TD 7 : Opérateurs relationnels

corrigé

Partie 1 : Algorithmes des opérateurs relationnels

Rappels et précisions sur le cours

On veut joindre deux relations R et S (non triées). R est plus petite que S

|R| = Nbre de page sur disque de R, ||R|| = Nbre de tuples de R, M = Nbre de page mémoire disponibles (on suppose que 3 pages mémoire sont allouées pour les I/O

Algorithme de jointure par sort-merge (SMJ)

Condition sur la mémoire	**Coût en I/O**
M > |R|+|S|	Cout = |R|+|S|
|R|+|S| > M > sqr(|R|+|S|)	Cout = 3|R|+3|S|

Algorithme de jointure par grace hash join (GHJ)

Condition sur la mémoire	**Coût en I/O**
M > |R|	Cout = |R|+|S|
M > sqr(|R|)	Cout = 3|R|+3|S|

Question 1

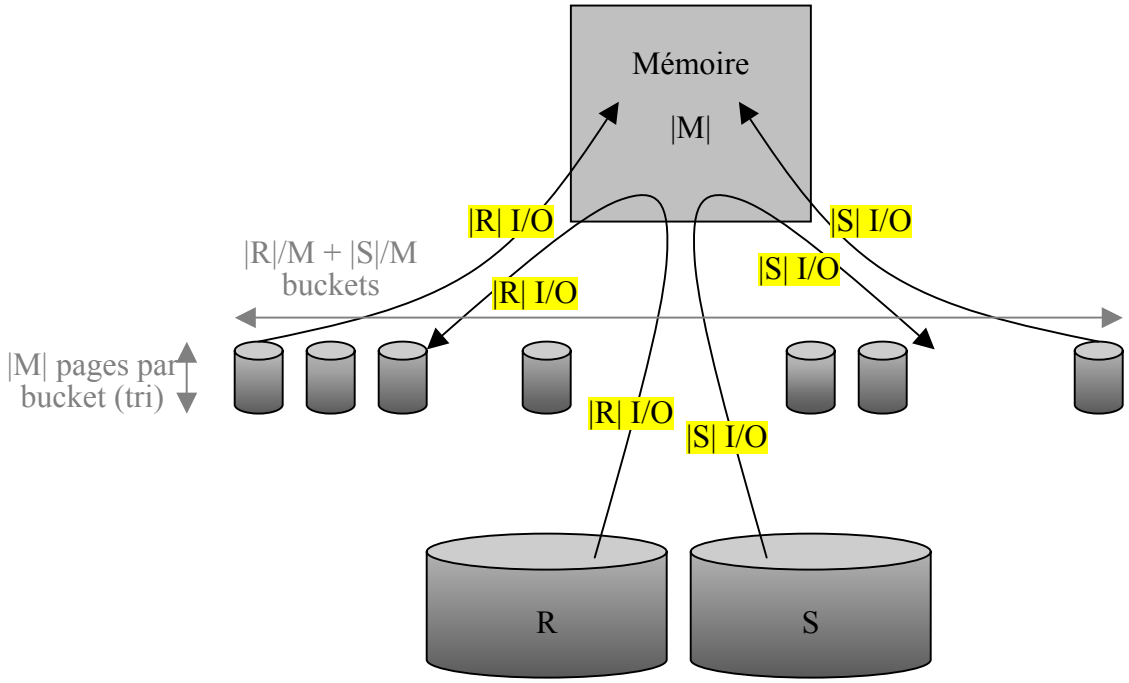
Retrouver d'après les algorithmes vus en cours les formules de coût ainsi que les conditions limites sur la mémoire (en supposant que M n'inclut pas 2 ou 3 buffers pour les I/O).

SMJ

- M> |R|+|S| : Lecture de R, tri de R en mémoire sur l'attribut de jointure, idem pour S en mémoire, merge des pages triées en mémoire. On a donc |R|+|S| entrée sorties disques pour lire les 2 relations. (NB : l'écriture du résultat n'est jamais comptabilisées dans les coûts des opérateurs).

- |R|+|S| > M > sqr(|R|+|S|) : Les deux relations ne tiennent pas en mémoire. L'algorithme est alors le suivant. Lecture de R en mémoire par paquets de M pages et trie du paquet en mémoire sur l'attribut de jointure. Ce paquet est écrit sur disque (s'appelle en anglais un bucket). Le procédé est réitéré |R|/M fois jusqu'à avoir consommé et trié tout R. On obtient |R|/M paquets de tuples de R triés. Le nombre d'entrées sorties est donc ici de 2|R|, pour lire R et écrire les paquets triés de R. Le même processus est appliqué à S, correspondant à 2|S| entrées sorties. On obtient alors |R|/M+|S|/M paquets de taille M. Il faut ensuite joindre les tuples. Pour cela, on effectue la phase de merge. On a

besoin d'au moins une page mémoire par paquet. La première page de chaque paquet est alors remontée en mémoire. Les tuples joignant sont produits dans le resultat. Quand on sait qu'un page d'un paquet contient des tuples qui ne peuvent plus joindre, on la remplace par la page suivante du paquet duquel elle provient. Ainsi, chaque page de chaque paquet est lue, générant |R|+|S| entrées sorties. Le nombre de buckets ne peut excéder M si on veut pouvoir faire le merge. La mémoire nécessaire pour effectuer ce traitement est d'au moins une page par paquet. Donc il faut que M > |R|/M+|S|/M d'ou le résultat du tableau.



GHJ :

- M> |R|+|S| : Lecture de R en mémoire, hachage de R, lecture et hachage de S en mémoire, jointure des paquets correspondants de hash en mémoire. On a donc |R|+|S| entrée sorties.

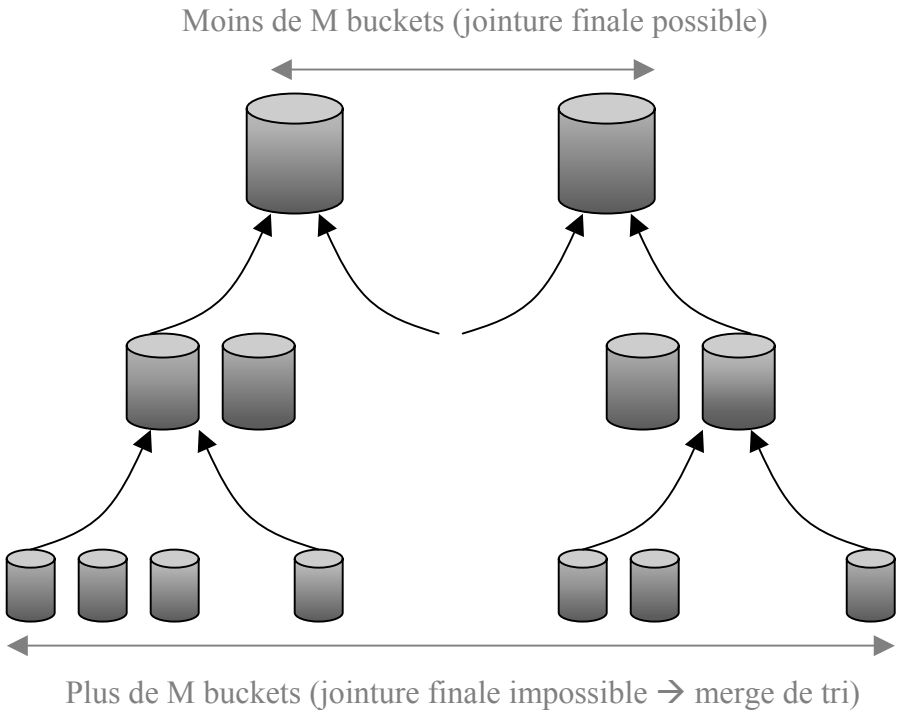
- M> sqr(|R|) : La relation R ne tient pas en mémoire. L'algorithme est alors le suivant. Lecture de R en mémoire par paquets de M et fabrication de N paquets de tuples de R hachés sur l'attribut de jointure (N est le nombre de valeurs de hash produites par la fonction de hachage), ceci jusqu'à avoir consommé tout R. Le nombre de buckets de R ne peut excéder M pour avoir au moins une page mémoire pour chaque entrée de la table de hachage. Le nombre d'entrées sorties est donc ici de 2|R|. Le même processus est appliqué à S, générant à 2|S| entrées sorties. Pour joindre les tuples, il faut lire un paquet haché entier de R dans la mémoire, et faire passer page par page le paquet haché correspondant de S. Pour tenir en mémoire, le paquet haché de R doit être inférieur à M. On a au plus M paquets, et en moyenne |R|/M pages par paquet. La mémoire nécessaire pour effectuer ce traitement est donc |R|/M < M d'où le résultat du tableau.

Question 2

Proposer une stratégie d'exécution du SMJ et du GHJ quand la mémoire est plus petite que sqr(|R|+|S|) (SMJ) ou que sqr(|R|) (GHJ)

SMJ : on peut faire plusieurs niveaux de runs. On augmente alors la taille des paquets et on réduit leur nombre. Ce ci peut se faire en mergeant par tri plusieurs paquets de R (on fait de plus gros paquets trié avec les

tuples des paquets initiaux) et de S. On obtient alors 3|R|+3|S| I/O (1 niveau de run), 5R|+5|S| (2 niveaux de runs), 7|R|+7|S| (3 niveaux de runs).



GHJ : On hache plus finement (autre fonction de hachage) les paquets de R dont la taille est supérieure à celle de la mémoire. (Pb : doit-on aussi rehacher S ?)

Question 3

En supposant que pour lire n pages consécutives sur le disque, il faut 10ms + n*0,2ms, comparer le Block Nested Loop Join (BNLJ), le GHJ et le SMJ. Pour le SMJ et le GHJ, on se place dans le cas ou on dispose d'une mémoire est plus petite que |R|+|S| et que |R| respectivement.

BNLJ : Lit un bloc de R, puis fait passer tout S page par page. Fait ceci R/M fois. On obtient donc : |R|/M (10 + M x 0,2ms + R|/M (10ms + |S| x 0,2ms) = temps de lire R plus temps de lire S.

SMJ : On obtient 2 x |R|/M(10ms + M x 0,2ms) + 2 x |S|/M (10ms + M x 0,2ms) + (|R|+|S|) (10ms + 0,2ms) = lecture/écriture séquentielle par blocs de R et S + lecture aléatoire des pages de R et S

GHJ : On obtient |R|/M(10ms + M x 0,2ms) + |S|/M (10ms + M x 0,2ms) + (|R|+|S|) (10ms + 0,2ms) = lecture séquentielle par blocs de R et S + lecture aléatoire des pages de R et S

Ref : « Seeking the truth about adhoc joins »

Question 4

En vous basant sur l'algorithme de Hybrid Hash Join vu en cours, proposez une stratégie adaptative d'exécution par hachage permettant de profiter du maximum de la mémoire disponible mais **sans connaître à priori la taille de R** (pour simplifier, on supposera que M > sqr(|R|) et M<|R|). Evaluez le coût en I/O pour M = R/2, M = R/3.

On peut décider de commencer à hacher R en mémoire comme si tout allait bien se passer (M > |R|), puis dès que l'on manque de mémoire, on met sur disque certains paquets de hash (en partant par la fin). Ces valeurs de hash se retrouveront maintenant dans un même paquet. Ensuite on lit S et on joint avec les paquets de R se

trouvant en mémoire. Les tuples de S qui joignent avec un paquet de R sur disque sont écrits sur disque sous forme de paquet de hash. Enfin on relit les paquets de R et S sur disque pour produire les resulats manquants.

Avec M=|R|/2, on écrit la moitié de |R|. A la fin, il reste la moitié des paquets de hash en mémoire, et sur disques les autres paquets contenant la moitié des tuples de R (|R|/2 entrées sorties). On fait passer S contre la table de hash page par page (|S| entrées sorties disque). On produit les résultats correspondants aux tuples de S joignant avec les paquets de R en mémoire et on fait écrit sur disque (sous forme de paquets de hash) les tuples de S correspondant aux paquets de R sur disque. Ceci génère |S|/2 entrées sorties. Enfin, on relit les paquets de R et de S sur disques, ce qui génère |R|/2+|S|/2 entrées sorties. Au total, il y a donc 2 (|R|+|S|) entrées sorties.

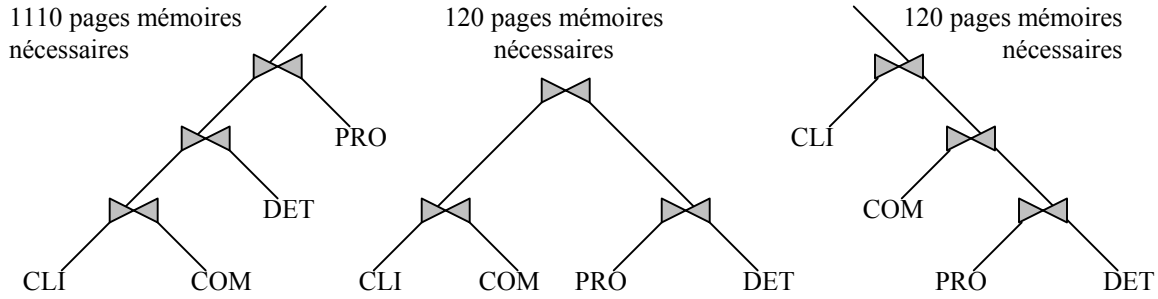
Avec M=|R|/3, on obtient |R| + 2/3 |R| + |S| + 2/3|S| + 2/3 |R| + 2/3|S| = 7/3 (|R|+|S|) entrées sorties.

Partie 2 : Optimisation et gestion mémoire

Question 5

Soit 4 relations CLI (client), COM (commande), DET (détails) et PRO (Produits) ayant respectivement les tailles sur disque suivantes : 10, 100, 1000, 10 pages.

En vous basant sur des algorithmes de jointure par hashage, proposez au moins 1 arbre droit, 1 arbre gauche et 1 arbre bushy permettant de joindre les 4 relations (essayez de trouver les meilleurs arbres).



Pour l'arbre droit, proposez la meilleure allocation de la mémoire et calculez le coût en considérant uniquement les I/O (avec la formule de coût donnée plus haut). Vous considérerez successivement une mémoire disponible (hors buffers d'I/O) de 150 pages, 50 pages.

Avec 150 pages, tout tient en mémoire. On a donc pour l'arbre droit un coût de |CLI| + |COM| + |DET| + |PRO|. Avec 50 pages, on choisit de donner toute la mémoire demandée par les itérateurs qui en demandent le moins et ce qu'il reste aux autres (algorithme « max to min »). Pour l'arbre droit, on donne donc 10, 30, 10 pages respectivement de bas en haut aux itérateurs. Le coût est alors de la même manière |CLI| + |COM| + |DET| + |PRO| + 1000 + 1000 + 2x100 (grace hash join) + 70x2 (hachage des commandes) ou |CLI| + |COM| + |DET| + |PRO| + 700x2 (hybride hash pour la jointure des details avec les produits).

Partie 3 : Modèle itérateur et mémoire très limitée (exécution sur carte à puce)

On se place maintenant dans un cas extrême où la mémoire disponible est seulement de quelques octets, par contre, les données sont stockées sur une mémoire électronique, le coût de lecture est donc similaire à une lecture en mémoire.

Question 6

Trouvez des algorithmes et exprimez les avec le formalisme du modèle itérateur pour réaliser les opérations suivantes : Sélection, projection, jointure.

Indiquez le minimum de mémoire nécessaire.

Les opérateurs de sélection, projection travaillent tuple à tuple, ils ne consomment donc pas de RAM. La jointure peut être effectuée par nested loop, il n'y a pas de consommation mémoire (ou 1 pointeur par relation).

Question 7

Peut-on réaliser des tris ou des calculs d'agrégats ? si oui, comment ?

On peut par exemple parcourir entièrement l'entrée de l'opérateur pour trouver le tuple partageant la plus petite clé de tri, puis refaire un parcours complet de l'entrée pour délivrer tous les tuples partageant cette clé, et ainsi de suite jusqu'à avoir produit tous les tuples. On peut donc de cette manière trier des tuples.

Pour calculer des agrégats, on procède de même en calculant l'agrégat pour les tuples partageant la clé.

Question 8

Vous disposez maintenant de quelques KO de mémoire RAM. Comment optimiser les jointures ?

On fait un bloc nested loop…

Question 9

Et les tris ou calcul d'agrégats ?

On bufferise plusieurs valeurs de tri ou d'agrégats à chaque passe sur les données. Chaque passe permet ainsi de traiter plusieurs clés de tri ou d'agrégat.