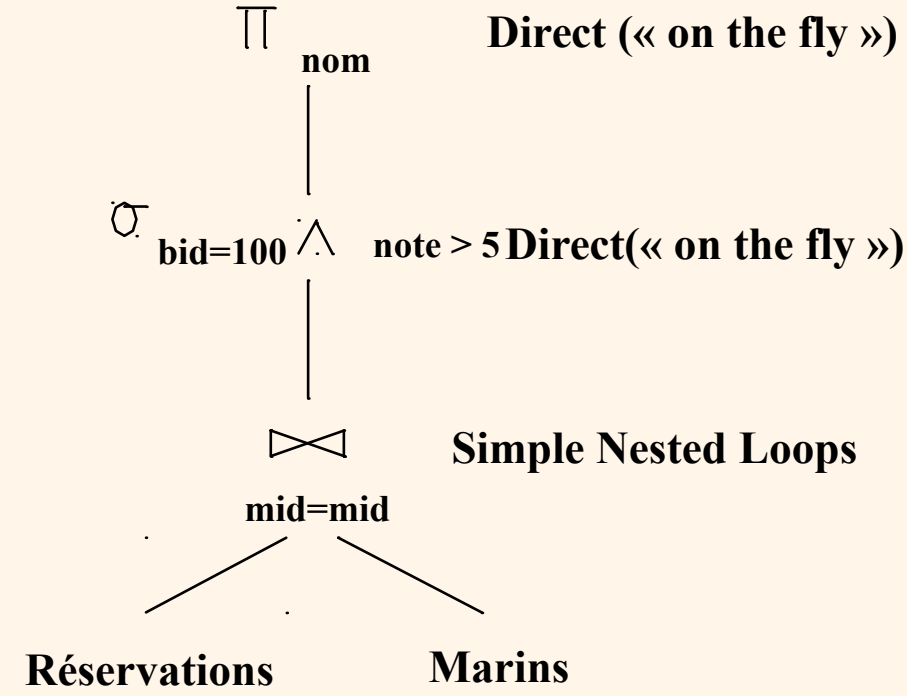
Marins (*mid*: integer, *mnom*: string, *note*: integer, *age*: real)
Réservations (*bid*: integer, *mid*: integer, *jour*: date, *rnom*: string)

- 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

Plans d'exécution

- *Plan (physique)*: *Arbre d'opérateurs d'Algèbre Relationnele, avec un choix d'algorithme pour chaque opérateur ; il contient implicitement l'ordre d'évaluation des opérateurs.*

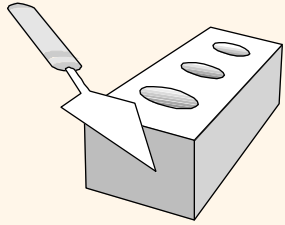




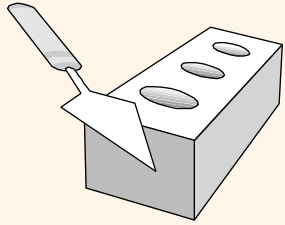
Arbre A.R. (« plan logique »)

Plan (« physique »)

Matérialisation et pipelining



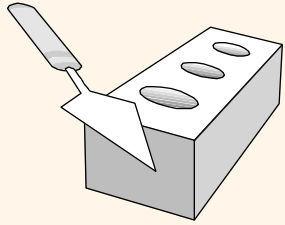
- Dans un plan, les opérateurs dans les nœuds parents prennent en entrée les résultats des opérateurs dans les nœuds fils. Deux alternatives pour récupérer ces résultats :
  - Matérialisation: générer les résultats du fils et les matérialiser = écrire sur les disque. Dans ce cas, l'opérateur parent utilise les résultats du fils de la même manière qu'il utilise une relation stockée sur disque.
  - Pipelining: l'opérateur fils "transmet" à l'opérateur parent les tuples qui font partie de son résultat pendant que ces tuples sont générés (pendant l'exécution donc de l'opérateur fils).



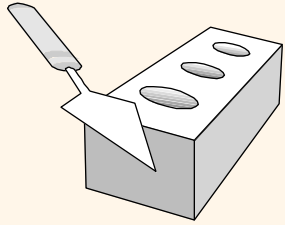
Matérialisation

- La stratégie de matérialisation peut toujours être appliquée
- Mais le coût d'écrire et puis relire les résultats vers / depuis le disque peut être élevé!
  - Les coûts des écritures de tous les résultats intermédiaires se rajouteront au coût total de l'exécution !
- Optimisation: double buffering = utiliser deux buffers pour le résultat de chaque opération; quand un des buffers est rempli, l'écrire sur disque et "switcher vers le remplissage de l'autre" :
  - Cela permets de paralléliser les «calculs des résultats» avec les écritures disque et diminue ainsi le temps d'exécution total (même sans réduire le nombre d'accès disque).

Pipelining

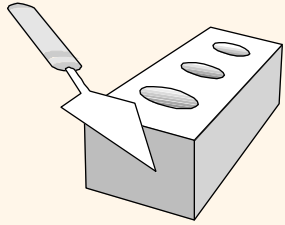


- Pipelining: “passer directement” les résultats d’un opérateur fils à l’opérateur parent
  - Cela permet d’évaluer “simultanément” plusieurs opérateurs (l’opérateur parent n’est pas forcé d’attendre la fin de l’évaluation des opérateurs fils)
- Pour avoir de vrais bénéfices suite au pipelining, il faut utiliser des algorithmes qui génèrent des tuples résultats au fur et à mesure que les tuples sont reçus ou lus en entrée
- Deux modalités opérationnelles pour le pipelining:
  - pipelining dirigé par la demande (demand-driven)
  - pipelining dirigé par la production (produce-driven)



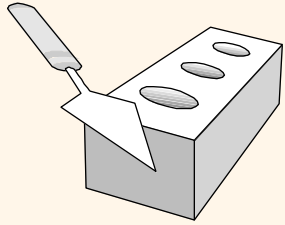
Demand driven pipelining

- L’ opérateur fils implémente une interface de type « itérateur » comprenant les méthodes :
  - Init()
    - Ex : pour une sélection basée sur le scan, initialiser la lecture du fichier = lecture de la première page, pointeur courant sur le premier tuple de la première page
  - Next() - retourne le prochain résultat (valeur null ou assimilée si fin des résultats)
    - Dans le cas de la séléction avec scan, vérification des conditions, avancement du pointeur de tuple, lecture potentielle de la prochaine page...
  - Close()
- Ce sont les appels *depuis les parent (les « demandes » du parent)* qui dirigeront la transmission du prochain résultat depuis le fils vers le parent



Produce-driven pipelining

- L'opérateur fils produit "à son propre rythme" les tuples dans le résultat et le transfère au parent
- Pour mieux gérer la synchronisation, il existe souvent un buffer (type « file ») entre l'opérateur fils et l'opérateur parent
  - Le fils dépose les résultats dans le buffer, le parent « collecte » ces résultats
  - Si le buffer est rempli, le fils doit attendre que de la place se crée
  - Le SGBD va « donner la priorité » aux opérateurs pour lesquels ils reste de la place dans le buffer (les opérateurs qui ne sont donc pas « bloqués en attente »)

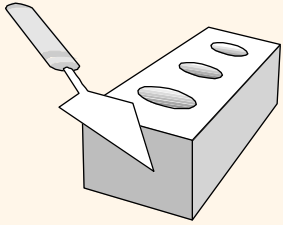


Choix des plans et optimisation

Plan: *Arbre d'opérateurs d'Algèbre Relationnele, avec un choix d'algorithme pour chaque opérateur*

- Aspects essentiels:
  - Pour une requête donnée, quels sont les plans considérés?
  - Algorithmes pour trouver le meilleur (moins cher) plan
- Idéalement: Trouver les meilleurs plan. En pratique: éviter les pires plans ;)
- Une approche classique : celle de l'optimiseur de System R (https://en.wikipedia.org/wiki/IBM_System_R), toujours reprise dans les SGBDs d'aujourd'hui !

Règles d'équivalence A.R. (1)



1. Une sélection avec une conjonction de conditions peut être remplacée par une séquence de séléctions "individuelles".

$$\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$$

2. Les sélections sont commutatives.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$$

3. Il y a besoin uniquement de la dernière projection dans une séquence de projections (les autres peuvent être ignorées).

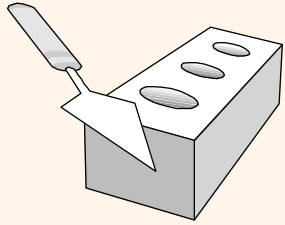
$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(R))\ldots)) = \Pi_{t_1}(R)$$

4. Les sélections peuvent être combinées avec le produit cartésien et les jointures à conditions (théta-jointures).

   α. $\sigma_\theta(R_1 \times R_2) = R_1 \bowtie_\theta R_2$

   β. $\sigma_{\theta 1}(R_1 \bowtie_{\theta 2} R_2) = R_1 \bowtie_{\theta 1 \wedge \theta 2} R_2$

Règles d'équivalence A.R. (2)



5. Les théta-jointures sont commutatives.

$$R_1 \bowtie_\theta R_2 = R_2 \bowtie_\theta R_1$$

6. (a) Les jointures naturelles sont associatives.

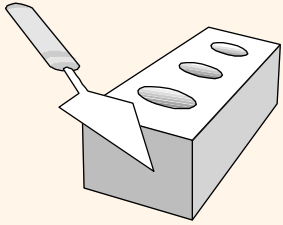
$$(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$$

(b) Les théta-jointures (jointures à conditions) générales sont "associatives" de la manière suivante:

$$(R_1 \bowtie_{\theta 1} R_2) \bowtie_{\theta 2 \wedge \theta 3} R_3 = R_1 \bowtie_{\theta 1 \wedge \theta 3} (R_2 \bowtie_{\theta 2} R_3)$$

=si θ_2 *ne concerne que des attributs de* R_2 et R_3.

Règles d'équivalence A.R. (3)



7. Les sélections peuvent se distribuer sur les théta-jointures dans les cas suivants:

(a) Les attributs de θ_0 ne concernent qu'une des relations R1 participant à la jointure:

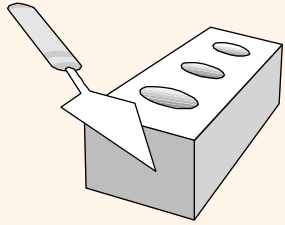
$$\sigma_{\theta 0}(R_1 \bowtie_{\theta} R_2) = (\sigma_{\theta 0}(R_1)) \bowtie_{\theta} R_2$$

(b) Les attributs de θ_1 concernent juste R_1 et ceux de θ_2 juste R_2.

$$\sigma_{\theta 1 \wedge \theta 2}(R_1 \bowtie_{\theta} R_2) = (\sigma_{\theta 1}(R_1)) \bowtie_{\theta} (\sigma_{\theta 2}(R_2))$$

!! Règles très importantes car elles permettent de « pousser les sélections » au plus proche des relations, et donc de diminuer le nombre de tuples participants aux jointures !!

Règles d'équivalence A.R. (4)



8. Les projections se distribuent sur les théta-jointures comme suit (L_1 and L_2 ensembles d'attributs de E_1 et E_2, respectivement) :

   (a) si θ concerne uniquement des attributs de $L_1 \cup L_2$:

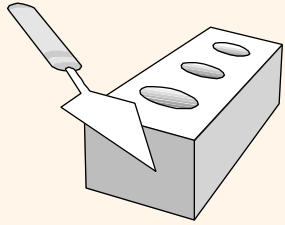
$$\prod_{L_1 \cup L_2}(E_1 \underset{\theta}{\bowtie} E_2) = (\prod_{L_1}(E_1)) \underset{\theta}{\bowtie} (\prod_{L_2}(E_2))$$

   (b) Pour une jointure $E_1 \bowtie_\theta E_2$, et:

   - L_3 attributs de E_1 qui apparaissent dans θ, mais ne sont pas dans $L_1 \cup L_2$
   - L_4 attributs de E_2 qui apparaissent dans θ, mais ne sont pas dans $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \underset{\theta}{\bowtie} E_2) = \prod_{L_1 \cup L_2}((\prod_{L_1 \cup L_3}(E_1)) \underset{\theta}{\bowtie} (\prod_{L_2 \cup L_4}(E_2)))$$

Exemple



- Schéma : branch (agence), account (compte), depositor(client)
- Requête: Trouver les noms de tous les clients qui ont un compte à l'agence de Brooklyn dont le solde est strictement supérieur à 1000.

$\Pi_{cname}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$

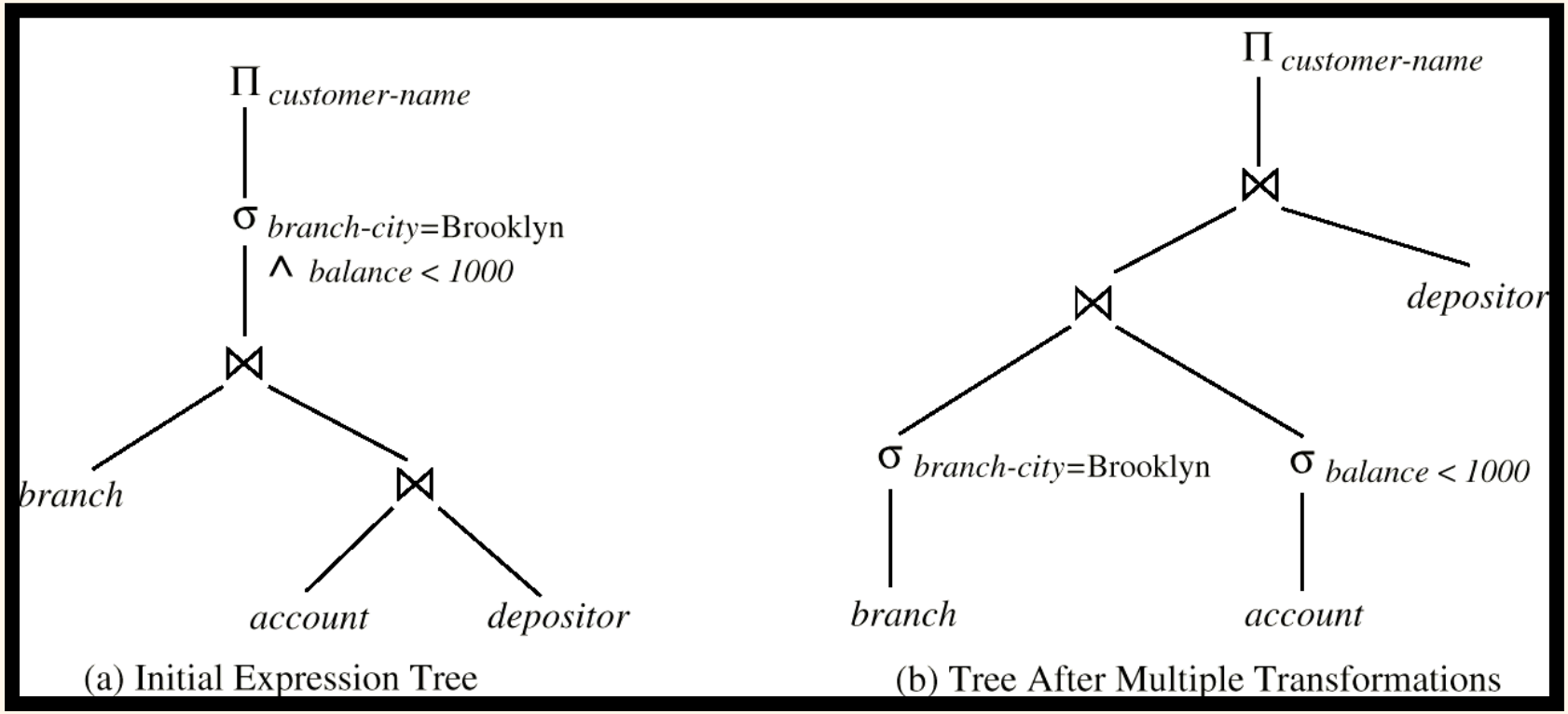
- Transformation avec Règle 6a (jointures naturelles associatives):

$\Pi_{cname}(\sigma_{branch\text{-}city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account) \bowtie depositor))$

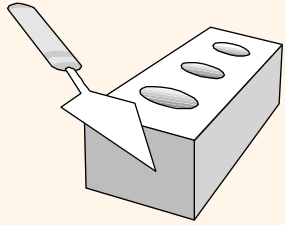
Puis avec 7a et 7b nous pouvons « pousser les sélections »!

$\Pi_{cname}((\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch) \bowtie \sigma_{balance > 1000}(account)) \bowtie depositor)$

Exemple

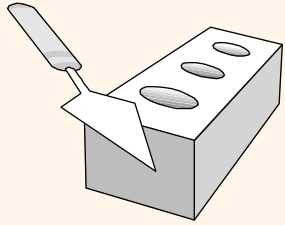

Π customer-name
σ branch-city=Brooklyn
∧ balance < 1000
⋈
branch
⋈
account
depositor
(a) Initial Expression Tree
Π customer-name
⋈
⋈
depositor
σ branch-city=Brooklyn
σ balance < 1000
branch
account
(b) Tree After Multiple Transformations

Choix de plans et optimisation

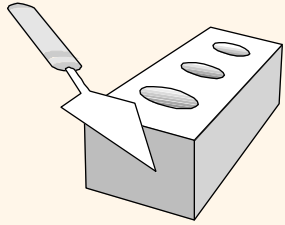


- Transformation / réécriture d'une requête :
  - Pour une expression A.R., trouver une expression équivalente qui peut être évaluée de manière plus efficace
  - Petite divergence entre SQL et A.R. « d'origine » : SQL accepte une sémantique « multi-set » / « bag » (doublons parmi les tuples) ; deux expressions R.A. sont considérée équivalentes si elle génèrent les mêmes résultats (sémantique multi-set) pour toute(s) instance(s) d'entrée
- Choix des algorithmes pour les opérateurs:
  - Basé sur des statistiques de la base de données contraintes de mémoire et index disponibles

Requêtes mono-relation

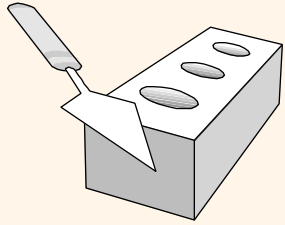


- Un chemin d'accès (access path) est une méthode pour accéder aux tuples d'une relation:
  - Scan, ou index qui correspond à / « marche avec » (matches) une sélection dans la requête ; plusieurs index peuvent être utilisés avec intersection des résultats !
- Un index de type arbre *matches* (une conjonction) des conditions qui ne concernent qu'un *préfixe de la clé de l'index*
  - Exemple: B+ Tree sur *<a, b, c>* matches la sélection *a=5 AND b=3*, et *a=5 AND b>6*, mais ne marche pas pour *b=3*.
- Un hash index *matches* (une conjonction) des conditions qui comprend une *égalité attribute = value* pour *tous les attributs dans la clé de l'index*
  - Exemple: Hash index sur *<a, b, c>* matches la sélection *a=5 AND b=3 AND c=5, mais ne marche pas pour b=3, ou a=5 AND b=3, ou a>5 AND b=3 AND c=5*.



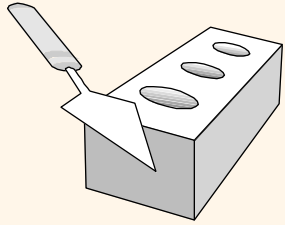
Requêtes mono-relation

- Les requêtes mono-relation combinent des sélections et des projections; pour évaluer de telles requêtes:
  - On considère chaque chemin d'accès possible (scan fichier / index / plusieurs index + intersection), et on choisit celui qui a le coût estimé le plus bas
  - Les différentes opérations complémentaires (projection, sélections restantes) sont en général directement appliquées sur les tuples obtenus (exemple: si un index est utilisé pour une sélection, on applique la projection pour tous les tuples obtenus via l'index…).
  - De manière générale, la projection ne va pas éliminer les doublons (sauf si DISTINCT).



Requêtes multi-relation

- Utilisation des règles d'équivalence pour générer des formulations alternatives
- Optimisation appliquée très souvent : « pousser les séléctions et les projections » pour limiter le nombre et la taille des tuples qui participent à une jointure
- Essentiel pour les requêtes multi-relations: l'ordonnancement des jointures = join (re) ordering!
  - Intuition : comme $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$, si $r_1 \bowtie r_2$ est "grand" et $r_2 \bowtie r_3$ est "petit", choisir la deuxième version peut s'avérer utile !

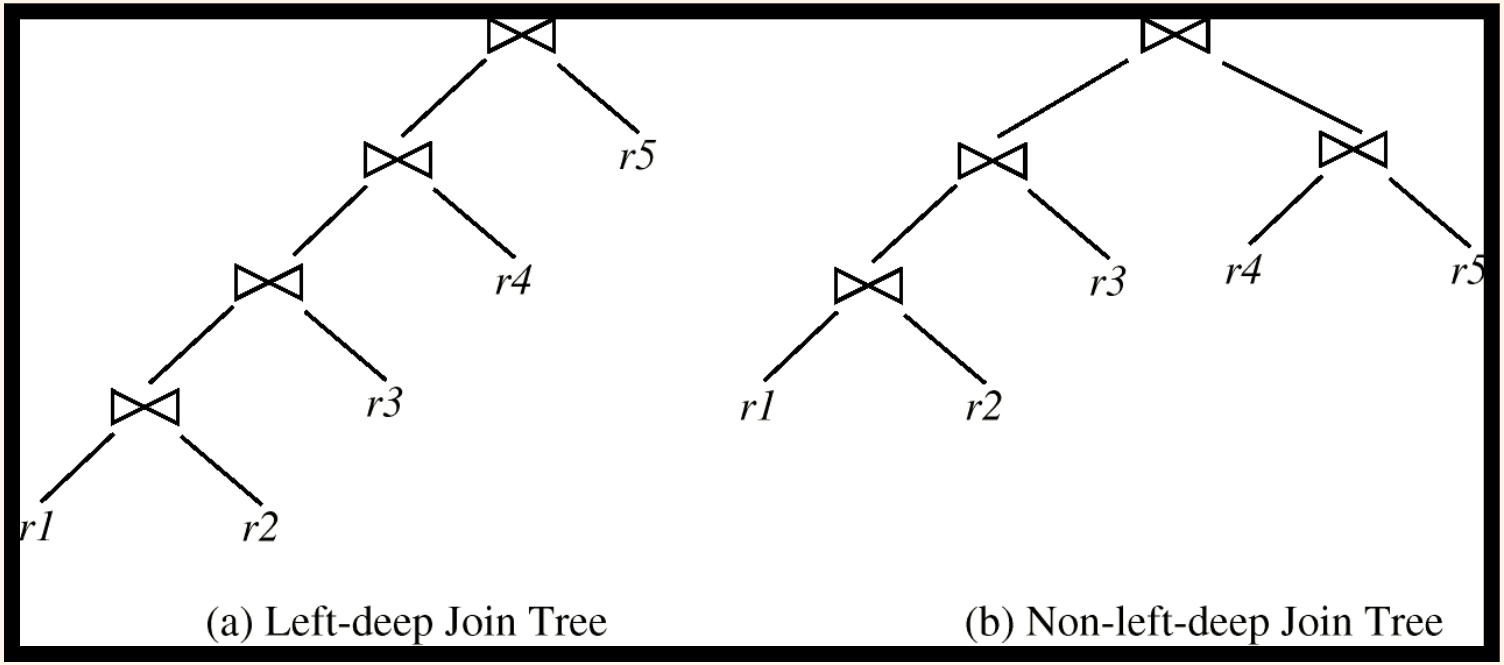


Join ordering

- Essentiel pour les requêtes multi-relations: l'ordonnancement des jointures = join (re) ordering!
  - Exploite la commutativité et l'associativité des jointures
  - Cet ordonnancement a un grand impact sur la taille des résultats intermédiaires et donc sur la performance globale
- Problème : nombre énorme d'alternatives possibles !
  - Jointure naturelle de R, S, T et W ?
- Décision fondamentale de System R : considérer uniquement les arbres de jointure « left deep » (left deep join trees)

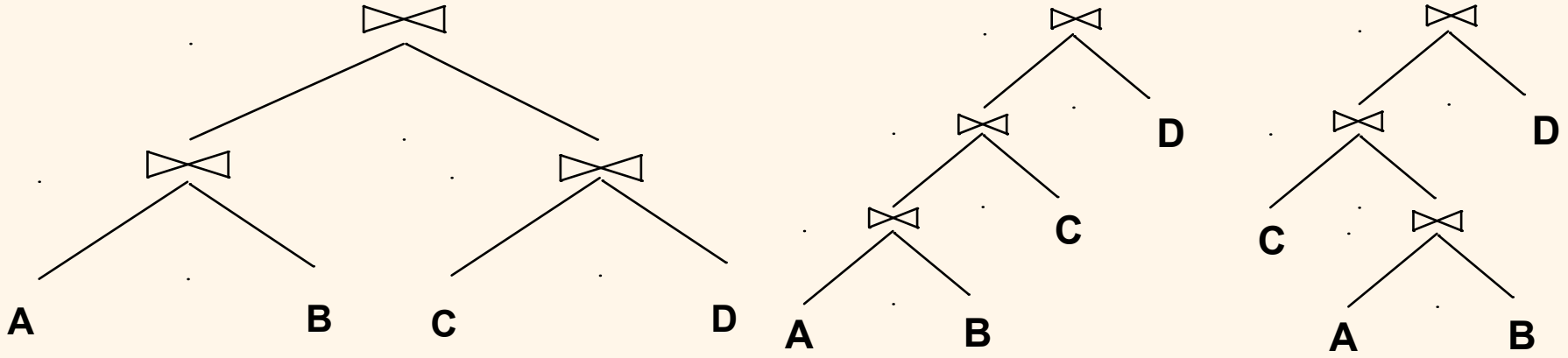
Left-deep join trees

- Dans les **left-deep join trees,** l'"input à droite" (relation interne) pour chaque jointure est une relation et non pas le résultat d'une jointure intermédiaire

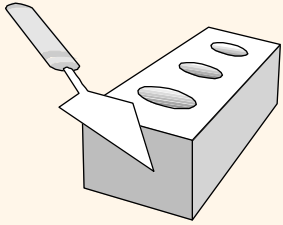


Left-deep join trees

- Décision fondamentale de System R: ne considérer que les *left-deep join trees* :
  - *Cela restreint l'espace de recherche des plans.*
  - *De plus, cela nous permet très souvent (mais pas toujours!) de bénéficier du pipelining.*

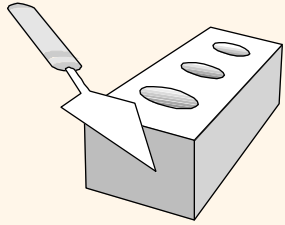


Trouver le meilleur plan de type left-deep join tree



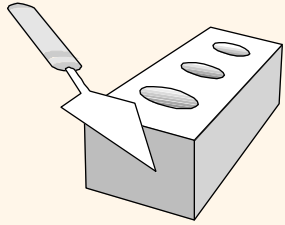
- Énumérer les plans en utilisant N passes (pour N relations)
  - La passe i va calculer les meilleurs plans pour tous les sous-ensembles de i relations; exemple pour 4 relations: la passe 3 va calculer les meilleurs plans pour joindre R1, R2 et R3; R1, R3 et R4; R2, R3 et R4
- Passe 1: Trouver le meilleur plan « à une relation » pour chaque relation R
  - Nous identifions les sélections qui ne portent que sur R ; elles seront considérées dans le choix du access path pour R
  - Nous retenons le plan le moins cher *pour chaque ordonnancement possible des tuples* !
    - Si index sur l'attribut A, il nous donnera les tuples triés par A
    - Donc si index sur A1 et index sur A2, nous retenons les deux plans (même si celui avec A1 est moins cher).

Trouver le meilleur plan de type left-deep join tree



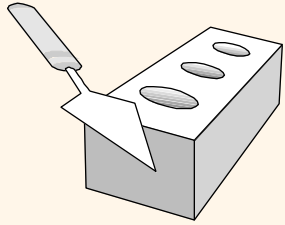
- Énumérer les plans en utilisant N passes (pour N relations)
  - La passe i va calculer les meilleurs plans pour tous les sous-ensembles de i relations; exemple pour 4 relations: la passe 3 va calculer les meilleurs plans pour joindre R1, R2 et R3; R1, R3 et R4; R2, R3 et R4
- Passe 2: Trouver le meilleur plan “à deux relations” pour chaque sous-ensemble de deux relations, en considérant chaque relation S à son tour comme relation interne et en utilisant pour la relation externe R un des plans construits dans la passe 1.
  - Nous considérons les divers algorithmes de jointure et pour chacun le meilleur access path pour S; les sélections qui ne portent que sur S peuvent être prises en compte à ce point pour le choix du access path de S
    - Exemple: si block nested loops, passer par un index pour “filtrer” suivant une sélection existante peut s’avérer intéressant
  - Dans le cas ou le pipelining n’est pas possible (hash join, sort-merge join où les inputs ne sont pas déjà triés….), nous devons prendre en compte le coût de la matérialisation

Trouver le meilleur plan de type left-deep join tree



- Énumérer les plans en utilisant N passes (pour N relations)
  - La passe i va calculer les meilleurs plans pour tous les sous-ensembles de i relations; exemple pour 4 relations: la passe 3 va calculer les meilleurs plans pour joindre R1, R2 et R3; R1, R3 et R4; R2, R3 et R4
    - Passe 1: trouver le meilleur plan à une relation
    - Passe 2: trouver les meilleurs plans à deux relations
    - Passe N: trouver les meilleurs plans à N relations
- Pour chaque sous-ensemble de relations, retenir le meilleur plan pour chaque ordonnancement possible des tuples
  - En réalité, que les meilleurs pour les ordonnancements « intéressants » (= qui peuvent servir plus tard! Ex: si la colonne est impliquée dans une jointure, pour faire du sort-merge…)
- Pour le coût: coût des algos + statistiques / estimation de cardinalité du résultat….
- Optimisation fréquente: “pousser” les sélections et les projections pour qu’elle soient appliquée aussi tôt que possible.

Trouver le meilleur plan de type left-deep join tree



- Approche par « passes » = programmation dynamique / « memoization » : stocker le résultat pour chaque ensemble de relations, pour ne pas devoir le re-calculer lorsqu'on considère un sur-ensemble de cet ensemble-là
  - Exemple, quand on veut calculer les meilleur plans pour R1;R2;R5;R8 et respectivement pour R1;R2;R5;R10 on va utiliser le résultat déjà calculé pour R1;R2;R5
- Complexité pour l'énumération des left-deep join trees: 2^N * N
    - à chaque passe i, on considère chaque relation avec chaque sous-ensemble de (i-1) relations; il y a 2^N sous-ensembles en tout…
- → coût exponentiel, vite prohibitif si beaucoup de jointures; diverses approches heuristiques / raccourcis …
  - l'optimisation des requêtes reste un sujet essentiel pour les Bases de Données (travaux de recherche + implémentations dans les SGBDs …)