TEHARI Ayoub **06-02-2020**

L3-informatique_inf1

Compte Rendue 2 : utilisation de tables de hachage ”Hash join”

Introduction :

j’ai commencé par implémenter la version du HashJoin, en replissant la table de hachage avec la première relation, en prenant comme key de hachage le champ en commun, ensuite pour chaque n-uplet de la deuxième relation on calcul le code de hashage correspondant, également en prenant le champ en commun comme key, ensuite si ça correspond pas, il n’y aura pas besoin d’aller plus loin, sinon on compare le champ en concret avec la méthode equals,

Deuxièmement, la méthode HashSelect permet d’utiliser la table de hashage, d’un coté pour éliminer les doublant, et en plus, de regrouper les 1-uplets obtenues,

Finalement, pour l'opérateur de soustraction, on a la méthode `NestedMinus() et HashMinus(),`

NestedMinus : cette méthode utilise un algorithme naïf qui parcours pour chaque n-uplet de la première relation (plus ou moins) toute la deuxième relation tant qu’elle n’as pas trouvé le n-uplet donné, Alors que la méthode HashMinus, se contente de remplir la table de hashage progressivement, puis en testant pour chaque n-uplet donné de la deuxième relation, s’il fait ou pas partie de notre table de hashage, par le biais de la clé.

*j’ai ajouté pour la raison des graphes, quelque fonctions pour le calcul de la taille de l’entrée, qui sont décrites dans l’annexe.

Exercice 2. Coût de l'algorithme naïf

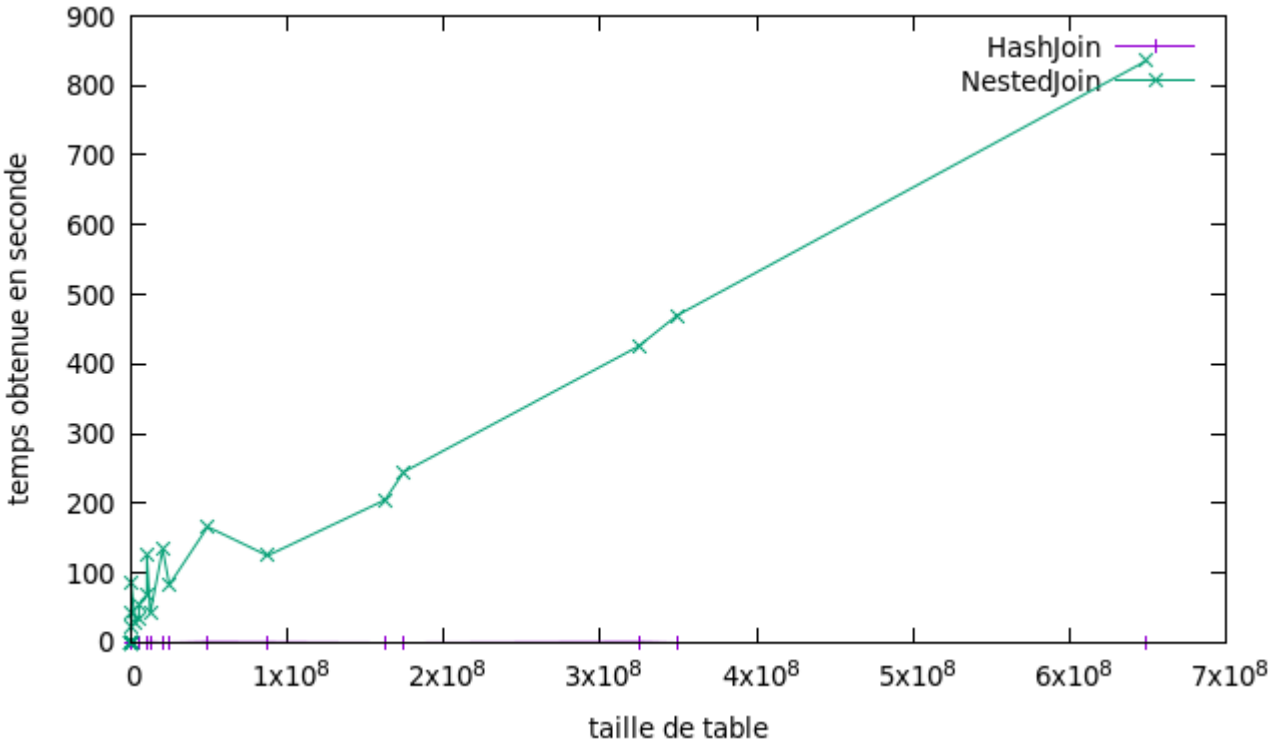
Variables :	Fichier f1, f2; Fichier sortie; chaine ligne1, ligne2; chiane champ1, champ2;

Algo :	tantque (nonFinFichier(f1)) {
	ligne1=f1.ligne();
	champ1=deuxiemeChamp(ligne1);
	tantque (nonFinFichier(f2)){
	ligne2=f2.ligne();
	champ2=premierChamp(ligne2);
	if (champ1 == champ2){
	sortie.println(ligne1 + “\t” + ligne2);
	}
	}

	recharger(f2); }

La complexité est de l'ordre de O(n*m), tel que:
n : le nombre de n-uplets du premier fichier
m : le nombre de n-uplets du deuxième fichier
*car il existe deux boucle "while" :
la boucle intérieure s'exécute m fois et la boucle extérieure s'exécute n fois.

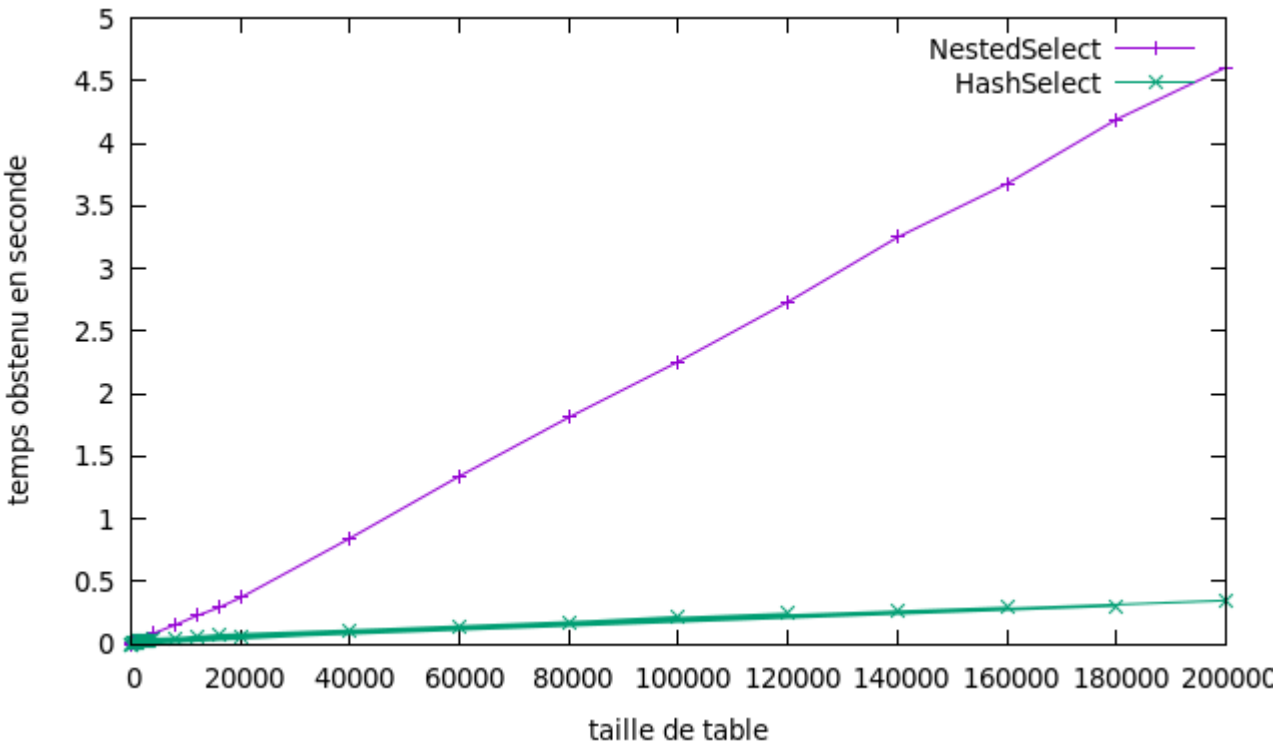
Graphes de Hashjoin VS NestedJoin:



On voit que lorsque $n < 10^8$, la fonction correspondante à l'algorithme naïf se comporte un peu bizarrement, mais lorsque $n >= 10^8$ ça devient linéaire, en revanche, la fonction qui représente l'algorithme qui utilise les tables de hashage, est presque instantané (en terme de grandeur : temps au pire cas N=649000000 ;0.589556826 seconde) et ceci quelque soit la taille de l'entrée ($< 7*10^8$), on peut dire que la complexité dans ce cas est de l'ordre O(1).

Graphes de HashSelect VS NestedSelect :

On voit clairement dans ce graphe, que la fonction correspondante à l'algorithme naïf est de l'ordre O(n) où n est la taille de l'entrée, en revanche, la fonction qui représente l'algorithme qui utilise les tables de hashage, est majoré par une certaine valeur (au alentour de 0,5 seconde) et que la complexité est donc dans ce cas de l'ordre O(1).



Graphes de HashSelect VS NestedSelect :

On voit clairement dans ce graphe, que la fonction correspondante à l'algorithme naïf est de l'ordre O(n) où n est la taille de l'entrée, en revanche, la fonction qui représente l'algorithme qui utilise les tables de hashage, est majoré par une certaine valeur (au alentour de 0,5 seconde) et que la complexité est donc dans ce cas de l'ordre O(1).

Conclusion:

Pour conclure, on peut dire que l'utilisation des table de hashage permet d'économiser beaucoup de temps dans la recherche et la consultation, alors qu'en procédant à l'aide des algorithmes naïfs, on risquerait d'avoir beaucoup attendre.

La force des table de hashage est dans son pouvoir à indexer les valeurs de l'entré, ce qui explique son efficacité par rapport à l'algorithme naïf.

Le passage au tables de hashage permet de rendre la complexité de l'ordre O(1), au lieu de O(n) où n est la taille de l'entrée, dans le cas de l'algorithme naïf.

Annexe:

La fonction « countLines_f1 » permet de retourner la taille de l'entrée f1,
La fonction « countLines_f2 » permet de retourner la taille de l'entrée f2,
La fonction «countLines» permet de retourner la taille des entrées, càd T(f2) * T(f1),