Université de Lille 1 SCIENCES ET TECHNOLOGIES

TP – Vers l'optimisation logique

Algorithmes des bases de données
Master Informatique – M1S2
2016–2017

Vous devez finir le TP3 avant de commencer ce TP.

L'optimisation logique d'une requête consiste à considérer plusieurs requêtes équivalentes à la requête d'origine et choisir celle de coût estimé minimal.

On ne peut pas explorer toutes les requêtes équivalentes à la requête d'origine, car leur nombre est trop grand et l'optimisation risque de prendre plus de temps que l'évaluation naïve de la requête. C'est pourquoi on applique des méthodes approchées.

Nous avons déjà vu qu'une première optimisation qui marche quasiment toujours consiste à pousser les sélections vers les feuilles. De manière similaire, on peut pousser les projections vers les feuilles, ou introduire des projections supplémentaires sous les jointures, dans le but de diminuer la taille des données manipulées par les jointures.

Dans ce TP nous vous guidons vers une méthode d'optimisation qui consiste à réordonner et réorganiser les jointures : c'est l'algorithme de *énumération gloutonne ascendante*.

Q 1. Soit la requête $R_1 \bowtie R_2 \bowtie R_3$. Rappelons que la jointure est un opérateur associatif et commutatif (comme par exemple l'addition). Donner tous les arbres d'algèbre relationnelle qui sont équivalents à cette requête.

Astuce : si vous deviez le faire pour l'expression arithmétique $1 + 2 + 3$, cela correspondrait à toutes les expressions équivalentes. Par exemple, $(1 + 2) + 3$, $1 + (2 + 3)$, $(2 + 1) + 3$, $2 + (1 + 3)$, etc.

Q 2. Combien il y a d'arbres différents pour la requête $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$.

L'algorithme d'énumération gloutonne ascendante. Il s'agit d'énumérer des variantes d'une requête qui sont obtenues par arrangement des jointures. Comme vous avez vu, le nombre d'arrangements différents croit très vite, c'est pourquoi cet algorithme ne va pas énumérer tous les arrangements possibles, mais va exclure certains arrangements suite à des optimisations locales. C'est pourquoi c'est un algorithme *glouton*.

L'algorithme commence par chercher de bons arrangements pour des parties de la requête et ensuite les combine en construisant l'arbre final des feuilles jusque la racine. C'est pourquoi c'est un algorithme *ascendant*.

L'algorithme utilise la programmation dynamique. Nous illustrons l'algorithme sur un exemple : $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$.

1. D'abord, énumérer tous les arrangements possibles impliquant 2 requêtes R_i, R_j. Pour chaque couple R_i, R_j, trouver le meilleur arbre et garder uniquement celui-ci. (Pour l'instant, le meilleur arbre sera ou bien $R_i \bowtie R_j$ ou bien $R_j \bowtie R_i$, mais dans la suite vous allez voir qu'on peut aussi utiliser différents algorithmes de jointure.)
2. Ensuite, considérer les arrangements possibles de 3 requêtes R_i, R_j, R_k. Pour chaque triplet R_i, R_j, R_k, trouver le meilleur arbre et garder uniquement celui-ci. Un tel arbre est obtenu en combinant des arbres pour lesquels le meilleur arbre est déjà trouvé. Ainsi, pour trois requêtes il y a 6 cas possibles.
   — joindre d'abord R_i, R_j, puis le résultat avec R_k. Ceci peut se faire avec R_k à droite ou R_k à gauche ;
   — joindre d'abord R_i, R_k, puis le résultat avec R_j. Ceci peut se faire avec R_j à droite ou R_j à gauche ;
   — de manière similaire, joindre d'abord R_j, R_k, puis le résultat avec R_j, et sa symétrique.
   Dans chacun de ces cas, on ne considère pas de réarrangements des requêtes intermédiaires. Ainsi, dans le cas où on joint d'abord R_i, R_j puis le résultat avec R_k, nous avons déjà décidé la manière de calculer la jointure de R_i avec R_j lors de l'étape précédente et cette décision ne sera pas mise en cause.
3. Finalement, considérer tous les arrangements possibles des 4 requêtes, et garder le meilleur. Comme précédemment, ces arrangements utilisent des sous-requêtes pour lesquelles un arrangement a déjà été choisi. Les différentes manières de joindre quatre relations sont :
   — joindre d'abord les relations 2 à 2, puis les résultats intermédiaires ensemble ;
   — joindre d'abord 3 relations, puis le résultat intermédiaire avec la quatrième.

Dans le cas de la jointure de n relations, ce procédé doit continuer jusque explorer la jointure de tous les éléments.

Q 3. Cette méthode ne permet pas de toujours obtenir l'arbre de coût minimal. Expliquez pourquoi. Astuce : en considérant une jointure de 3 relations, montrer qu'au moins un arrangement possible ne sera jamais considéré par l'algorithme, et donc ne sera pas trouvé si c'était l'arrangement optimal.

Implémentation de l'algorithme Vous devez maintenant implémenter l'algorithme ci-dessus. La fonction de coût que nous allons utiliser pour l'instant est celle de la jointure à boucles imbriquées calculée en utilisant deux pages mémoire, et sans écrire sur disque les résultats intermédiaires.

Voici quelques conseils pour l'implémentation.

Supposons qu'on calcule l'arrangement pour une jointure de n relations. Les résultats intermédiaires que vous devez stocker correspondent au meilleur arrangement trouvé pour chaque sous-ensemble de $\{1, \ldots, n\}$. Vous pouvez représenter un tel sous-ensemble par un `BitSet` en java. Ainsi, le meilleur résultat intermédiaire peut être stocké dans une `Map` dont les clés sont des `BitSet`. Le meilleur résultat intermédiaire est un arbre de type `RAQuery`.

Vous devez également stocker le coût calculé pour le meilleur résultat, par exemple dans une deuxième `Map`.

Pour calculer le résultat intermédiaire pour k relations, vous devez énumérer tous les k_1, k_2 t.q. $k = k_1 + k_2$, et ensuite toutes les jointures utilisant k_1 relations à gauche et k_2 relations à droite, ainsi que le cas symétrique.

Il peut pour cela être utile, en plus des `Map`s déjà mentionnées, vous donner un moyen d'accéder rapidement à tous les sous-ensembles de $\{1, \ldots, n\}$ à k éléments, pour un k donné.