

Plan

Bases de données

Polytech Paris-Sud

Apprentis 4^{ème} année

Cours 1 : Généralités & rappels

kn@lri.fr

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

1 Rappels

- 1.1 Avant-propos
- 1.2 Algèbre relationnelle
- 1.3 SQL



But du cours

Le but du cours est de donner une formation avancée sur un aspects central des bases de données : l'évaluation de requêtes. Le plan suivi par le cours est le suivant:

- Rappels de l'algèbre relationnelle et d'SQL (rapide)
- Propriétés physiques des disques (Rotatifs, SSD), notion de page mémoire, hierarchie d'accès mémoire
- Index: généralités, coût, structures de données (Arbres B+, Hash Index, Bitmap Index)
- Algorithmes de jointure
- Plan de requête et optimisations algébriques
- Bonus: ce que vous voulez (XML, Cloud, J2SE, ...)

Organisation du cours

9 séances de 4h:

Date	Type	Heure
2/2	Cours/TD	13h30-17h30
5/2	Cours/TD	8h30-12h30
9/2	TP	13h30-17h30
12/2	Cours/TD	8h30-12h30
7/4	TP	8h30-12h30
9/4	Cours/TD	8h30-12h30
13/4	Cours/TD	12h30-17h30
14/4	TP	13h30-17h30
16/4	Cours bonus/exam	8h30-12h30

- Cours/TD : Kim Nguyen
- TP: Cécile Pereira (certains TP seront notés)



Plan

1 Rappels

1.1 Avant-propos ✓

1.2 Algèbre relationnelle

1.3 SQL

Qu'est-ce que l'algèbre relationnelle?

Une **algèbre** (ou **structure algébrique**) est un **ensemble d'objets** (que l'on étudie) muni d'un ensemble d'**opérations** (qui permettent de manipuler les objets)

Les objets manipulés par l'algèbre relationnelle sont les **relations** i.e. des **ensembles de n-uplets**.

(Rappel: une relation n-aire est juste un ensemble de n-uplets. Par exemple, la relation d'égalité sur les entiers est l'ensemble qui contient tous les couples $(0, 0), (1, 1), (2, 2) \dots$)

On ne considère que des relations **finies**, sur des n-uplets **fixes** dont les composantes ont un type **simple**

$\{ (1, "Kim", 32, T), (3, "Foo", 28, F), (2, "Bar", 77, T) \}$

- Les relations représentent des tables: ensemble finis
- Les relations contiennent des n-uplets de la même taille
- Un n-uplet ne peut pas contenir un ensemble (pas de table dans une table)
- (optionel) on ajoute un **schema** à la relation (ex. (id, nom, age, prof)).

Les opérateurs de l'algèbre relationnelle (1/2)

(attention, plusieurs présentations possibles)

R et S sont deux relations, munies chacune d'un schéma ($\mathbb{R}=(a_1, \dots, a_m)$ et $\mathbb{S}=(b_1, \dots, b_n)$)

Opérateurs ensemblistes:

Union : $R \cup S \stackrel{\text{def}}{=} \{ r \mid r \in R \vee r \in S \}$ (requiert $\mathbb{R} = \mathbb{S}$)

Différence : $R \setminus S \stackrel{\text{def}}{=} \{ r \mid r \in R \wedge r \notin S \}$ (requiert $\mathbb{R} = \mathbb{S}$)

Produit : $R \times S \stackrel{\text{def}}{=} \{ (r_1, \dots, r_m, s_1, \dots, s_n) \mid (r_1, \dots, r_m) \in R \wedge (s_1, \dots, s_n) \in S \}$

Q1: A-t-on besoin de l'intersection ? ($R \cap S$)

R1: Non car $R \cap S = (R \cup S) \setminus ((S \setminus R) \cup (R \setminus S))$

Les opérateurs de l'algèbre relationnelle (2/2)

(attention, plusieurs présentations possibles)

R est une relation, munie d'un schéma ($\mathbb{R}=(a_1, \dots, a_m)$)

Opérateurs relationnels:

Projection : $\pi_{a_1, \dots, a_k}(R) \stackrel{\text{def}}{=} \{ (r.a_1, \dots, r.a_k) \mid r \in R \}$

Sélection : $\sigma_{\phi}(R) \stackrel{\text{def}}{=} \{ r \in R \mid \sigma(r) \}$

σ est une
formule logique
sur r

Renommage : $\rho_{a_1 \mapsto b_1, \dots}(R)$ associe R au schéma $\mathbb{R}'=(b_1, \dots)$

Opérateurs dérivés

R et S sont deux relations, munies chacune d'un schéma (R et S)

■ **Jointure**: $R = (a_1, \dots, a_m, c_1, \dots, c_l)$ et $S = (b_1, \dots, b_n, c_1, \dots, c_l)$

$$R \bowtie S \stackrel{\text{def}}{=} \{ (r.a_1, \dots, r.a_m, r.c_1, \dots, r.c_l, s.b_1, \dots, s.b_n) \mid r \in R \wedge s \in S \wedge \forall 1 \leq i \leq l, r.c_i = s.c_i \}$$

■ **Intersection**: $R \cap S = \{ r \mid r \in R \wedge r \in S \}$

■ **Division**: $R \div S \stackrel{\text{def}}{=} T$, telle que $T \times S \subseteq R$ (les attributs de S sont un sous-ensemble des attributs de T)

Pourquoi utiliser l'algèbre relationnelle ?

- Modèle abstrait qui permet de raisonner sur les requêtes sans se soucier de la syntaxe
- Permet de déduire des **optimisations algébriques**

Par exemple:

$$\sigma_\phi(R \cup S) = \sigma_\phi(R) \cup \sigma_\phi(S)$$

Avantageux si R et S ont beaucoup d'éléments mais que σ_ϕ en sélectionne peu.

Plan

1 Rappels

1.1 Avant-propos ✓

1.2 Algèbre relationnelle ✓

1.3 SQL

SQL

SQL (*Structured Query Language*) est un langage de programmation dédié permettant de manipuler les données d'une BD relationnelle. Il permet de:

- Créer et détruire des tables
- Insérer, supprimer, modifier des lignes d'une table
- Interroger des tables
- ...

SQL ≠ Algèbre relationnelle

- Table ≠ Relation : les tables peuvent avoir plusieurs copies de la même ligne, alors que les relations sont des ensembles
- Opérations de comptage, d'agrégat, groupage, ...
- Les types sont finis et ont toujours une taille fixe (INTEGER, VARCHAR(40), DATE, ...)

Insertion/suppression/mise à jour

```
INSERT INTO MaTable [ (col1,...,coln) ] VALUES (val1,...,valn);
```

- Si la liste de colonnes est précisée les valeurs sont insérées dans les colonnes correspondantes, sinon dans l'ordre du schéma

```
DELETE FROM MaTable [ WHERE condition ];
```

- Supprime les lignes pour lesquelles *condition* est vraie (expression booléenne sur les colonnes). Si WHERE est absent, supprime toutes les lignes.

```
UPDATE MaTable SET col1=val1, ..., coln=valn [ WHERE condition ];
```

- Mise à jour de toutes les colonnes *i* des lignes pour lesquelles *condition* est vraie (expression booléenne sur les colonnes). Si WHERE est absent, modifie toutes les lignes.

Création/destruction de table

```
CREATE TABLE MaTable (  
    att1 type1 [constr_col1], ..., attn typen [constr_coln]  
    [, constr_table]);
```

- MaTable : nom de la table
- att_i : nom de l'attribut *i*
- att_i : type de l'attribut *i*. Exemples de types: INTEGER, VARCHAR(*n*), ... (**dépend du système utilisé**)
- constr_col_i : contrainte sur la colonne *i*. Exemple de contraintes: PRIMARY KEY, NOT NULL, DEFAULT *n*, ...
- constr_table : contrainte de table. Exemple de contrainte de table: CHECK *cond*, UNIQUE (*col1*, ..., *coln*), ...

```
DROP TABLE Table1, ..., Tablen [CASCADE];
```

- CASCADE : détruit aussi les objets dépendants de la table (vues, autres tables avec clés étrangères, ...) (**dépend du système utilisé**)

Requêtes SQL 1/3

```
SELECT [ALL|DISTINCT] res1, ..., resn  
FROM tab_ref1, ..., tab_refm  
[WHERE condition_w]  
[GROUP BY col1, ..., colk]  
[HAVING condition_h]  
[ORDER BY col1, ..., coll [ASC|DESC]]
```

- ALL force à garder tous les résultats, DISTINCT retire les doublons
- res_i peut être un nom de colonne, * (toutes les colonnes), un agrégat (SUM(price), éventuellement nommé : AS TotalPrice)
- tab_ref_i est soit un nom de table, soit une sous-requête ((SELECT ...)) éventuellement nommé (AS T1)
- condition_w est une condition booléenne sur les attributs des *m* tables mentionnées
- GROUP BY et HAVING définissent des conditions de groupage
- ORDER BY trie les résultats en ordre croissant (par défaut ou ASC) ou décroissant (DESC)

Requêtes SQL 2/3

```
(req1) UNION [ALL] (req2)  
(req1) INTERSECT (req2)  
(req1) EXCEPT (req2)
```

Union, intersection et différence de deux requêtes. Par défaut, retire les doublons des résultats des requêtes (comportement ensembliste) sauf pour UNION ALL ou si SELECT ALL a été utilisé dans les sous-requêtes

Requêtes SQL 3/3

Exemple de conditions de groupage. On considère une table d'employés (nom), appartenant chacun à un département (num_dept) et ayant chacun un salaire (sal). On souhaite avoir les salaires moyens, pour chaque département, pour les départements ayant plus de 10 employés.

```
SELECT num_dept, AVERAGE(sal)  
FROM TABLE_EMP  
GROUP BY num_dept  
HAVING COUNT(nom) >= 10;
```

HAVING est nécessaire car la clause WHERE s'applique ligne à ligne, ici on veut groupe à groupe (i.e. pour chaque département, i.e. pour toutes les lignes qui ont le même département).