Administration : Le traitement des requêtes

Anne-Cécile Caron

Master MIAGE - BDA

1er trimestre 2010-2011

Traitement d'une requête

- Analyse lexicale et syntaxique : construction d'un flux de tokens et vérification de la syntaxe de la requête
- Analyse sémantique et construction de l'arbre représentant l'expression algébrique de la requête
- La représentation algébrique est appelée *plan logique* de la requête. Il est réécrit afin de trouver un plan logique équivalent qui devrait prendre moins de temps à l'exécution
- A partir du plan logique, on génère un plan physique d'exécution. La encore, il faut trouver le plan le moins coûteux (estimation !)
- Exécution du plan physique par le moteur d'exécution des requêtes.

Plan logique

- Il existe certaines règles syntaxiques pour transformer le plan logique en un plan logique équivalent moins coûteux
- Le coût est estimé selon la taille des résultats (relations) intermédiaires

Exemple :

Tables
```
StarsIn(title, year, starName)
Movie(title, year, length, studioName, ...)
```

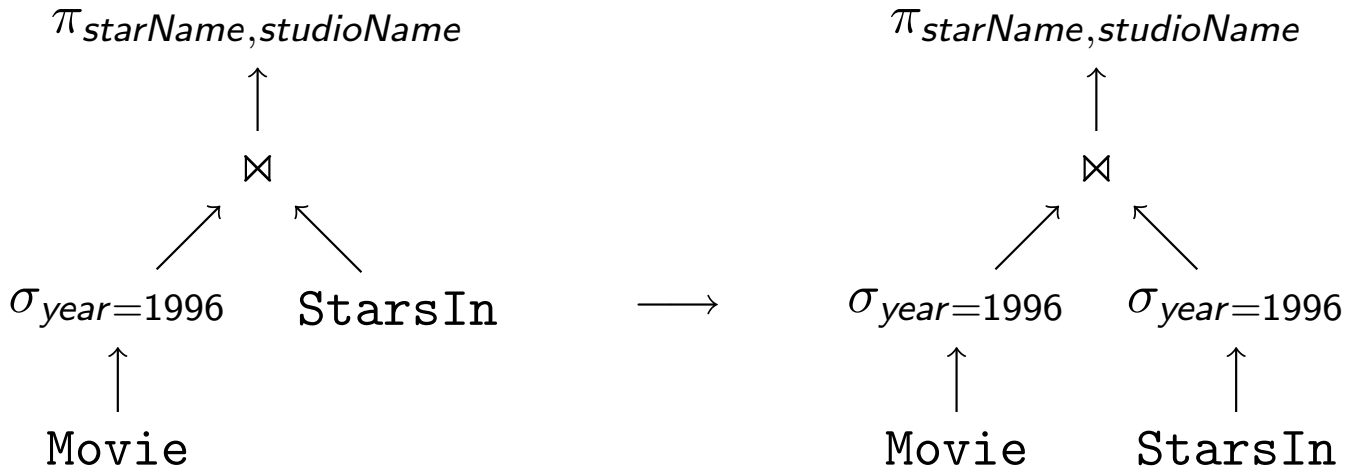
Vue
```
create view MovieOf1996 as
select * from Movie
where year = 1996
```

$\sigma_{year=1996}(Movie)$

Exemple

Requête :

```
select starName, studioName
from MovieOf1996 natural join StarsIn
```



Plan physique

- A partir du plan logique choisi, on construit un plan physique d'exécution.
- Il existe différents plans physiques :
  - pour les opérations associatives-commutatives ($\bowtie$, $\cup$, $\cap$), il faut choisir un ordre d'exécution, éventuellement grouper les opérations
  - Trouver un algorithme pour chaque opérateur du plan logique
  - Ajouter des opérateurs supplémentaires (tris, ...) qui n'apparaissent pas dans le plan logique
  - Définir comment on va passer les arguments d'un opérateur à l'autre (mémoire centrale, écritures disques intermédiaires, ...)
- Parmi tous ces plans, on choisi celui qui semble le moins coûteux ; le coût est essentiellement le nombre de lectures/écritures disque.

Estimation des coûts

On utilise des paramètres statistiques pour déterminer le coût de chaque opérateur et ainsi choisir un plan d'exécution parmi d'autres.

- M espace de la mémoire principale (nombre de blocs) utilisé par l'opérateur.
- $B(R)$ nombre de blocs nécessaires au stockage de R (on suppose que R est stocké dans des blocs qui ne contiennent que R)
- $T(R)$ nombre de tuples de R.
- $V(R, a)$ nombre de valeurs distinctes de la colonne a dans R. (généralisé en $V(R, [a_1, \ldots, a_n])$.

Implémentation d'un opérateur logique

Il existe plusieurs grandes classes d'algorithmes :

1. traitement séquentiel de la relation
2. tri d'une partie des données lues
3. construction d'une table de hashage à partir des données lues
4. utilisation des indexes existants

Ces algorithmes ont plusieurs degrés de coût en terme d'I/O

1. une seule lecture des données sur le disque (one-pass)
2. les données ne tiennent pas en mémoire centrale mais le traitement peut se faire en 2 lectures (2-passes = lecture-écriture-lecture)
3. algorithmes multi-passes : méthodes qui fonctionnent sans limite sur la taille des données.

Nous allons étudier en particulier des algorithmes de jointure

Algorithme de jointure en 1 passe

$R(X, Y) \bowtie S(Y, Z)$.
On suppose que S est la plus petite relation et qu'elle tient en mémoire principale.

1. On lit tous les tuples de S et on forme une structure de recherche (B-arbre, hashtable) en mémoire principale, de clef de recherche Y
2. Pour chaque bloc b de R, pour chaque tuple t de b, on trouve tous les tuples de S qui peuvent s'apparier avec t (même valeur de Y).

Algorithme en 1 passe.
Coût $B(R) + B(S)$

Algorithme de jointure : nested-loop

On suppose qu'aucune des deux relations ne tient en mémoire principale.

```
Pour chaque paquet de M-1 blocs de S faire
  lire ces blocs en mémoire principale
        et les organiser en structure de recherche (clef Y)
  Pour chaque bloc b de R faire
    lire b en memoire principale
    pour chaque tuple t de b faire
      trouver les tuples de S en mémoire principale
        qui peuvent s'apparier avec t
    finPour
  finPour
finPour
```

Algorithme en 1 passe pour S mais plusieurs passes pour R.
Coût $\frac{B(S)}{(M-1)} \times (M - 1 + B(R))$ de l'ordre de $\frac{B(S) \times B(R)}{M}$

Algorithme de jointure : merge-join

1. Créer des sous-listes triées de taille M utilisant Y comme clef de tri à la fois pour R et S. Cette étape nécessite une lecture et une écriture pour chaque bloc de données.
2. Prendre le premier bloc de chaque sous-liste dans un buffer (on suppose qu'il n'y a pas plus de M sous-listes)
3. Trouver y la plus petite valeur de Y parmi les premiers tuples non traités des sous-listes (blocs lus dans l'étape 2) et faire la jointure des tuples de R et S qui ont cette valeur $Y = y$. Quand un bloc est entièrement traité, on lit le bloc suivant de la sous-liste.

Algorithme en 2 passes, de coût $3(B(R) + B(S))$.

Algorithme de jointure : avec index

$R(X, Y) \bowtie S(Y, Z)$, tel que S a un index sur Y.

```
pour b bloc de R
  pour t tuple de b
    utiliser l'index pour accéder aux tuples de S
    qui ont la même valeur de Y que t.
  end loop
end loop
```

Coût : $B(R) + T(R) \times (T(S)/V(S, Y))$, sans tenir compte de la lecture de l'index.
Le coût de la lecture de l'index dépend du type d'index utilisé, et il est bien inférieur au coût de la lecture des données.
Si on dispose à la fois d'un index de type B-arbre sur Y pour R et S, on peut utiliser un "merge join" à partir des indexes.

Calcul de statistiques

- Le recalcul périodique des statistiques est de plus en plus courant dans les SGBD
- Ce calcul est effectué selon une période de temps (pas trop courte !) ou un nombre de mises-à-jour.
- L'administrateur peut demander explicitement le calcul de statistiques s'il constate que les plans d'exécution choisis ne paraissent pas les "meilleurs".
- Le choix du plan logique et du plan physique sont basés sur des **estimations**. Choisir le meilleur plan nécessiterait d'exécuter tous les plans !

Evaluation du coût sous Oracle

- On peut connaître le plan d'exécution choisi grâce à la commande `EXPLAIN PLAN`
- On peut obliger l'optimiseur à choisir un plan d'exécution plutôt qu'un autre, grâce à de nombreux "hints" précisées avec la requête.
- Pour choisir un plan d'exécution parmi plusieurs possibles, l'optimiseur utilise le mode `rule-based` ou `cost-based`
  - `rule-based` : l'optimiseur examine s'il peut utiliser des structures de stockage particulières (clusters, index, ...)
  - `cost-based` : idem mais il utilise en plus des statistiques contenues dans le dictionnaire des données sur les tables, index, clusters. Ces statistiques sont obtenues à l'aide de la commande `Analyze`.
- Depuis Oracle 10g, le mode par défaut est basé sur les coûts.

Explain Plan

```
EXPLAIN PLAN FOR
 SELECT *
 FROM   emp e, dept d
 WHERE  e.deptno = d.deptno AND e.ename  = 'SMITH';
```

```
-------------------------------------------------------------------------------
| Id  | Operation                     |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |     1 |    57 |     3 |
|   1 |  NESTED LOOPS                 |             |     1 |    57 |     3 |
|*  2 |   TABLE ACCESS FULL           | EMP         |     1 |    37 |     2 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT         |     1 |    20 |     1 |
|*  4 |    INDEX UNIQUE SCAN          | PK_DEPT     |     1 |       |       |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("E"."ENAME"='SMITH')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
```

Explain Plan

Interprétation du plan précédent :

```
pour b bloc de la table EMP
  pour t tuple de b tel que EMP.ENAME = 'SMITH'
    utiliser l'index PK_DEPT pour accéder aux tuples de DEPT
    qui ont la même valeur de DEPTNO que t.
  end loop
end loop
```

Quelques conseils pour terminer

illustration en TP

- Eviter `select *` : on a rarement besoin de toutes les colonnes.
- Eviter les `distinct` : enlever les doublons nécessite un tri. Pour la même raison, l'opérateur `union` est coûteux (on peut parfois le remplacer par `union all`).
- Eviter les requêtes qui ne peuvent pas exploiter les indexes : inégalité <>, comparaison avec `NULL` (`is null` et `is not null`), sous requête avec `not in`
- Attention aux conversions de types (cf début du TP, `a = 10` ou `a = '10'`)
- Eviter les opérateurs ensemblistes souvent coûteux (`union`, `intersect`, `minus`, `all`, `any`)