

## Plan

### Bases de données

Polytech Paris-Sud

Apprentis 4<sup>ème</sup> année

### Cours 4 : Optimisation des opérateurs

kn@lri.fr  
<http://www.lri.fr/~kn>

- 1 Rappels ✓
- 2 Stockage ✓
- 3 Indexation ✓
- 4 Optimisation des opérateurs
  - 4.1 Motivation
  - 4.2 Algorithmes de jointure
  - 4.3 Autres opérateurs

## Principe d'évaluation d'une requête

## Plan

1. *Parsing* de la requête
2. Traduction en arbre d'opérateurs de l'algèbre relationnelle ( $\pi, \sigma, \bowtie, \dots$ )
3. Optimisation :
  1. Génération de **plans d'évaluation** (en réordonnant les **opérations élémentaires**)
  2. Estimation du **coût** de chacun des plans (en fonction du **coût des opérations élémentaires**)
  3. Choix du plan le plus **efficace**
4. Évaluation du plan choisi
5. (Éventuellement mise à jour des statistiques)

Avant de s'intéresser à l'évaluation complète d'une requête, on étudie l'évaluation des opérateurs et leur coût respectifs

- 1 Rappels ✓
- 2 Stockage ✓
- 3 Indexation ✓
- 4 Optimisation des opérateurs
  - 4.1 Motivation ✓
  - 4.2 Algorithmes de jointure
  - 4.3 Autres opérateurs

## Jointure naturelle sur une colonne

Considérons :

```
SELECT *
FROM people,role
WHERE people.pid = role.pid;
```

Opération **fondamentale** utilisée par **toutes** les applications BD.

L'AR nous dit que  $R \bowtie S = \sigma_{(R \times T)}$ , mais c'est **très inefficace**, on veut optimiser ce cas!

On suppose dans la suite  $M$  enregistrements dans  $R$ ,  $P_R$  enregistrements/page,  $N$  enregistrement dans  $S$ ,  $P_S$  enregistrements/page.

On pose pour les exemples:  $M=1000$ ,  $N=500$ ,  $P_R=120$ ,  $P_S=100$

L'attribut commun est  $a$ .

Le coût est toujours le nombre d'E/S (en ignorant l'écriture du résultat)

## Jointure itérative simple

On effectue une double boucle imbriquée:

```
for each record r ∈ R do
  for each record s ∈ S do
    if r.a = s.a then res += r ⋈ s #jointure élémentaire de
    done #2 enregistrements
done
```

Pour chaque **enregistrement** de la relation externe ( $R$ ) on scanne entièrement la relation interne ( $S$ )

Coût:  $M + P_R * M * N$

Exemple:  $1000 + 120 * 1000 * 500 = 60\,001\,000$  E/S!

## Jointure itérative page à page

On effectue une double boucle imbriquée:

```
for each page P ∈ R do
  for each page Q ∈ S do
    res += P ⋈ Q #jointure entre 2 pages
  done # peut être faite de manière simple
done
```

Pour chaque **page** de la relation externe ( $R$ ) on scanne entièrement la relation interne ( $S$ )

Coût:  $M + M * N$

Exemple:  $1000 + 1000 * 500 = 501\,000$  E/S!

Optimisation: mettre la relation la plus petite à l'extérieur:  
 $500 + 500 * 1000 = 500\,500$

## Jointure itérative avec index

On effectue une double boucle imbriquée:

```
for each record r ∈ R do
  for each record s ∈ S where r.a = s.a do
    #l'index doit permettre un accès rapide à la colonne a
    res += r ⋈ s
  done
done
```

On exploite l'existence d'un index sur l'une des relation pour en faire la relation interne.

Coût:  $M + P_R * M * (\text{coût d'accès index} + \text{coût index} \mapsto \text{données})$

Plusieurs variantes: B+-tree/Hash-index, groupant/non-groupant,...

## Jointure par bloc (avec pages bufferisées)

On exploite le *buffer* (de taille  $B+2$  pages,  $B = 10$ ) de la manière suivante:

- 1 page du *buffer* pour l'écriture du résultat
- 1 page du *buffer* pour la relation interne (S)
- B pages du *buffer* pour la relation externe

```
for each block b of size B ∈ R do
  for each page Q ∈ S do
    res += b ⋈ Q #en utilisant la méthode simple
  done
done
```

Coût:  $M + N * \lceil M/B \rceil$

Exemple:  $1000 + 500 * 1000/10 = 51\ 000$

Variante: blocs sur R et S

## Jointure par tri-fusion 1/2

Idee: « l'intersection » de deux listes est aisée si les deux listes sont triées

```
sort R on a as Rs
sort S on a as Ss
r := Rs.next(); #On suppose R et S non vides
s := Ss.next(); #Sinon jointure vide directement
while Rs and Ss are not empty do
  while r.a != s.a do #avance jusqu'à trouver la même valeur
    if r.a < s.a do
      if Rs.hasNext() then r:= Rs.next() else finished
    done
    while s.a < r.a do
      if Ss.hasNext() then s:= Ss.next() else finished
    done
  done
  val := r.a #on prend la liste des enregistrements
  #ayant la même valeur d'attribut de jointure
  P, Q := empty pages
  while r.a = val do P += r; r:= Rs.next() done
  while s.a = val do Q += s; s:= Ss.next() done
  res += P ⋈ Q
done
```

## Jointure par tri & fusion 2/2

Coût:  $M \cdot \log_2 M + N \cdot \log_2 N + M + N$

Exemple:  $1000 * (1 + \log_2 1000) + 500 * (1 + \log_2 500) = 15492$

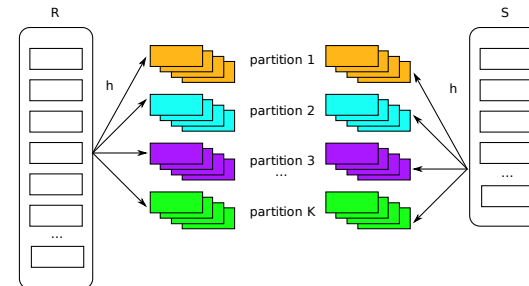
Le tri n'est pas toujours nécessaire:

- si on a un index de type B+-tree sur l'attribut de jointure (pour l'une des relations)
- si R ou S (ou les deux) sont déjà le résultat de tris (ORDER BY)

## Jointure par hachage

On choisit une fonction de hachage  $h$  et on partitionne R selon  $h(r.a)$  pour obtenir K partitions

On partitionne S selon  $h(s.a)$  pour obtenir K partitions



K choisi en fonction de la taille du *buffer*

Coût:  $2(M+N) + M+N$  (pourquoi ?)

## Jointures généralisées

- Égalité sur plusieurs attributs ( $R.a = S.a$  AND  $R.b = S.b$ ) :
  - Jointure itérative par index: on peut créer un index pour S sur (a,b) ou utiliser des indexes existants sur l'un ou l'autre
  - On peut aussi utiliser jointure par tri-fusion et hachage en utilisant (a,b) comme clé de tri/hachage
- conditions d'inégalité:
  - Pour les jointures par index, il faut un arbre B+ **groupant** (sinon sur-coût pour aller chercher les données)
  - Jointure par tri-fusion et hachage impossible
  - Jointure itérative par bloc est la meilleure option en général

## Plan

- 1 Rappels ✓
- 2 Stockage ✓
- 3 Indexation ✓
- 4 Optimisation des opérateurs
  - 4.1 Motivation ✓
  - 4.2 Algorithmes de jointure ✓
  - 4.3 Autres opérateurs

## Sélectivité

**Taux de sélectivité** d'une condition  $\phi$  (ou d'une requête) pour une relation donnée:

$$\frac{\# \text{ d'enregistrements sélectionnés}}{\# \text{ d'enregistrements}}$$

Le choix de certains algorithmes dépend de la sélectivité

On ne connaît la « vraie » valeur de la sélectivité **qu'après** avoir évalué la requête

On utilise des statistiques sur les relations pour tenter une **approximation** du taux de sélectivité

## Statistiques sur les relations

Le SGBD conserve, entre autres, les statistiques suivantes pour chaque relations R:

- Nombre d'enregistrements (N), taille d'un enregistrement, nombre d'attributs/page (P)
- Nombre de pages de la relation (les pages ne sont pas toutes remplies de manière optimale)
- $V(a)$  : nombre de valeurs distinctes pour l'attribut a (dans la relation R)
- Estimation de sélectivité pour l'attribut a:  $V(a)/N$
- Profondeur pour les arbres B+
- Nombre de page pour les feuilles d'un arbre B+
- Nombre de valeurs distinctes pour la clé de recherche d'un index

## Sélection

Sélection simple, égalité avec une constante : Scan ou Index (si groupant)

Sélection simple avec index non groupant : Index + **tri des adresses de pages**, parcours ordonné. Très efficace si l'ensemble des adresses à trier tiens en mémoire

Sélection généralisés, deux approches:

1. On choisit une sous-condition (qui concerne le moins de pages = la plus sélective) et on applique les autres sous-conditions au résultat intermédiaire
2. Si on a deux sous-conditions « AND » avec 2 indexes (types 2 ou 3) séparés, calcul des ensembles de **rid** et intersection des résultats. On applique ensuite les autres critères.

## Résultat trié/élimination des doublons

Plusieurs techniques :

- Utilisation d'un index (type B+-tree) si groupant ou si coût d'accès au données « raisonnable » (résultat dans l'index ou peu de résultats + accès aux données)
- Utilisation d'un tri explicite après calcul des résultats (+ élimination des doublons durant la phase de tri)

## Projection

On parle ici de la projection,  $\pi$  de l'algèbre relationnelle, donc avec élimination des doublons :

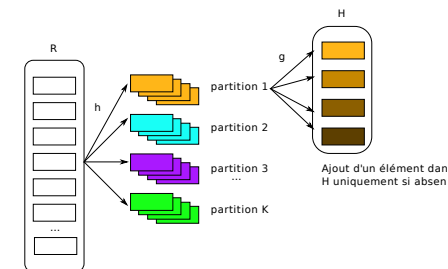
```
SELECT DISTINCT a,b FROM t
```

- Si index sur (a,b) disponible, utilisation directe de l'index
- Sinon tri et projection durant la phase de tri
- Double partitionnement par hachage

## Double partitionnement

Repose sur l'utilisation de **deux** fonctions de hachage **h** et **g** **distinctes**

1. On partitionne R en K partitions en utilisant **h**
2. En suite pour chaque partition entre 1 et K, on crée une table de hachage en mémoire (avec **g** comme fonction) pour pour éliminer les doublons de la partition



Permet d'effectuer « DISTINCT » sans tri!

## Opérations ensemblistes

- Intersection et produit cartésien: cas dégénérés de jointure (comment?)
- UNION DISTINCT et EXCEPT sont similaires. 2 approches:
  1. Par tri. On tri les deux relations sur tous les attributs et on fusionne en éliminant les doublons. Résultat trié
  2. Par hachage. Technique du double partitionnement. On partitionne R et S avec h. Pour chaque partition de S et R, on ajoute les éléments dans une table H, en éliminant les doublons

## Opérations d'agrégat

- Sans GROUP BY:
  - En général, il faut faire un scan de la relation
  - Si les attributs agrégés sont dans un index, on peut faire un scan d'index uniquement (en espérant que l'index est plus petit)
- Avec GROUP BY: identique au cas sans GROUP BY mais tri préalable pour déterminer les groupes, et scan « groupe par groupe » pour calculer la fonction d'agrégat

## Conclusion

L'algèbre relationnelle est simple (quelques opérateurs pour exprimer l'ensemble des requêtes)

Chaque opérateur peut être réalisé de plusieurs manières différentes, avec différents compromis

Tout cela est encore complexifié quand on considère les compositions d'opérateurs (prochain cours)

Tout est encore plus complexifié si on considère que le SGBD gère plusieurs requêtes en parallèle (hors programme)

En pratique, une part importante du moteur de requête des SGBD est l'implantation d'heuristiques pour faire les meilleurs choix (ou plutôt, les moins pires).