Analyse de bases de données SQL avec les agrégats (GROUP BY)

Table des matières

I - Cours **3**

1. Introduction aux agrégats avec GROUP BY....3
   - 1.1. Exercice : La somme finale....3
   - 1.2. Définition de l'agrégation....3
   - 1.3. Exemple d'attribut d'agrégation....5
   - 1.4. Fonctions d'agrégation....6
   - 1.5. Exemple de fonctions d'agrégation....7

2. Approfondissement des agrégats avec GROUP BY et HAVING....9
   - 2.1. Single-Value Rule....9
   - 2.2. Approfondissement de la fonction COUNT....11
   - 2.3. Restriction après agrégation (HAVING)....13
   - 2.4. Ordre de résolution des requêtes SQL....13
   - 2.5. Exercice : Re-représentation de représentants....14

II - Exercices **16**

1. Exercice : Location d'appartements en groupe....16
2. Exercice : Championnat de Formule 1....16
3. Exercice : Questions scolaires....17
4. Quiz : SQL LMD....18

Abréviations **19**

Index **20**

Crédits des ressources **21**

Cours

1. Introduction aux agrégats avec GROUP BY

Objectifs

Savoir réaliser un agrégat en SQL en utilisant les fonctions de regroupement et les clause GROUP BY

1.1. Exercice : La somme finale

Que renvoie la dernière instruction SQL de la séquence ci-dessous ?

```
1 CREATE TABLE R1 (a INTEGER, b INTEGER);
2 CREATE TABLE R2 (b INTEGER);
3 INSERT INTO R1 VALUES (1,1);
4 INSERT INTO R1 VALUES (2,1);
5 INSERT INTO R2 VALUES (1);
6 INSERT INTO R2 VALUES (2);
7 SELECT sum(R1.a) FROM R1, R2 WHERE R1.b=R2.b;
```

1.2. Définition de l'agrégation

Agrégat *Définition*

Un agrégat est un partitionnement horizontal d'une table en sous-tables, en fonction des valeurs d'un ou plusieurs attributs de partitionnement, suivi éventuellement de l'application d'une fonction de calcul à chaque attribut des sous-tables obtenues.

Synonyme : Regroupement

Syntaxe

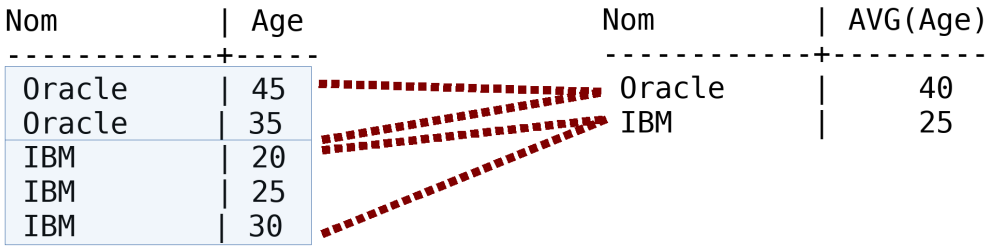
```
1 SELECT liste d'attributs de partionnement à projeter et de fonctions de calcul
2 FROM liste de relations
3 WHERE condition à appliquer avant calcul de l'agrégat
4 GROUP BY liste ordonnée d'attributs de partitionnement
```

1. La table est divisée en sous-ensembles de lignes, avec un sous-ensemble pour chaque valeur différente des attributs de partitionnement projetés dans le SELECT
2. Les fonctions d'agrégation sont appliquées sur les attributs concernés

Exemple

```
1 SELECT Societe.Nom, AVG(Personne.Age)
2 FROM Personne, Societe
3 WHERE Personne.NomSoc = Societe.Nom
4 GROUP BY Societe.Nom
```

Societe.Nom est un ici le seul attribut de partitionnement, donc un sous ensemble est créé pour chaque valeur différente de *Societe.Nom*, puis la moyenne (fonction AVG) est effectuée pour chaque sous-ensemble.



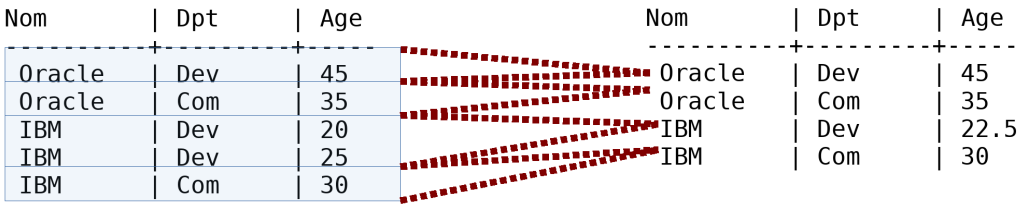
Regroupement avec un attribut et une fonction

Cette requête calcul l'âge moyen du personnel pour chaque société.

Exemple

```sql
SELECT Societe.Nom, Societe.Dpt AVG(Personne.Age)
FROM Personne, Societe
WHERE Personne.NomSoc = Societe.Nom
GROUP BY Societe.Nom, Societe.Dpt
```

Societe.Nom et *Societe.Dpt* sont les deux attributs de partitionnement, donc un sous-ensemble est créé pour chaque valeur différente du couple (*Societe.Nom*,*Societe.Dpt*).



Regroupement avec deux attributs et une fonction

Cette requête calcul l'âge moyen du personnel pour chaque département de chaque société.

Requête inutile

Remarque

La requête est inutile dès lors que l'agrégat est défini sur une valeur unique dans la relation, puisqu'on aura bien une ligne par ligne de la table source.

```sql
SELECT Societe.Nom
FROM Societe
GROUP BY Societe.Nom
```

Exemple d'agrégat inutile (countrycode est une clé de country)

1.3. Exemple d'attribut d'agrégation

Schéma relationnel

```
country(#countrycode:char(2), name:varchar, population:numeric)
city(#code:char(3), countrycode=>country, name:varchar, population:numeric):
```

Schéma de base de données

```
CREATE TABLE country (
countrycode CHAR(2) NOT NULL,
name VARCHAR NOT NULL,
population NUMERIC(3),
PRIMARY KEY (countrycode)
);

CREATE TABLE city (
citycode CHAR(3) NOT NULL,
countrycode CHAR(2) NOT NULL,
name VARCHAR NOT NULL,
population NUMERIC(2,1),
PRIMARY KEY (citycode),
FOREIGN KEY (countrycode) REFERENCES country(countrycode)
);
```

Données

```
INSERT INTO country VALUES ('ES', 'Spain', 46);
INSERT INTO country VALUES ('FR', 'France', 67);
INSERT INTO country VALUES ('DE', 'Germany', 82);

INSERT INTO city VALUES ('BAR', 'ES', 'Barcelona', 1.9);
INSERT INTO city VALUES ('MAD', 'ES', 'Madrid', 3.3);
INSERT INTO city VALUES ('ZAR', 'ES', 'Zaragoza', 0.7);

INSERT INTO city VALUES ('PAR', 'FR', 'Paris', 2.2);
INSERT INTO city VALUES ('LYO', 'FR', 'Paris', 0.5);
INSERT INTO city VALUES ('LLL', 'FR', 'Lille', 0.2);
INSERT INTO city VALUES ('AMN', 'FR', 'Amiens', 0.1);
```

Sélectionne les countrycode existants dans la table city

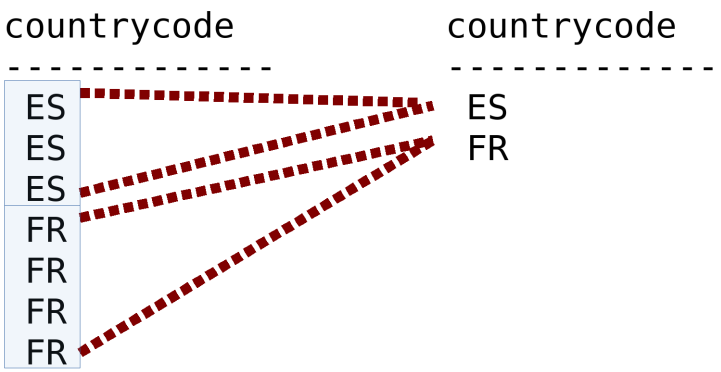
```
SELECT countrycode
FROM city;
```

```
 countrycode
-------------
 ES
 ES
 ES
 FR
 FR
 FR
 FR
```

Sélectionne les countrycode existants dans la table city avec agrégat

```
SELECT countrycode
FROM city
GROUP BY countrycode;
```

```
 countrycode
-------------
 FR
 ES
```



Principe de l'agrégation

1.4. Fonctions d'agrégation

Fonctions d'agrégation — *Définition*

Une fonction d'agrégation (ou fonction de regroupement) s'applique aux valeurs du sous-ensemble d'un agrégat e relation avec pour résultat la production d'une valeur atomique unique (entier, chaîne, date, etc).

Les cinq fonctions prédéfinies sont :

- **COUNT (Relation.Propriété)**

  Renvoie le nombre de valeurs non nulles d'une propriété pour tous les tuples d'une relation ;
- **SUM (Relation.Propriété)**

  Renvoie la somme des valeurs d'une propriété des tuples (numériques) d'une relation ;
- **AVG (Relation.Propriété)**

  Renvoie la moyenne des valeurs d'une propriété des tuples (numériques) d'une relation ;
- **MIN (Relation.Propriété)**

  Renvoie la plus petite valeur d'une propriété parmi les tuples d'une relation .
- **MAX (Relation.Propriété)**

  Renvoie la plus grande valeur d'une propriété parmi les tuples d'une relation.

Fonctions de calcul sans partitionnement — *Attention*

Si une ou plusieurs fonctions de calcul sont appliquées sans partitionnement, le résultat de la requête est un tuple unique.

Exemple

```
1 SELECT Min(Age), Max(Age), Avg(Age)
2 FROM Personne
3 WHERE Qualification='Ingénieur'
```

Comptage d'une relation — *Remarque*

Pour effectuer un comptage sur tous les tuples d'une relation, appliquer la fonction `count` à un attribut de la clé primaire. En effet cet attribut étant non nul par définition, il assure que tous les tuples seront comptés.

1.5. Exemple de fonctions d'agrégation

Schéma relationnel *Rappel*

```
country(#countrycode:char(2), name:varchar, population:numeric)
city(#code:char(3), countrycode=>country, name:varchar, population:numeric):
```

Exemple d'attribut d'agrégation *(cf. p.5)*

a) Agrégat avec un attribut d'agrégation et une fonction d'agrégation

Requête sans agrégat

Sélectionne les *countrycode* et les *citycode* existants dans la table *city*.

```
SELECT countrycode, citycode
FROM city;
```

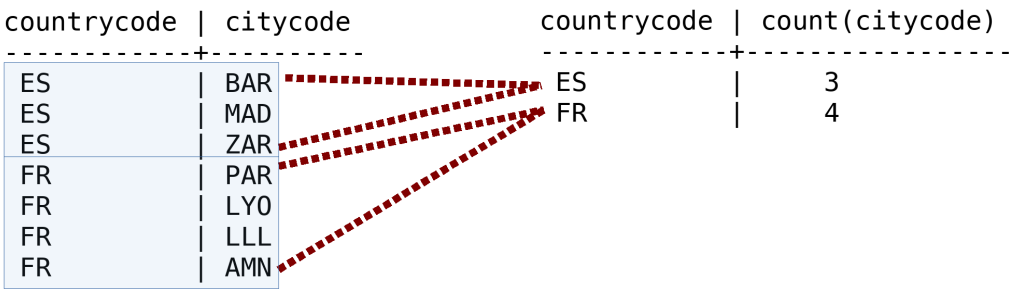
```
 countrycode | citycode
-------------+----------
 ES          | BAR
 ES          | MAD
 ES          | ZAR
 FR          | PAR
 FR          | LYO
 FR          | LLL
 FR          | AMN
```

Requête avec agrégat

- Sélectionne les *countrycode* et les *citycode* existants dans la table *city*,
- puis agrège par valeurs distinctes de *countrycode*.

```
SELECT countrycode, count(citycode)
FROM city
GROUP BY countrycode;
```

```
 countrycode | count
-------------+-------
 FR          |     4
 ES          |     3
```



Regroupement avec fonction d'agrégation

b) Agrégat avec un attribut d'agrégation et deux fonctions d'agrégation

Sélectionne les *countrycode*, les *citycode* et les *populations* existants dans la table *city*.

```
SELECT countrycode, citycode, population
FROM city;
```

```
 countrycode | citycode | population
-------------+----------+------------
 ES          | BAR      |        1.9
 ES          | MAD      |        3.3
 ES          | ZAR      |        0.7
```

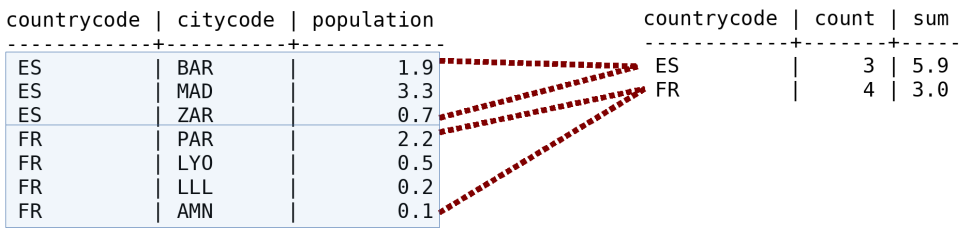
```
 FR          | PAR       |         2.2
 FR          | LYO       |         0.5
 FR          | LLL       |         0.2
 FR          | AMN       |         0.1
```

Requête avec agrégat

- Sélectionne les *countrycode*, les *citycode* et les *populations* existants dans la table *city*,
- puis agrège par valeurs distinctes de *countrycode*
- puis calcule les fonctions *count* et *sum*.

```
SELECT countrycode, count(citycode), sum(population)
FROM city
GROUP BY countrycode;
```

```
 countrycode | count | sum
-------------+-------+-----
 FR          |     4 | 3.0
 ES          |     3 | 5.9
```



Regroupement avec deux fonctions d'agrégation

c) Agrégat sans attribut d'agrégation et avec une fonction d'agrégation

Requête sans agrégat

Sélectionne les *populations* existants dans la table *city*.

```
SELECT population
FROM city;
```

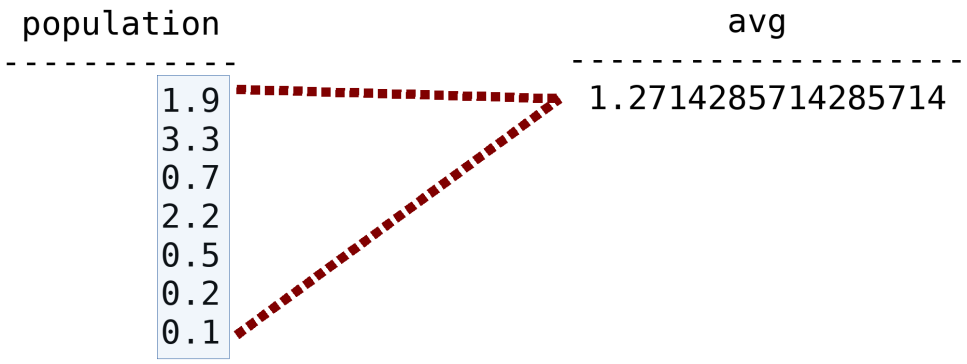
```
 population
------------
        1.9
        3.3
        0.7
        2.2
        0.5
        0.2
        0.1
```

Requête avec agrégat

- Sélectionne les *populations* existants dans la table *city*,
- puis calcule la fonction *avg* (pour *average*, moyenne).

```
SELECT avg(population)
FROM city;
```

```
        avg
--------------------
 1.2714285714285714
```



Regroupement avec fonction d'agrégation sans GROUP BY

2. Approfondissement des agrégats avec GROUP BY et HAVING

Objectifs

Savoir réaliser un agrégat en SQL en utilisant les fonctions de regroupement et les clause GROUP BY et HAVING

2.1. Single-Value Rule

Principe de la "Single-Value Rule" établie par le standard SQL — ***Fondamental***

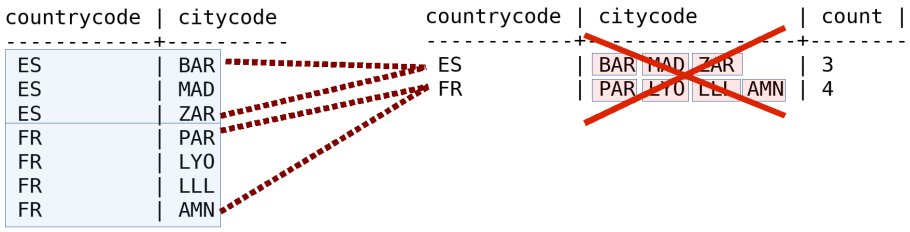
Toute colonne de la clause SELECT doit soit :

- être un attribut d'agrégation (et être également présente dans la clause GROUP BY) ;
- être attribut de calcul (présent dans une fonction d'agrégation.)

Requête illégale — ***Attention***

```
1 SELECT countrycode, citycode, COUNT(citycode)
2 FROM city
3 GROUP BY countrycode;
```

```
1 ERROR:  column "city.citycode" must appear in the GROUP BY clause or be used in an
  aggregate function;
```



Tentative de GROUP BY illégal

La requête est non standard et non logique car on cherche à mettre plusieurs données dans la case *citycode* après le GROUP BY (il y a plusieurs *citycode* par *countrycode*).

Elle sera refusée par tous les SGBD.

Single-Value Rule — ***Complément***

https://mariadb.com/kb/en/sql-99/the-single-value-rule[1]

[1] https://mariadb.com/kb/en/sql-99/the-single-value-rule/

a) Tolérance (de certains SGBD)

Requête non standard tolérée par certains SGBD

```
SELECT citycode, countrycode, AVG(population)
FROM city
GROUP BY citycode;
```

```
 citycode | countrycode |          avg
----------+-------------+------------------------
 LLL      | FR          | 0.20000000000000000000
 BAR      | ES          | 1.90000000000000000000
 PAR      | FR          |     2.2000000000000000
 LYO      | FR          | 0.50000000000000000000
 ZAR      | ES          | 0.70000000000000000000
 AMN      | FR          | 0.10000000000000000000
 MAD      | ES          |     3.3000000000000000
```

La requête est non standard, même si c'est logiquement acceptable, car il n'y a qu'un *countrycode* par *citycode* ici, *city* étant une clé.

Certains SGBD comme Postgres acceptent donc cette syntaxe, en ajoutant implicitement l'attribut qui manque au GROUP BY, car il n'y a pas d'ambiguïté si l'on analyse le graphe des *DF∗* : *citycode* → *countrycode*. Mais le standard impose d'être explicite, car le SGBD a le droit de ne pas le deviner.

b) Tolérance partielle

Requête légale

```
SELECT ci.countrycode, co.name, count(*)
FROM city ci JOIN country co
ON ci.countrycode=co.countrycode
GROUP BY ci.countrycode, co.name;
```

```
 countrycode |  name   | count
-------------+---------+-------
 FR          | France |     4
 ES          | Spain  |     3
```

Requête non standard tolérée par certains SGBD

```
SELECT co.countrycode, co.name, count(*)
FROM city ci JOIN country co
ON ci.countrycode=co.countrycode
GROUP BY co.countrycode;
```

La requête est non standard, car *co.name* est dans le SELECT mais pas dans le GROUP BY. Comme dans l'exemple précédent certains SGBD accepteront cette requête grâce à la DF évidente : *co.countrycode* → *co.name*

Requête non standard non tolérée — ***Attention***

Mais si l'on utilise à présent ***ci.countrycode*** à la place de *co.countrycode*...

```
SELECT ci.countrycode, co.name, count(*)
FROM city ci JOIN country co
ON ci.countrycode=co.countrycode
GROUP BY ci.countrycode;
```

```
ERROR:  column "co.name" must appear in the GROUP BY clause or be used in an
aggregate function
```

La requête est non standard, *co.name* doit être mentionné dans la clause GROUP BY. Logiquement on pourrait juger cela superflu, car par transitivité *ci.countrycode* → *co.name*, mais le SGBD (ici Postgres) ne va pas jusque là.

Conseil

Il est donc conseillé d'appliquer le *Single-Value Rule* et de toujours expliciter tous les attributs dans le GROUP BY, pour éviter la dépendance au SGBD ou bien les erreurs liées à des variations mineures de syntaxe.

2.2. Approfondissement de la fonction COUNT

COUNT(*) **Syntaxe**

- `COUNT(*)` renvoie le nombre de lignes de la relation.
- `COUNT(a)` renvoie le nombre de valeurs **non nulles** de la relation (donc `COUNT(*)` et `COUNT(a)` sont équivalents si a est non null)
- `COUNT (DISTINCT a)` renvoie le nombre de valeurs **non nulles** et **distinctes** de la relation

Exemple

```
1 CREATE TABLE t (
2 pk INTEGER PRIMARY KEY,
3 a VARCHAR(1),
4 b VARCHAR(1));
5 INSERT INTO t VALUES (1, 'a', 'x');
6 INSERT INTO t VALUES (2, 'a', 'y');
7 INSERT INTO t VALUES (3, 'b', NULL);
```

table t

#pk	a NOT NULL	b
1	a	x
2	a	y
3	b	

- `SELECT COUNT(*) FROM t:3`
- ≡ `SELECT COUNT(pk) FROM t:3`
- ≡ `SELECT COUNT(a) FROM t:3`
- `SELECT COUNT (DISTINCT a) FROM t:2`
- ≡ `SELECT COUNT(*) FROM t GROUP BY a:2`

Attention

Pour des raisons de performance, on préfère l'usage d'un `GROUP BY` à `COUNT (DISTINCT a)`. On utilise ce dernier que s'il n'existe pas de solution ne mobilisant qu'une seule requête.

Exemple

```
1 SELECT COUNT(*) AS nb_total, COUNT (DISTINCT a) AS nb_a FROM t;
```

```
1  nb_total | nb_a
2 ----------+------
3         3 |    2
```

COUNT(*) et résultat de requête

Attention

Il faut prendre garde à l'usage de `COUNT(*)` dès que l'on adresse des requêtes portant sur plusieurs tables, en effet cela comptera **toutes** les lignes produite par le produit cartésien, ce qui n'est pas toujours ce que l'on souhaite.

On préférera compter un attribut non nul.

COUNT(*) et jointure externe

Exemple

Soit la séquence suivante permettant d'instancier t1 et t2, tel que référence t1 référence t2 et qu'il existe un élément de t2 qui n'est jamais référencé.

```
CREATE TABLE t2 (
pk INTEGER PRIMARY KEY,
a VARCHAR
);

CREATE TABLE t1 (
pk VARCHAR PRIMARY KEY,
fk INTEGER REFERENCES t2(pk)
);

INSERT INTO t2 VALUES (1, 'x');
INSERT INTO t2 VALUES (2, 'x');

INSERT INTO t1 VALUES ('a',1);
INSERT INTO t1 VALUES ('b',1);
```

Si l'on utilise un `COUNT(*)` alors on obtient la valeur 1 pour `b` au lieu du 0 escompté :

- la ligne `(b,NULL)` est ajoutée au résultat parce que `b` n'est jamais référencé (principe de la jointure externe)
- étant donné que la jointure externe a lieu **avant** le regroupement et que `COUNT(*)` compte toutes les lignes

```
SELECT t2.pk, COUNT(*)
FROM t2
LEFT JOIN t1 ON t1.fk=t2.pk
GROUP BY t2.pk;
```

```
 pk | count
----+-------
  1 |     2
  2 |     1
```

Si l'on utilise un `COUNT(a)` sur la clé de la table t1, celle-ci étant nulle dans le cas des lignes ajoutées par la jointure externe, le fonctionnement est celui attendu.

```
SELECT t2.pk, COUNT(t1.pk)
FROM t2
LEFT JOIN t1 ON t1.fk=t2.pk
GROUP BY t2.pk;
```

```
 pk | count
----+-------
  1 |     2
  2 |     0
```

Complément

https://sql.sh/fonctions/agregation/count

2.3. Restriction après agrégation (HAVING)

Définition

La clause HAVING permet d'effectuer une second restriction après l'opération d'agrégation.

Syntaxe

```
SELECT liste d'attributs de partionnement à projeter et de fonctions de calcul
FROM liste de relations
WHERE condition à appliquer avant calcul de l'agrégat
GROUP BY liste ordonnée d'attributs de partitionnement
HAVING condition sur les fonctions de calcul
```

Exemple

```
SELECT Societe.Nom, AVG(Personne.Age)
FROM Personne, Societe
WHERE Personne.NomSoc = Societe.Nom
GROUP BY Societe.Nom
HAVING COUNT(Personne.NumSS) > 2
```

Cette requête calcul l'âge moyen du personnel pour chaque société comportant plus de 2 salariés.

2.4. Ordre de résolution des requêtes SQL

Fondamental

L'ordre de résolution standard d'une requête SQL est :

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Usage des alias dans le GROUP BY ou le HAVING : requête non standard, acceptée par certains SGBD

Attention

```
SELECT substring(name,1,1) AS initiale, count(citycode)
FROM city
GROUP BY initiale;
```

```
 initiale | count
----------+-------
 L        |     1
 Z        |     1
 B        |     1
 M        |     1
 P        |     2
 A        |     1

```

La requête n'est pas standard, les résolutions de l'alias et de la fonction *substring* sont dans le SELECT, qui est postérieur à la résolution du GROUP BY. Postgres acceptera néanmoins cette syntaxe, ce qui évite de créer une vue ou d'utiliser une sous-requête.

Restriction *Remarque*

Une restriction peut être appliquée avant calcul de l'agrégat, au niveau de la clause WHERE, portant ainsi sur la relation de départ, mais aussi après calcul de l'agrégat sur les résultats de ce dernier, au niveau de la clause HAVING.

Utilisation de fonctions d'agrégation *Remarque*

Les fonctions d'agrégation peuvent être utilisées dans la claude HAVING ou dans la clause ORDER BY. Les fonctions **ne peuvent pas** être utilisées dans la clause WHERE (qui est résolu avant le GROUP BY).

2.5. Exercice : Re-représentation de représentants

[30 minutes]

Soit le schéma relationnel suivant :

```
REPRESENTANTS (#NR, NOMR, VILLE)
PRODUITS (#NP, NOMP, COUL, PDS)
CLIENTS (#NC, NOMC, VILLE)
VENTES (#NR=>REPRESENTANTS(NR), #NP=>PRODUITS(NP), #NC=>CLIENTS(NC), QT)
```

Écrire en SQL les requêtes permettant d'obtenir les informations suivantes.

```sql
/* Les requêtes peuvent être testées dans un SGBDR, en créant une base de données
avec le script SQL suivant */

/*
DROP TABLE VENTES ;
DROP TABLE CLIENTS ;
DROP TABLE PRODUITS ;
DROP TABLE REPRESENTANTS ;
*/

CREATE TABLE REPRESENTANTS (
  NR INTEGER PRIMARY KEY,
  NOMR VARCHAR,
  VILLE VARCHAR
);

CREATE TABLE PRODUITS (
  NP INTEGER PRIMARY KEY,
  NOMP VARCHAR,
  COUL VARCHAR,
  PDS INTEGER
);

CREATE TABLE CLIENTS (
  NC INTEGER PRIMARY KEY,
  NOMC VARCHAR,
  VILLE VARCHAR
);

CREATE TABLE VENTES (
  NR INTEGER REFERENCES REPRESENTANTS(NR),
  NP INTEGER REFERENCES PRODUITS(NP),
  NC INTEGER REFERENCES CLIENTS(NC),
  QT INTEGER,
  PRIMARY KEY (NR, NP, NC)
);

INSERT INTO REPRESENTANTS (NR, NOMR, VILLE) VALUES (1, 'Stephane', 'Lyon');
```

```sql
INSERT INTO REPRESENTANTS (NR, NOMR, VILLE) VALUES (2, 'Benjamin', 'Paris');
INSERT INTO REPRESENTANTS (NR, NOMR, VILLE) VALUES (3, 'Leonard', 'Lyon');
INSERT INTO REPRESENTANTS (NR, NOMR, VILLE) VALUES (4, 'Antoine', 'Brest');
INSERT INTO REPRESENTANTS (NR, NOMR, VILLE) VALUES (5, 'Bruno', 'Bayonne');

INSERT INTO PRODUITS (NP, NOMP, COUL, PDS) VALUES (1, 'Aspirateur', 'Rouge', 3546);
INSERT INTO PRODUITS (NP, NOMP, COUL, PDS) VALUES (2, 'Trottinette', 'Bleu', 1423);
INSERT INTO PRODUITS (NP, NOMP, COUL, PDS) VALUES (3, 'Chaise', 'Blanc', 3827);
INSERT INTO PRODUITS (NP, NOMP, COUL, PDS) VALUES (4, 'Tapis', 'Rouge', 1423);

INSERT INTO CLIENTS (NC, NOMC, VILLE) VALUES (1, 'Alice', 'Lyon');
INSERT INTO CLIENTS (NC, NOMC, VILLE) VALUES (2, 'Bruno', 'Lyon');
INSERT INTO CLIENTS (NC, NOMC, VILLE) VALUES (3, 'Charles', 'Compiègne');
INSERT INTO CLIENTS (NC, NOMC, VILLE) VALUES (4, 'Denis', 'Montpellier');
INSERT INTO CLIENTS (NC, NOMC, VILLE) VALUES (5, 'Emile', 'Strasbourg');

INSERT INTO VENTES (NR, NP, NC, QT) VALUES (1, 1, 1, 1);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (1, 1, 2, 1);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (2, 2, 3, 1);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (4, 3, 3, 200);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 4, 2, 190);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (1, 3, 2, 300);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 1, 2, 120);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 1, 4, 120);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 4, 4, 2);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 1, 1, 3);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 4, 1, 5);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 1, 3, 1);
INSERT INTO VENTES (NR, NP, NC, QT) VALUES (3, 1, 5, 1);
```

Question 1

Le nombre de clients.

Question 2

Le nombre de produits de couleur rouge.

Question 3

Le nombre de clients par ville.

Question 4

La quantité totale des produits rouge vendus par chaque représentant.

Question 5

La quantité totale de produits rouges vendus pour chaque représentant ayant vendu plus de 5 fois des produits rouges (ayant réalisé plus de 5 ventes différentes de produits rouges).

Exercices

1. Exercice : Location d'appartements en groupe

[20 min]

Soit le schéma relationnel suivant gérant le fonctionnement d'une agence de location d'appartements.

```
APPARTEMENT(#code_appt:String, adresse:String, type:{studio,F1,F2,F3,F4,F5+},
prix_loyer:Real)
LOCATAIRE(#code_loc:String, nom:String, prenom:String)
LOCATION(#code_loc=>Locataire, #code_appt=>Appartement)
PAIEMENT_LOYER(#code_loc=>Locataire, #code_appt=>Appartement, #date_payement:Date,
prix_paye:Real)
```

Question 1

En SQL afficher le nombre d'appartements de chaque type, uniquement pour les types qui commencent par la lettre F.

Question 2

En SQL afficher le total payé par locataire (avec son code, nom et prenom) pour l'ensemble de ses appartements.

Question 3

En SQL afficher les locataires (code uniquement) qui louent au moins 2 appartements, en précisant le nombre d'appartements loués et la moyenne des loyers, et trié par ordre décroissant de cette moyenne.

2. Exercice : Championnat de Formule 1

[20 min]

La base de données suivante permet de gérer les résultats des courses de Formule 1 dans un championnat.

```
CHAMPIONNAT(#nom:string, annee:integer)
CIRCUIT(#nom:string, ville:string)
COURSE(#nom:string,circuit=>CIRCUIT(nom), championnat=>CHAMPIONNAT(nom))
SPONSOR(#nom:string)
ECURIE(#nom:string, devise:string, couleur:string, sponsor=>SPONSOR(nom))
PILOTE(#id:integer, nom:string, prenom:string, ecurie=>ECURIE(nom))
TOUR(#pilote => PILOTE(id), #course => COURSE(nom), #num:integer, duree:integer)
```

Question 1

En algèbre relationnel et en SQL, afficher la liste de tous les pilotes dont le nom commence par la lettre 'D'.

Question 2

En SQL, afficher le nombre de pilotes par écurie en classant les résultats par ordre alphabétique des noms des écuries.

Question 3

En algèbre relationnel et en SQL, afficher les noms des sponsors qui ne sont pas liés à une écurie.

Question 4

En SQL, afficher le numéro, nom, prénom et écurie, avec leur temps moyen par tour, des pilotes participant à la course `Daytonutc 500` ; mais en ne conservant que les pilotes qui ont effectués au moins deux tours de piste, et en classant le résultat par temps moyen décroissant.

3. Exercice : Questions scolaires

[30 min]

Soit le schéma relationnel suivant gérant le fonctionnement d'une école et les notes des élèves.

```
CLASSE(#intitule)
MATIERE(#intitule)
ELEVE(#login, nom, prenom, age, classe=>CLASSE)
ENSEIGNANT(#login, nom, prenom, mail, matiere=>MATIERE)
ENSEIGNE(#enseignant=> ENSEIGNANT, #classe=>CLASSE)
NOTE(#eleve=>ELEVE, #enseignant=>ENSEIGNANT, #date_examen, note)

Contrainte : un enseignant ne peut mettre une note à un élève que si celui-ci se
trouve dans une classe dans laquelle il enseigne.
```

NB : Nous n'utiliserons pas de sous-requêtes.

Question 1

En algèbre relationnel **et** en SQL, afficher la liste des tous les étudiants dont le nom commence par A.

Question 2

En algèbre relationnel **et** en SQL, afficher les noms et prénoms des enseignants qui n'enseignent à aucune classe.

Question 3

En SQL, affichez le nombre d'étudiants enregistrés en "Terminale S 2".

Question 4

En SQL, affichez les logins, noms, prénoms, classes et moyennes des élèves en cours de "Mathématiques", par ordre décroissant de moyenne, à condition qu'ils aient au minimum 2 notes dans cette matière.

Question 5

En SQL, à des fins de statistiques, nous souhaitons rechercher par enseignant et par classe, les classes qui n'ont pas la moyenne générale, et afficher pour celles-ci : le nom, prénom et mail de l'enseignant en question, la matière enseignée, la classe, la moyenne d'âge des étudiants avec les extrêmes (minimum et maximum), la moyenne générale de la classe avec les valeurs extrêmes, ainsi que le nombre d'étudiants présents dans cette classe ; le tout classé par ordre alphabétique de classe, puis de nom et de prénom de l'enseignant.

Indice :

On fera l'hypothèse que tous les étudiants d'une classe ont le même nombre de notes (pas d'absence aux examens).

4. Quiz : SQL LMD

Exercice 1

Soit les deux relations suivantes UV1 et UV2 représentant respectivement les places disponibles dans les cours de l'UTC et les inscriptions effectives pour les cours ouverts ce semestre :

Nom	Nombre
NF17	168
NF18	48
NF19	48
NF20	48
NF21	48
NF22	168

UV1 : Place disponibles

Nom	Nombre
NF17	182
NF18	46
NF19	44
NF22	98

UV2 : Inscriptions

Relations UV1 et UV2

Compléter la requête SQL suivante pour qu'elle retourne pour **tous** les cours avec leur nom, plus le nombre d'inscrits en excès ou en défaut pour les cours ouverts et NULL pour les cours fermés.

SELECT UV1.Nom,

FROM UV1 UV2

ON

Exercice 2

Soit les deux relations suivantes UV1 et UV2 représentant respectivement les places disponibles dans les cours de l'UTC et les inscriptions effectives pour les cours ouverts ce semestre :

Nom	Nombre
NF17	168
NF18	48
NF19	48
NF20	48
NF21	48
NF22	168

UV1 : Place disponibles

Nom	Nombre
NF17	182
NF18	46
NF19	44
NF22	98

UV2 : Inscriptions

Relations UV1 et UV2

Compléter la requête SQL suivante pour qu'elle retourne, pour les cours ouverts uniquement, le nombre d'inscrits moyen selon le nombre de places (nombre d'inscrits moyens pour les cours de 168 places, pour ceux de 48 places, etc.)

SELECT UV1.Nombre,

FROM UV1 UV2

ON

Exercice 3

Que renvoie la dernière instruction SQL de la séquence ci-dessous ?

```
1 CREATE TABLE R (a INTEGER, b INTEGER);
2 INSERT INTO R VALUES (1,1);
3 INSERT INTO R VALUES (1,2);
4 INSERT INTO R VALUES (3,3);
5 SELECT sum(R1.b) FROM R R1, R R2 WHERE R1.a=R2.a;
```

Abréviations

DF : Dépendance Fonctionnelle

Index

Agrégat 3, 3, 5, 6, 7, 9, 9, 13, 13
Agrégation 9, 13
Calcul 6, 7, 9, 13
Fonction 6, 7, 9, 13
GROUP BY 3, 5, 9, 13, 13
HAVING 13
Langage 3, 9
LMD 3, 9
Question 3, 9
Regroupement 3, 5
Requête 3, 9
SQL 3, 9

Crédits des ressources

Regroupement avec un attribut et une fonction p. 4
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat

Regroupement avec deux attributs et une fonction p. 4
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat

Principe de l'agrégation p. 6
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat

Regroupement avec fonction d'agrégation p. 7
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat

Regroupement avec deux fonctions d'agrégation p. 8
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat

Regroupement avec fonction d'agrégation sans GROUP BY p. 9
http://creativecommons.org/licenses/by-sa/4.0/fr/, Stéphane Crozat