

Laboratoires de bases de données

Laboratoire n°4

Groupements, imbrications et ensembles

par Danièle BAYERS et Louis SWINNEN
Révision 2018 : Vincent Reip

Ce document est disponible sous licence Creative Commons indiquant qu'il peut être reproduit, distribué et communiqué pour autant que le nom des auteurs reste présent, qu'aucune utilisation commerciale ne soit faite à partir de celui-ci et que le document ne soit ni modifié, ni transformé, ni adapté.



<http://creativecommons.org/licenses/by-nc-nd/2.0/be/>

La Haute Ecole Libre Mosane (HELMo) attache une grande importance au respect des droits d'auteur. C'est la raison pour laquelle nous invitons les auteurs dont une oeuvre aurait été, malgré tous nos efforts, reproduite sans autorisation suffisante, à contacter immédiatement le service juridique de la Haute Ecole afin de pouvoir régulariser la situation au mieux.

Février 2018

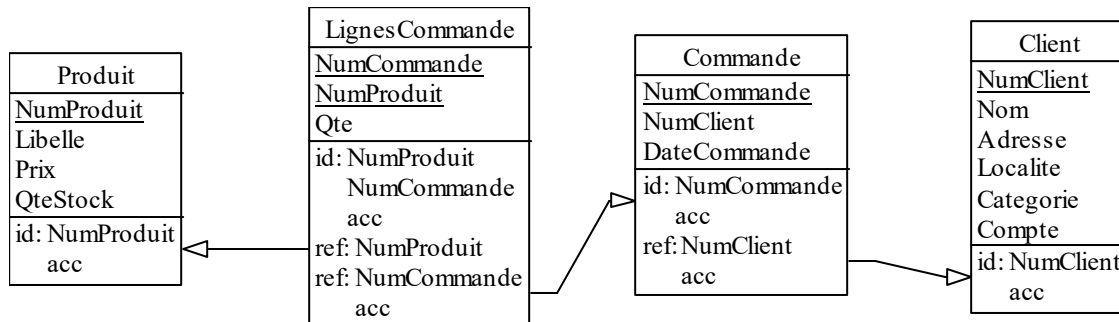
1. Objectif

L'objectif de ce laboratoire est de décrire les opérations de groupement, les requêtes imbriquées (également appelée sous-select) et les requêtes ensemblistes (au moyen des opérateurs UNION, INTERSECT, MINUS/EXCEPT).

Nous terminerons en abordant les requêtes particulières avec *case*, *exists*, *any* et *all*.

2. La base de données

Dans les exemples, nous supposons que nous disposons de la base de données suivante :



3. Requêtes sur les groupes

Grâce aux requêtes sur les groupes, il est possible d'obtenir des résultats groupés qui peuvent s'avérer intéressants. Ces requêtes sur les groupes utilisent des *fonctions de groupe* (appelées aussi fonctions d'agrégat).

3.1 Les fonctions de groupe

Le langage SQL prévoit les fonctions de groupe suivantes applicables à un attribut :

- AVG – calcule la moyenne
- MAX – Retourne la plus grande valeur
- MIN – Retourne la plus petite valeur
- COUNT – Compte le nombre de résultat
- SUM – Effectue une somme

Ces fonctions de groupe prennent en argument le nom de l'attribut. Elles peuvent également intégrer le mot clé DISTINCT qui supprime les valeurs identiques.

Exemples :

```
SELECT AVG(prix)
FROM produit
```

Obtenir le prix moyen des produits

Pour clarifier l'utilisation de COUNT, voici quelques exemples en guise d'illustration :

```
SELECT *
FROM client
```

Retourne tous les enregistrements de la table client (n lignes)

```
SELECT COUNT(*)  
FROM client
```

Retourne le nombre d'enregistrements présents dans la table client (i.e. le nombre de lignes du résultat précédent).

```
SELECT COUNT(categorie)  
FROM client
```

Retourne le nombre d'enregistrements présents dans la table client pour lesquels la catégorie n'est pas 'null'.

```
SELECT COUNT(DISTINCT categorie)  
FROM client
```

Retourne le nombre de valeurs différentes pour la catégorie (l'absence de valeur 'null' n'étant pas comptabilisée).

3.2 Les clauses **GROUP BY** et **HAVING**

La clause GROUP BY permet de regrouper les résultats autour de valeurs d'attribut déterminées. Cette clause est utilisée en combinaison avec des fonctions de groupe.

Forme :

```
SELECT attr1, ..., attri, fct_groupe1(attrj), ..., fct_groupen(attrp)  
FROM table1  
[JOIN tablen ON conditionx]  
[WHERE conditiony]  
GROUP BY attr1, ..., attri  
[HAVING conditionz]
```

Dans la forme présentée ci-dessus, il faut remarquer ceci :

- L'utilisation conjointe d'attributs et de fonctions de groupe dans le **SELECT impose** l'utilisation d'une clause **GROUP BY**
- Le **GROUP BY** doit souvent reprendre les attributs n'intégrant pas les fonctions de groupe
- La clause **WHERE** ne peut reprendre dans sa condition que des attributs (pas de fonction de groupe)
- La clause **HAVING** permet d'appliquer une condition sur les groupements en utilisant une fonction de groupe
- Dans **fct_groupe**, on retrouve les fonctions de groupe montrées plus haut : **SUM**, **MAX**, **MIN**, **AVG** et **COUNT**.

Exemple :

```
SELECT cli.localite, SUM(p.prix * lc.qte) CA  
FROM client cli  
JOIN commande c ON cli.numClient = c.numClient  
JOIN lignesCommande lc ON lc.numCommande = c.numCommande  
JOIN produit p ON lc.numProduit = p.numProduit  
GROUP BY cli.localite
```

Cette requête retourne comme résultat le chiffre d'affaires par localité. Le résultat montrera les localités différentes et le chiffre d'affaires calculé.

La clause `HAVING` peut être définie comme suit : « *il s'agit d'une clause `WHERE` sur un groupe défini par `GROUP BY`* » (définition extraite de [1]). Tout comme la clause `WHERE`, la clause `HAVING` limite le résultat aux seuls enregistrements vérifiant la condition énoncée.

Exemple :

```
SELECT cli.categorie, SUM(p.prix * lc.qte) CA
FROM client cli
JOIN commande c ON cli.numClient = c.numClient
JOIN lignesCommande lc ON lc.numCommande = c.numCommande
JOIN produit p ON lc.numProduit = p.numProduit
WHERE cli.localite IN ('Liege', 'Bruxelles')
GROUP BY cli.categorie
HAVING SUM(p.prix * lc.qte) > 500
```

Dans cet exemple, on calcule le chiffre d'affaires par catégorie de client, uniquement pour les clients de Liège et de Bruxelles. On filtre le résultat pour garder uniquement les catégories dont le chiffre d'affaires est supérieur à 500.

On remarque dans cet exemple l'utilisation différente de `WHERE` et `HAVING`. `WHERE` est utilisé pour filtrer les catégories, attribut de la table client tandis que `HAVING` est utilisé pour filtrer le résultat d'une fonction de groupe, ici `SUM`.

4. Les requêtes imbriquées

Les requêtes imbriquées sont l'illustration de *diviser pour régner*. Grâce à l'imbrication des requêtes, il est possible d'obtenir des résultats relativement complexes par la combinaison de plusieurs requêtes simples. De telles requêtes sont très souvent utilisées car elles permettent d'obtenir des résultats plus intéressants.

Le schéma des requêtes imbriquées est souvent le suivant :

```
SELECT attr1, ..., attrn
FROM table1
[JOIN table2 ON condition]
WHERE attrx = ( SELECT attry
                FROM table3
                [JOIN table4 ON condition]
                WHERE condition )
```

Il existe plusieurs formes de requêtes imbriquées. La forme présentée ici comprend un opérateur de comparaison (=). Nous pourrions également trouver un opérateur d'appartenance (`IN` ou `NOT IN`). Nous allons étudier ces deux formes ci-après.

Exemples :

```
SELECT c.NumCommade, c.DateCommande
FROM Commande c
WHERE c.Numclient = (SELECT cl.NumClient
                    FROM Client cl
                    WHERE Compte = '001-3215487-89')
```

La requête ci-dessus permet de retrouver les numéros et dates de commande d'un client dont on connaît le numéro de compte (mais pas le numéro de client). On considère que la sous-requête retournera une seule valeur car le numéro de compte est une clé candidate de la relation Client.

Il est également possible d'imbriquer des requêtes déjà imbriquées (voir *exemple 1* ci-dessous). Il arrive également de créer une requête ayant deux conditions imbriquées séparées par des opérateurs logiques comme OR ou AND (voir *exemple 2* ci-après). **Dans l'illustration ci-dessous, ces deux formes ne sont pas équivalentes.**

Exemple 1 :

```
SELECT attr1, ..., attrn
FROM table1
WHERE attrx = (SELECT attry
                FROM table2
                WHERE attrz < (SELECT attrp
                                FROM table3
                                WHERE condition)
                )
```

Exemple 2 :

```
SELECT attr1, ..., attrn
FROM table1
WHERE attrx = (SELECT attry
                FROM table2
                WHERE condition1 )
OR attrz < ( SELECT attrp
             FROM table3
             WHERE condition2)
```

Les requêtes imbriquées peuvent aussi être utilisées au sein de requêtes de type INSERT, UPDATE ou DELETE. Par exemple, on peut utiliser une sous-requête pour « générer » une nouvelle valeur de clé primaire :

```
INSERT INTO Produit (NumProduit, Libelle, Prix, QteStock)
VALUES ( (SELECT MAX(NumProduit)+1 FROM Produit), 'Lecteur DVD Sony
RX-14', 114, 10)
```

On peut aussi utiliser une sous-requête pour trouver la valeur à affecter à une clé étrangère lorsqu'on ne connaît pas l'identifiant du tuple à référencer :

```
INSERT INTO LignesCommande (NumCommande, NumProduit, Qte)
VALUES (43, (SELECT NumProduit FROM Produit WHERE Libelle = 'Lecteur
DVD Sony RX-14'), 1)
```

Attention, il faut évidemment s'assurer que la sous-requête renvoie une et une seule valeur ; en d'autres termes, on considère dans l'exemple ci-dessous que l'attribut `Libelle` est une clé candidate.

4.1 Utilisation d'un opérateur de comparaison

Pour utiliser une requête imbriquée impliquant un opérateur de comparaison (=, <, >, <=, >=, <>), il est nécessaire que la requête interne **ne retourne qu'une seule colonne et une seule ligne** (une seule valeur donc), qui doit pouvoir être comparée à l'attribut mentionné dans le `WHERE` de la 1^{ère} requête.

C'est le cas dans le format présenté ci-dessous puisque la clause `SELECT` ne précise qu'une seule colonne et qu'on a émis l'hypothèse que le libellé d'un produit était une clé candidate.

Exemples :

```
SELECT Nom, adresse
FROM client cli
JOIN Commande c on cli.numClient = c.numClient
JOIN ligneCommande lc ON c.numCommande = lc.numCommande
WHERE lc.numProduit = (SELECT p.numProduit
                        FROM produit
                        WHERE libelle = 'DVD WALL-E')
```

Obtenir le nom et l'adresse des clients ayant commandé le produit portant le libelle DVD WALL-E.

```
SELECT numProduit, libelle
FROM produit
WHERE prix > ( SELECT AVG(prix)
               FROM produit )
```

Obtenir les produits dont le prix est supérieur au prix moyen.

Afin d'assurer que le résultat de la requête interne ne contient qu'une seule ligne, il est donc très fréquent que le `WHERE` porte sur une valeur précise d'une clé candidate.

4.2 Utilisation d'un opérateur d'appartenance

La forme précédente est souvent trop limitée, c'est pourquoi on trouve régulièrement, dans les requêtes imbriquées, l'opérateur d'appartenance.

La forme de la requête change peu :

```
SELECT attr1, ..., attrn
FROM table1
WHERE attrx [NOT] IN (SELECT attry
                     FROM table2
                     WHERE condition )
```

On remarque la présence des opérateurs `IN` ou `NOT IN` de **comparaison à une liste de données**.

Cette forme impose à la requête imbriquée de retourner **une seule colonne**. Mais la valeur de la colonne peut maintenant prendre en ensemble de valeurs. Ainsi le SGBD sélectionnera les enregistrements dont l'attribut prend une valeur parmi l'ensemble donné.

Exemple :

```
SELECT Nom, adresse
FROM client cli
JOIN Commande c on cli.numClient = c.numClient
JOIN ligneCommande lc ON c.numCommande = lc.numCommande
WHERE lc.numProduit IN (SELECT p.numProduit
                       FROM produit
                       WHERE libelle LIKE '%digital%')
```

Sélectionne le nom et l'adresse des clients ayant commandé un produit dont le libellé contient le mot « digital ».

5. Les opérateurs ensemblistes

SQL-2 définit les opérateurs ensemblistes suivant : *l'union*, *l'intersection* et *la différence*. Ces opérateurs travaillent sur les résultats des requêtes qui peuvent être considérés comme des ensembles.

Ainsi, grâce à l'opérateur *union* il est possible de réunir, dans le même résultat, les enregistrements issus de 2 requêtes. L'*intersection* va permettre de garder uniquement les résultats communs à deux requêtes. Finalement *la différence* permet de garder uniquement les résultats de la première requête qui n'apparaissent pas dans la seconde requête.

Il faut noter que les SGBD supportent diversement les opérateurs ensemblistes.

5.1 L'union

Format :

```
SELECT attr1, ..., attrn
FROM table1 t1
[JOIN table2 t2 ON t1.attrx = t2.attry]
WHERE ...
[ORDER BY ...]
```

UNION [ALL]

```
SELECT attr1, ..., attrn
FROM table3 t3
[JOIN table4 t4 ON t3.attrv = t4.attrw]
WHERE ...
[ORDER BY ...]
```

Le résultat contiendra les résultats des deux requêtes *sans les éléments dupliqués*. Pour faire apparaître tous les résultats, on utilisera UNION ALL. Pour obtenir un résultat correct, il est nécessaire que les deux requêtes retournent un nombre de colonne identique et des valeurs compatibles (si le 1^{er} attribut dans le résultat de la 1^{ère} requête est un entier, il est nécessaire qu'il en soit de même dans la seconde requête).

Cette commande est disponible sous *Oracle* et *SQL Server*.

Exemple :

```
SELECT numCommande, numClient, dateCommande
FROM commande
WHERE numClient = 'C001'
UNION
SELECT numCommande, numClient, dateCommande
FROM commande
WHERE numClient = 'C003'
```

Liste les commandes des clients C001 et C003.

5.2 L'intersection

Format :

```
SELECT attr1, ..., attrn
FROM table1 t1
[JOIN table2 t2 ON t1.attrx = t2.attry]
WHERE ...
[ORDER BY ...]
```

INTERSECT

```
SELECT attr1, ..., attrn
FROM table3 t3
[JOIN table4 t4 ON t3.attrv = t4.attrw]
WHERE ...
[ORDER BY ...]
```

Le résultat contiendra uniquement les enregistrements communs aux deux requêtes. Le nombre de colonne ainsi que leur type doit être compatible afin que le SGBD puisse opérer l'intersection entre les deux résultats.

Cette commande est disponible sous *Oracle* et *SQL Server* (≥ 2005).

Exemple :

```
SELECT numclient
FROM commande
WHERE EXTRACT(YEAR FROM DateCommande) = 1995
INTERSECT
SELECT numclient
FROM commande
WHERE EXTRACT(MONTH FROM DateCommande) = 10
```

Liste les clients ayant commandés en 1995 et durant le mois d'octobre. Attention, dans cette formulation, un client ayant commandé en octobre 1994 et en juin 1995 apparaîtrait dans le résultat.

5.3 La différence

Format :

```
SELECT attr1, ..., attrn
FROM table1 t1
[JOIN table2 t2 ON t1.attrx = t2.attry]
WHERE ...
[ORDER BY ...]
```

EXCEPT

```
SELECT attr1, ..., attrn
FROM table3 t3
[JOIN table4 t4 ON t3.attrv = t4.attrw]
WHERE ...
[ORDER BY ...]
```

Le résultat contiendra les enregistrements de la première requête qui ne sont pas présents dans la seconde.

Cette commande est disponible sous *Oracle* (et s'appelle `MINUS`) et *SQL Server* (≥ 2005).

Exemple :

```
SELECT numProduit
FROM produit
WHERE LOWER(libelle) LIKE '%informatique%'
EXCEPT
SELECT numProduit
FROM produit
WHERE LOWER(libelle) LIKE '%numerique%'
```

Sélectionne les produits dont le nom contient informatique mais pas numerique.

6. Les requêtes plus complexes

Parmi les requêtes plus complexes, nous allons étudier les expressions CASE et EXISTS.

6.1 CASE

La commande CASE permet de gérer des conditions. Le CASE s'insère très souvent dans le SELECT (à l'intérieur de la liste des résultats) ou dans un UPDATE.

Format :

```
SELECT attr1, ..., attrn,
CASE
  WHEN attri = vali THEN valq
  WHEN attri = valj THEN valr
  ...
  [ELSE NULL]
END
FROM table1 t1
[JOIN ...]
WHERE ...
[ORDER BY ...]
```

Dans le résultat, à la suite de l'attribut attr_n, le SGBD ajoutera une des valeurs val_q, val_r, ... suivant la condition qui sera vérifiée dans le CASE ou NULL si aucune condition ne l'est.

```
UPDATE table1
SET attr1 = CASE
                WHEN attri = vali THEN valq
                WHEN attrj = valj THEN valr
                ...
                [ELSE NULL]
            END
```

Ici, l'attr1 prendra une valeur qui dépendra de la condition qui sera vérifiée dans le CASE ou NULL si aucune condition ne l'est.

Exemples :

```
SELECT nom, adresse,
CASE
  WHEN categorie = 'C1' THEN 'Particulier'
  WHEN categorie = 'C2' THEN 'Administration'
  WHEN categorie = 'C3' THEN 'PME'
  WHEN categorie = 'C4' THEN 'Entreprise'
  ELSE 'Inconnu'
END
FROM Client
```

Dans cet exemple, on affiche le nom, le prénom et la catégorie du client.

```

UPDATE Produit
SET prix = CASE
    WHEN QteStock < 10 THEN prix * 2
    WHEN QteStock < 100 THEN prix * 1.5
    WHEN QteStock >=100 THEN prix
END
WHERE libelle LIKE '%digital%'

```

Dans cet exemple, on ajuste le prix des produits dont le libellé contient le mot 'digital' en fonction de la quantité en stock..

6.2 Exists

Format :

```

SELECT attr1, ..., attrn
FROM table1 t1
[JOIN table2 t2 ON t1.attrx = t2.attry]
WHERE EXISTS (SELECT *
              FROM table3 t3
              ...)
[ORDER BY ...]

```

La clause `EXISTS` est un peu particulière et déroutante. En fait, une expression `WHERE EXISTS` est évaluée à VRAI si la requête qui suit retourne un résultat (quel qu'il soit). Dans notre format, la requête `WHERE EXISTS` sera vraie si la requête concernant la table `t3` retourne un résultat.

La clause inverse est `WHERE NOT EXISTS`. Elle est évaluée à VRAI si la requête qui suit ne retourne aucun résultat.

Les requêtes de type `[NOT] EXISTS` utilisent la forme `SELECT *` puisque la seule chose importante est de déterminer si un résultat est retourné (peu importe les colonnes impliquées).

Attention ! L'évaluation de `EXISTS` n'est pas toujours aisée en présence des valeurs `NULL`. Ainsi, nous avons vu précédemment qu'une expression dont une opérande est évaluée à `NULL` retourne elle aussi un résultat vide. Dans l'évaluation d'un `NOT EXISTS`, il convient de prendre en compte cette possibilité. Si la requête interne ne retourne pas de résultat à cause d'une valeur `NULL`, la clause `NOT EXISTS` sera évalué à vraie (car il n'y a pas de résultat).

Exemple :

```

SELECT *
FROM client cl
WHERE NOT EXISTS ( SELECT *
                  FROM Commande co
                  WHERE cl.numClient = co.numClient)

```

Liste les clients n'ayant pas commandé.

Il faut noter que la requête interne liée à une clause `[NOT] EXISTS` intégrera généralement dans sa clause `WHERE` une condition faisant intervenir un attribut d'une table interrogée dans la requête principale.

6.3 ALL

Format :

```
SELECT attr1, ..., attrn
FROM table1 t1
[JOIN ...]
WHERE attrx [>|<|=|>=|<=] ALL ( SELECT attry
                                FROM table2 t2
                                ...)
[ORDER BY ...]
```

L'expression WHERE sera évaluée à vraie si la comparaison entre attr_x et chaque résultat retourné par la requête interne (sur la table t₂) est évaluée à vraie. Cette expression sera également vraie si la requête interne retourne un résultat vide. En conséquence, si la comparaison entre attr_x et une valeur du résultat n'est pas vérifiée, l'expression sera évaluée à faux.

Ainsi, comme le précise [2], nous pouvons noter que <> ALL est équivalent à NOT IN.

Exemple :

```
SELECT lc.numCommande
FROM ligneCommande lc
WHERE lc.qte > ALL (SELECT lc2.qte
                   FROM ligneCommande lc2
                   WHERE lc2.numcommande <> lc.numcommande)
```

Sélectionne la commande dont la quantité commandée est supérieure à toutes celles des autres commandes (cette requête aurait pu être écrite en utilisant MAX).

6.4 ANY

Format :

```
SELECT attr1, ..., attrn
FROM table1 t1
[JOIN ...]
WHERE attrx [>|<|=|>=|<=] ANY ( SELECT attry
                                FROM table2 t2
                                ...)
```

L'expression WHERE sera évaluée à vraie si la comparaison entre attr_x et l'un des résultats retourné par la requête interne (sur la table t₂) est évaluée à vraie. En conséquence, si la comparaison entre attr_x et chaque résultat n'est pas vérifiée, l'expression sera évaluée à faux.

Il faut également noter que = ANY est équivalent à l'opérateur IN (tiré de [2]).

Bibliographie

- [1] C. MAREE et G. LEDANT, *SQL-2 : Initiation, Programmation*, 2^{ème} édition, Armand Colin, 1994, Paris
- [2] P. DELMAL, *SQL2 – SQL3 : application à Oracle*, 3^{ème} édition, De Boeck Université, 2001, Bruxelles
- [3] Microsoft, MSDN Microsoft Developer Network, <http://msdn.microsoft.com>, consulté en février 2009, Microsoft Corp.
- [4] Diana Lorentz, et al., Oracle Database SQL Reference, 10g Release 2 (10.2), published by Oracle and available at <http://www.oracle.com/pls/db102/homepage>, 2005
- [5] JL. HAINAUT, *Bases de données: concepts, utilisation et développement*, Dunod, 2009, Paris.