Bases de données – cours 3
Introduction en SQL

Catalin Dima

Présentation de SQL

- SQL est un LDD :
  - Création/modification/suppression de tables.
  - Création d'utilisateurs et gestion des droits.
  - Implémentations en architectures client-serveur : MySQL, PostgreSQL.
- SQL est un LMD :
  - Implémentation de l'algèbre relationnelle.
  - Langage de **requêtes** select-from-where.
  - Variables de type tuple et attributs référencés à la C/Java.
  -
- Notre initiation commence par la partie LMD.
- SQL est "case-insensitive".
  - ... mais nous allons mettre les mots-clé en majuscules pour les différencier des autres constructions.

Première version de SELECT

- Partons de notre table “Employés”, de schéma *Employés(Id, Nom, Prénom, Dept, Salaire)*.
- Afficher les tuples correspondant aux employés du 1er département dont le salaire dépasse les 34kE :

```
SELECT *
FROM Employés
WHERE Dept = 1 AND Salaire >= 34;
```

- Forme de requête de base en SQL, select-from-where :
  - La clause `FROM` donne la (ou les) relation(s) sur laquelle (lesquelles) la requête s’applique.
  - La clause `WHERE` est une condition jouant le même rôle que la condition d’une opération de sélection, type σ_C.
  - La clause `SELECT` permet d’indiquer quels attributs sont à garder et à afficher lors du calcul du résultat, similaire à une opération de projection $\pi_{A_1,\ldots,A_k}$.
  - Utiliser `*` dans un `SELECT` indique que tous les attributs produits pendant le calcul seront retenus.
- Résultat pour notre requête ....

Projection en SQL

- Comme indiqué, la partie `SELECT` de toute requête de type select-from-where permet d'implémenter l'opération de projection.
  - Attention donc ! La clause `SELECT` ne correspond pas à une sélection !
- Afficher les noms et prénoms des employés du 1er département dont le salaire dépasse les 34kE :

```
SELECT Nom, Prénom
FROM Employés
WHERE Dept = 1 AND Salaire >= 34;
```

- Syntaxe permettant d'implémenter une projection suivie d'un renommage : afficher les ID des employés du 1er département dont le salaire dépasse les 34kE, tout en renommant la colonne de la table résultat en `NoCrt` :

```
SELECT Id AS NoCrt
FROM Employés
WHERE Dept = 1 AND Salaire >= 34;
```

Projection en SQL (2)

- Encore une autre variante de syntaxe, permettant de recalculer un nouvel attribut (ici `SalMens`) et de créer un autre nouvel attribut possédant une valeur constante (ici `Monnaie`) :
  - Afficher les ID des employés du 1er département dont le salaire dépasse les 34kE, et afficher aussi leur salaire mensuel (en Euro cette fois-ci) plus une colonne supplémentaire contenant la chaîne `'Euro'` :

```
SELECT Id AS NoCrt, Salaire/12*1000 AS SalMens,
       'Euro' AS Monnaie
FROM Employés
WHERE Dept = 1 AND Salaire >= 34;
```

Sélection en SQL

- La partie `WHERE` permet de filtrer des tuples selon la condition déclarée.
- Conditions construites à l'aide d'opérateurs sur les types primitifs.
  - Variables utilisées dans les conditions = noms d'attributs de la (des) relation(s) indiquée(s) dans la clause `FROM`.
  - Désambiguation parfois nécessaire (on en reviendra).
  - Opérateurs arithmétiques et prédicats de comparaison comme en C ou Java sur les données de type entier ou flottant.
  - Particularité : <> pour représenter le prédicat "différent", au lieu de `!=`.
  - Opérateurs booléens pour la construction de contraintes complexes : `AND, OR, NOT`.
  - Concaténation de chaînes de caractères : opérateur binaire `||`.
  - Comparaisons de chaînes de caractères : opérateurs `<, <=, >,` >= faisant référence à l'ordre lexicographique.

Sélection en SQL

- Conditions de type `LIKE` dans les comparaisons de chaînes de caractères :
  - `s LIKE p` : la valeur de l'attribut `s` respecte le patron `p`.
  - Patrons de chaînes de caractères : `%` remplace n'importe quelle chaîne de caractères, `_` remplace un seul caractère, et `''` dénote un seul apostrophe.
- Les attributs de type `DATE` peuvent être comparés avec des valeurs constantes de la forme `DATE '2011-12-31'`.
- Similaire pour `TIME`.
- Exemple pour comparer un attribut `Attr` de type `TIME` avec l'heure 23h45m18s2/10 :
  `WHERE Attr > TIME '23:45:18.2';`

Comparaisons impliquant la valeur NULL et valeur booléenne UNKNOWN

- Les attributs peuvent prendre la valeur NULL.
  - Exemple : jointure externe.
- Pour cela, WHERE doit permettre la manipulation et la comparaison de cette valeur.
- Deux règles importantes pour la prise en compte de cette valeur :
  1. Le résultat de chaque opération devient NULL lorsqu'au moins une opérande est NULL.
  2. Le résultat de chaque comparaison devient UNKNOWN lorsqu'au moins une opérande est NULL.
  3. Mais NULL ne peut pas être utilisé en tant que constante !
- Valeur UNKNOWN = troisième valeur de vérité !

Logique à trois valeurs de vérité

- Tables de vérité pour les opérateurs booléens :

x	*y*	*x* AND *y*	*x* OR *y*	NOT *x*
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE	FALSE

Produit cartésien et jointure naturelle en SQL

- La clause `FROM` force l'utilisation de plus d'une seule table dans le reste de la requête.
- Les paramètres de `FROM` forment une liste de tables...
- ... lesquelles sont combinés par produit cartésien, avant de passer au traitement du reste de la requête.
- Pour deux relations `A` et `B`, le produit cartésien $A \times B$ se calcule en SQL comme suit :

```
SELECT *
FROM A, B;
```

  - Clause `WHERE` optionnelle !

- Pour calculer la jointure naturelle, il faut utiliser l'implémentation de la projection par `SELECT` et la sélection par `FROM`.
- En supposant que les schémas des deux tables sont `A(X,Y,Z)` et `B(U,X,V,Y)`, leur jointure naturelle se calcule en SQL comme suit :

```
SELECT A.X, A.Y, Z, U, V
FROM A, B
WHERE A.X=B.X AND A.Y=B.Y;
```

Utilisation de select-from-where sur plusieurs relations

- Prenons notre table “Employés” de schéma *Employés(Id, Nom, Prénom, Dept, Salaire)*.
- Et une deuxième table “Départements”, de schéma *Départements(NoDept, Nom, Projet, Budget)*.
- Afficher les personnels qui travaillent dans un projet de plus de 500kE et gagnent moins de 34kE :
  - Il faut rassembler les deux tables pour avoir toutes les informations.
  - Rassembler = produit cartésien.

```
SELECT Nom, Prénom
FROM Employés, Départements
WHERE Dept=NoDept AND Budget>=500 AND Salaire<=34;
```

- Expression en algèbre relationnelle :

$$\pi_{\text{Nom},\text{Prénom}}\left(\sigma_{\text{Dept}=\text{NoDept}\wedge\text{Budget}\geq 500\wedge\text{Salaire}\leq 34}\left(\text{Employés}\times\text{Départements}\right)\right)$$

Opérations booléennes en SQL

- Intersection :

```
(SELECT Nom, Prénom
FROM Employés
WHERE Dept = 1)
INTERSECT
(SELECT Nom, Prénom
FROM Employés
WHERE Salaire >= 34);
```

- Union : similaire, mot-clé `UNION`.
- Différence : similaire, mot-clé `EXCEPT` :
  - Afficher les tuples correspondant aux employés du 1er département dont le salaire est *inférieur à* 34kE :

```
(SELECT Nom, Prénom
FROM Employés
WHERE Dept = 1)
EXCEPT
(SELECT Nom, Prénom
FROM Employés
WHERE Salaire >= 34);
```

Variables de type tuple

- Parfois on a besoin de référencer la même relation *deux fois* dans la même requête.
  - Afficher les personnels qui travaillent dans *au moins deux* projets.
  - Il nous faut pouvoir référencer la table “Employés” deux fois.
- La syntaxe de `FROM` permet de définir des variables de type tuple, prenant comme valeur une table et pouvant être référencée à l’intérieur de la requête :

```
SELECT Nom, Prénom
FROM Employés, Départements Dep1, Départements Dep2
WHERE Dept=Dep1.NoDept AND Dept=Dep2.NoDept
      AND Dep1.Projet<>Dep2.Projet;
```

- Petit problème : le personnel qui travaille sur trois ou plusieurs projets apparaît au moins deux fois...
- Construire une relation sans duplicats : `SELECT DISTINCT`

```
SELECT DISTINCT Nom, Prénom
FROM Employés, Départements Dep1, Départements Dep2
WHERE Dept=Dep1.NoDept AND Dept=Dep2.NoDept
      AND Dep1.Projet<>Dep2.Projet;
```

Imbrication des requêtes dans les conditions `WHERE`

- On peut utiliser des *sous-requêtes* pour construire des valeurs/tables sur lesquelles on peut vérifier certaines conditions.
  - Retrouver les départements qui sont impliqués dans le développement du jeu en réseau peut se faire dans une sous-requête, dont le résultat peut être utilisé dans la clause `WHERE` :

```
SELECT Nom, Prénom
FROM Employés
WHERE Id IN
  (SELECT NoDept
   FROM Départements
   WHERE Projet = 'Jeu en réseau'
  );
```

Conditions dans les clauses `WHERE`

- Autres conditions sur les tables obtenues par sous-requêtes :
  - Test d'égalité `s = R`, erreur si le résultat `R` de la sous-requête contient plus d'une seule ligne ou plus d'une seule colonne.
  - `EXISTS R`, vrai si `R` n'est pas vide.
  - `s > ALL R`, vrai si `s` est plus grand que toutes les valeurs de la table `R` unaire (c.à.d. ayant une seule colonne).
    - Erreur si `R` contient deux colonnes ou plus !
  - Similaire avec toutes les autres prédicats de comparaison, `<`, `<=`, `>=`, `<>`.
  - `s > ANY R` vrai s'il existe une valeur dans `R` (encore une fois relation unaire) qui est plus petite que `s`. (Aurait dû être nommé `SOME`...).
  - `NOT` peut être employé sur n'importe quelle condition.

Requêtes corrélées

- Prenons une table de schéma *Projets(PrjId,Nom,Année)*.
- On cherche les projets qui se sont déroulés sur deux années ou plus.
- Solution en utilisant des requêtes corrélées :

```
SELECT DISTINCT PrjId, Nom
FROM Projets OldPrj
WHERE Année < ANY
    (SELECT Année
     FROM Projets
     WHERE PrjId=OldPrj.PrjId
    );
```

- Hypothèse : `PrjId` est une clé primaire.
- La sous-requête est évaluée pour toute instance de l'attribut `Old.PrjId`.

Sous-requêtes dans les clauses FROM

- Revenons aux tables *Employés* et *Départements*.
- Rajoutons une table *RespProj* de responsables de projets, de schéma *RespProj(Nom, Prénom,Projet)*.
- Afficher les personnels qui travaillent dans les projets dont le responsable est M. Untel Quelqun :
  - Retrouver d'abord les *ProjId* dont le responsable est ce monsieur,
  - ... en mettant le résultat dans une relation.
  - Construire en parallèle une relation qui indique qui travaille dans quel projet.
  - Enfin, combiner les deux relations et afficher le résultat.

```
SELECT Nom, Prénom
FROM RespProj, (SELECT Nom, Prénom, Projet
      FROM Employés, Départements
      WHERE Dept=NoDept) PersProj
WHERE RespProj.Nom='Quelqun' AND RespProjet.Prénom='Untel'
      AND RespProj.Projet = PersProj.Projet;
```

- Remarquer l'utilisation d'une variable de type tuple récupérant le résultat de la sous-requête.

Construction de nouvelles relations par différentes variantes de jointure

- La clause `FROM` est censée s'appliquer à une relation.
- La virgule remplace le produit cartésien.
- `CROSS JOIN` est un synonime pour le produit cartésien, donc la clause suivante :

  ```
  FROM Employés, Départements
  ```

  est équivalente à :

  ```
  FROM Employés CROSS JOIN Départements
  ```

Construction de nouvelles relations par différentes variantes de jointure (2)

- La θ-jointure est aussi implémentée :

```
Employés JOIN Départements ON
   Employés.Dept = Département.NoDept
```

- Enfin, si on veut calculer la jointure naturelle des deux tables, il suffira de projeter le résultat précédent sur les attributs non-rédondants, c.à.d. :

```
SELECT Id, Nom, Prénom, Dept, Salaire,
       Départements.Nom, Budget, Projet
FROM Employés JOIN Départements ON
   Employés.Dept = Département.NoDept;
```

- Et enfin, les jointures externes sont implémentées aussi :
  - La jointure externe bilatérale : `NATURAL FULL OUTER JOIN`.
  - La jointure externe gauche : `NATURAL LEFT OUTER JOIN`.
  - La jointure externe droite : `NATURAL RIGHT OUTER JOIN`.

Aggrégation en SQL

- SQL offre cinq opérateurs d'aggrégation : `SUM, AVG, MIN, MAX` et `COUNT`.
- Ils sont appliqés, habituellement, à un nom d'attribut (colonne), construisant une table ayant une seule unaire et ayant une seule ligne.
- Exception pour l'expression `COUNT(*)`, qui calcule le nombre de tuples obtenus par les clauses `FROM/WHILE` de la requête.
  - Afficher la moyenne des salaires et le nombre d'employés :

```
SELECT AVG(Salaire), COUNT(*)
FROM Employés;
```

- On peut éliminer les duplicats en mettant le mot-clé `DISTINCT` devant l'opérande, comme dans `COUNT(DISTINCT Salary)`.

Groupement en SQL

- Afficher la liste des salaires dans l'entreprise, et pour chaque valeur de salaire le nombre de salariés ayant le salaire respectif.
  - Il nous faut grouper les tuples par salaire,
  - ... puis, pour chaque groupe, appliquer un opérateur `COUNT`.
- Solution pour grouper : `GROUP BY`, suivi par le nom d'attribut sur lequel le groupement se fait.

```
SELECT Salaire, COUNT(Id)
FROM Employés
GROUP BY Salaire
```

- Une table "intermédiaire" est construite, dans laquelle les lignes sont groupées selon la valeur de l'attribut *Salaire*.
- On peut imaginer qu'on aura, dans cette nouvelle table, autant de lignes que de valeurs de salaire différentes, et, dans chaque ligne, des "sous-lignes" contenant les tuples de notre table initiale.
- L'opérateur aggrégé `COUNT` produira alors un résultat pour chaque groupe, donc pour chaque (macro)-ligne.

Clauses de type HAVING

- En groupant, il peut être utile de filtrer les groupes par une condition.
- C'est l'utilité du mot clé HAVING.
- Exemple : produire la liste des départements avec, pour chacun, la somme des budgets des projets dans lesquels l'employé respectif participe, le tout seulement pour des budgets supérieurs à 500kE :

```
SELECT Nom, SUM(Budget)
FROM Départements
GROUP BY Budget
HAVING MIN(Budget) > 500;
```

- À noter que l'aggrégation dans la clause HAVING s'applique seulement aux tuples du groupe en cours de construction.
- Tout attribute des relations dans la clause FROM peut être utilisé dans la clause HAVING, mais seulement les attributs dans la clause GROUP BY peuvent apparaître non-aggrégés dans HAVING.

Trier les résultats

- Enfin, on peut trouver parfois utile de trier les résultats d'une requête.
- Mots-clé `ORDER BY`, suivis par la liste d'attributs sur lesquels le tri est fait.
- Si plusieurs attributs dans la liste, alors on utilise l'ordre lexicographique sur les tuples :
  - Si deux tuples t_1 et t_2 ont les mêmes valeurs pour les attributs A_1, A_2, alors le premier à être affiché sera celui qui a la valeur de l'attribut A_3 la plus petite.

Exécution des requêtes SQL

- Supposons une requête simple SQL :

```
SELECT A, B
FROM R1, R2
WHERE C
```

- L'évaluation et la réponse à cette requête est construite ainsi :
  1. On prend chaque tuple dans la relation `R1`,
  2. ... et chaque tuple dans la relation `R2`,
  3. Si les deux tuples satisfont la clause `WHERE`,
  4. ... alors on évalue les expressions dans la clause `SELECT`,
  5. ... et on construit les tuples de valeurs calculés dans le `SELECT`.