

# Démythifier le GROUP BY

par Roland Bouman Cédric Duprez (traducteur)

Date de publication : 28 mars 2009

Dernière mise à jour :

Cet article est la traduction de *Debunking GROUP BY Myths* (disponible [ici](#)) et a pour but de vous expliquer le comportement de MySQL vis à vis de la clause SQL GROUP BY.

---

Introduction.....	3
1 - La clause GROUP BY.....	3
2 - Calcul d'agrégats sur des groupes de lignes.....	4
3 - Les difficultés avec GROUP BY.....	5
4 - Comprendre le problème.....	6
5 - Eviter le problème.....	6
6 - Inclure ONLY_FULL_GROUP_BY dans le sql_mode de MySQL.....	7
7 - Que dit « la » norme SQL ?.....	8
8 - Dépendances fonctionnelles.....	8
8-A - Dépendance fonctionnelle et normalisation.....	9
8-B - Dépendances fonctionnelles et GROUP BY.....	9
9 - Pourquoi donc voudrais-je faire cela ?.....	10
9-A - Inconvénients ?.....	11
9-B - Avantages.....	11
9-C - Agrégation sur des colonnes fonctionnellement dépendantes.....	14
Conclusion.....	14

## Introduction

Il existe une légende populaire à propos de la clause SQL **GROUP BY**. Cette légende soutient que la "norme SQL" impose, dans une requête, que les colonnes référencées dans la liste du **SELECT** apparaissent également dans la clause **GROUP BY**, à moins que ces colonnes n'apparaissent exclusivement dans une expression d'agrégation. MySQL est souvent accusé de violer cette norme.

Dans cet article, je vais tenter de briser ce mythe et d'apporter par la même occasion un regard plus mesuré sur le traitement du **GROUP BY** dans MySQL.

Pour ce faire, je vais d'abord démontrer que MySQL peut être configuré pour accepter des clauses **GROUP BY** qui ne comportent que des colonnes dans la liste du **SELECT** contenues dans des fonctions d'agrégation, rapprochant ainsi le comportement de MySQL de celui des autres célèbres SGBDR.

Ensuite, je vais montrer qu'il est très important de bien préciser à quelle version de la norme SQL on fait référence. Les deux versions les plus récentes présentent une définition plus sophistiquée de la relation entre les expressions contenues dans le **GROUP BY** et la liste du **SELECT**. Contrairement à une croyance populaire, ces normes n'imposent pas noir sur blanc que toutes les colonnes non contenues dans les fonctions d'agrégation de la liste du **SELECT** apparaissent dans la clause **GROUP BY**.

Enfin, j'emploierai un exemple simple mais concret pour illustrer de manière informelle ce que je pense que les dernières versions de la norme SQL essaient d'exprimer. Avec un peu de chance, j'arriverai à vous convaincre du bien-fondé de ne pas toujours inclure aveuglément toutes les colonnes non agrégées de la liste du **SELECT** dans la clause **GROUP BY**.

Avant de plonger dans le vif du sujet, je vais commencer par une brève présentation du **GROUP BY**, pour ceux qui ne sont pas familiers de cette notion. En introduction, je présenterai pourquoi la plupart des SGBD imposent que toutes les colonnes non agrégées et référencées dans la liste du **SELECT** apparaissent dans la clause **GROUP BY**, et pourquoi les utilisateurs rencontrent parfois des difficultés à cause de la façon dont MySQL traite la clause **GROUP BY**.

## 1 - La clause GROUP BY

Allons au fait ! Qu'est-ce que la clause **GROUP BY** et que fait-elle ?

La clause **GROUP BY** est un élément optionnel des requêtes SQL **SELECT**. Syntaxiquement, la clause **GROUP BY** est constituée de la séquence de mots clefs **GROUP BY**, suivie d'une liste d'expressions (scalaires) séparées par des virgules. Si une requête **SELECT** contient une clause **GROUP BY**, celle-ci doit figurer après la clause **WHERE** (si la clause **WHERE** est omise, la clause **GROUP BY** suit immédiatement la clause **FROM**).

Quand elle est présente, la clause **GROUP BY** signifie que les données provenant d'un ensemble intermédiaire issu d'une requête doivent être divisées selon certains groupes, ne retournant qu'une seule ligne pour chacun de ces groupes. La liste des expressions précisées dans le **GROUP BY** définit la façon dont le regroupement doit se faire. Toutes les lignes présentant la même combinaison de valeurs pour toutes les expressions précisées dans le **GROUP BY** sont dans le même groupe.

Prenons comme exemple quelques requêtes simples pour illustrer les effets de la clause **GROUP BY** (sur ces exemples, j'utilise la table **pet** de la base de données **menagerie**). La requête suivante ramène toutes les lignes de la table **pet** :

```
SELECT *
FROM menagerie.pet;
```

La requête ramène un résultat qui ressemble à ceci :

name	owner	species	sex	birth	death
Fluffy	Harold	cat	f	1993-02-04	NULL
Claws	Gwen	cat	m	1994-03-17	NULL
Buffy	Harold	dog	f	1989-05-13	NULL
Fang	Benny	dog	m	1990-08-27	NULL
Bowser	Diane	dog	m	1979-08-31	1995-07-29
Chirpy	Gwen	bird	f	1998-09-11	NULL

Whistler	Gwen	bird	NULL	1997-12-09	NULL	
Slim	Benny	snake	m	1996-04-29	NULL	
Puffball	Diane	hamster	f	1999-03-30	NULL	
+-----+-----+-----+-----+-----+-----+						

(La clause **ORDER BY** n'ayant pas été précisée, les lignes sont renvoyées dans un ordre déterminé par la base de données, votre résultat pourrait être légèrement différent de celui ci-dessus. Cependant, pour cet exemple, seules les lignes importent, pas leur ordre).

Maintenant, supposons que l'on souhaite faire des groupes pour chaque espèce. L'ajout de la clause **GROUP BY** permet cela :

```
SELECT species
FROM menagerie.pet
GROUP BY species -- crée un groupe pour chaque espèce
```

Cette requête retourne le résultat suivant :

+-----+
species
+-----+
bird
cat
dog
hamster
snake
+-----+

A première vue, il semble que la clause **GROUP BY** ne fait rien de plus qu'un scan de chaque occurrence unique dans la colonne **species** et les renvoie. Toutefois, il vaut mieux considérer que chaque ligne du **GROUP BY** correspond à une ligne résumant un groupe de lignes qui ont la même valeur dans la colonne **species**. Ainsi, dans le cas présent, la ligne **bird** représente le groupe des oiseaux ("Chirpy" et "Wistler") ; la ligne **cat** représente le groupe des chats ("Fluffy" et "Claws"), et ainsi de suite.

## 2 - Calcul d'agrégats sur des groupes de lignes

Une requête avec **GROUP BY** permet l'utilisation de **fonctions d'agrégation** sur une série de lignes réunies dans chaque groupe par la clause **GROUP BY**. Une fonction d'agrégation peut traiter des expressions pour chaque ligne dans un groupe de lignes afin de retourner une seule valeur. Quelques fonctions d'agrégation standard bien connues : **COUNT**, **MIN**, **MAX** et **SUM** (les fonctions d'agrégation peuvent également être employées sans clause **GROUP BY**, auquel cas le jeu de données intermédiaire est traité comme un seul grand groupe. Essayez d'imaginer l'effet d'une opération **GROUP BY** avec une liste **GROUP BY** vide : la requête ne va retourner qu'une seule ligne qui résume toutes les lignes du jeu de données intermédiaire).

Pour compléter notre exemple précédent avec la clause **GROUP BY**, l'exemple suivant montre l'effet de quelques fonctions d'agrégation :

```
SELECT species
, GROUP_CONCAT(name) -- crée une liste d'animaux par espèce
, COUNT(*) -- décompte les animaux par espèce
, MIN(birth) -- date de naissance du plus vieil animal par espèce
, MAX(birth) -- date de naissance du plus jeune animal par espèce
FROM menagerie.pet
GROUP BY species
```

Cet exemple inclut également l'utilisation d'une fonction d'agrégation spécifique à MySQL, **GROUP\_CONCAT**, qui va se révéler très pratique pour montrer les effets de la clause **GROUP BY**.

Le résultat ressemble à ceci :

--

species	GROUP_CONCAT(name)	COUNT(*)	MIN(birth)	MAX(birth)
bird	Chirpy,Whistler	2	1997-12-09	1998-09-11
cat	Fluffy,Claws	2	1993-02-04	1994-03-17
dog	Buffy,Fang,Bowser	3	1979-08-31	1990-08-27
hamster	Puffball	1	1999-03-30	1999-03-30
snake	Slim	1	1996-04-29	1996-04-29

Une fois de plus, on voit une ligne pour chaque groupe de lignes ayant la même valeur dans la colonne `species`, mais cette fois-ci, on peut également constater l'effet du traitement des lignes individuelles pour chaque espèce par une fonction d'agrégation :

- La fonction `GROUP_CONCAT` a été appliquée à la colonne `name`. Pour chaque espèce dans notre table d'animaux, il peut y avoir plusieurs animaux, et `GROUP_CONCAT` concatène leurs noms, séparant par défaut le nom de chacun par une virgule. Ainsi, dans cet exemple, l'expression `GROUP_CONCAT` fournit la constitution de chaque groupe d'animaux d'une même espèce ;
- La fonction `COUNT` est employée avec un joker `*`, lui demandant de compter le nombre de lignes associé à chaque groupe. On peut vérifier cela avec la colonne précédente et l'on voit immédiatement que le nombre de lignes au sein de chaque groupe est cohérent avec le nombre de noms concaténés par la fonction `GROUP_CONCAT` ;
- Les fonctions `MIN` et `MAX` sont appliquées à la colonne `birth` et ramène respectivement, pour chaque espèce, la date de naissance du plus vieil animal (`MIN(birth)` la date de naissance qui est plus petite que toutes les autres dates de naissance) et du plus jeune (`MAX(birth)` la date de naissance qui est plus grande que toutes les autres dates de naissance).

Les fonctions d'agrégation sont donc appliquées à des expressions issues d'un groupe de lignes, et ont pour effet de "résumer" (agrèger) le groupe en une seule valeur. La plupart des fonctions d'agrégation calculent ou déterminent des statistiques qui permettent de caractériser le groupe comme un tout. La fonction `GROUP_CONCAT`, propre à MySQL, est une exception : elle énumère simplement tous les membres d'un groupe passé à la fonction, et ainsi n'est pas à proprement parler une fonction statistique. Toutefois, elle possède la principale propriété des fonctions d'agrégation, à savoir la capacité à transformer les expressions d'un groupe de lignes en une seule valeur.

### 3 - Les difficultés avec GROUP BY

Jusque là, nous avons vu quelques exemples avec `GROUP BY` parfaitement limpides. Il est pourtant facile de rencontrer des difficultés avec `GROUP BY`. Prenez l'exemple suivant :

```
SELECT species
,      MIN(birth)  -- date de naissance du plus vieil animal par espèce
,      MAX(birth)  -- date de naissance du plus jeune animal par espèce
,      birth       -- date de naissance de ... oh oh...!
FROM   menagerie.pet
```

Cette requête ressemble à la précédente, où l'on avait calculé un certain nombre d'agrégats pour chaque groupe d'animaux appartenant à la même espèce. Cependant, cette fois-ci, on inclut aussi la colonne `birth` dans la liste du `SELECT`.

Si on tente de lancer cette requête sous Oracle, on récupère une erreur :

```
SQL> SELECT species
2 ,      MIN(birth)
3 ,      MAX(birth)
4 ,      birth
5 FROM   sakila.pet
6 GROUP BY species;
,
*
ERROR at line 4:
```

ORA-00979: not a GROUP BY expression

Le lancement de cette requête sous MySQL renvoie toutefois un résultat, qui ressemble à quelque chose comme :

```

SELECT  species
,       MIN(birth)  -- date de naissance du plus vieil animal par espèce
,       MAX(birth)  -- date de naissance du plus jeune animal par espèce
,       birth       -- date de naissance de ... oh oh...!
FROM    menagerie.pet

+-----+-----+-----+-----+
| species | MIN(birth) | MAX(birth) | birth |
+-----+-----+-----+-----+
| bird   | 1997-12-09 | 1998-09-11 | 1998-09-11 |
| cat    | 1993-02-04 | 1994-03-17 | 1993-02-04 |
| dog    | 1979-08-31 | 1990-08-27 | 1989-05-13 |
| hamster| 1999-03-30 | 1999-03-30 | 1999-03-30 |
| snake  | 1996-04-29 | 1996-04-29 | 1996-04-29 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

Que se passe-t-il donc ? Pourquoi une telle différence ? En fait, comment MySQL se comporte-t-il dans le cas présent ? Parfois, la colonne **birth** ramène une ligne qui a l'air d'être sa valeur maximale au sein de l'espèce (première ligne), et parfois on voit la valeur minimale (deuxième ligne). On trouve même un cas où la valeur retournée se situe entre le minimum et le maximum (ligne 3). Comment expliquer un comportement apparemment aussi aléatoire ?

## 4 - Comprendre le problème

Il n'est pas trop compliqué de comprendre ce qui se passe ici. Tout ce que nous avons à faire, c'est de revenir à notre explication de l'effet de la clause **GROUP BY**, et regarder comment elle s'applique à notre dernière requête.

Nous avons déjà expliqué que la clause **GROUP BY** ramenait une ligne de chaque groupe de lignes présentes dans le résultat intermédiaire, et que les groupes sont déterminés par la liste définie dans la clause **GROUP BY**. Ainsi, dans le cas présent, nous créons une ligne de résultats pour chaque groupe de lignes qui appartiennent à la même espèce, puisque la liste du **GROUP BY** ne contient que la colonne **species**. Mais comme il y a plusieurs animaux qui peuvent appartenir à une espèce donnée, la colonne **birth** peut avoir (et a souvent) une valeur différente pour chaque ligne dans un groupe d'espèce donné.

Une fois qu'on a compris cela, on est en droit de se demander : puisqu'il y a plusieurs valeurs dans la colonne **birth** pour une espèce donnée, laquelle sera retournée ? Qu'a-t-on fait en spécifiant la colonne **birth** dans la liste du **SELECT** ? Il n'y a pas de réponse juste à cette question. Il est certainement possible de ne sélectionner qu'une seule valeur parmi toutes celles de la colonne **birth** : en fait, c'est exactement ce que fait MySQL. Cependant, il est impossible de définir laquelle des valeurs possibles est choisie. En fait, ça n'a pas de sens de vouloir mélanger les valeurs des lignes qui appartiennent à un groupe avec le groupe lui-même. C'est pourquoi ça n'a pas non plus de sens de faire figurer la colonne **birth** dans la liste du **SELECT**.

Une autre manière de considérer la question revient à dire que le niveau des valeurs des espèces est différent (et par-là même incompatible) avec le niveau des valeurs de **birth**. Ça ne signifie pas pour autant qu'on ne puisse pas accéder aux valeurs de la colonne **birth**, ça signifie plutôt que l'on doit utiliser une fonction d'agrégation pour récupérer la "bonne" valeur dans le groupe de ces valeurs.

## 5 - Eviter le problème

Qu'en est-il d'un comportement comme celui d'Oracle ? Est-ce que ça a plus de sens de renvoyer un message d'erreur plutôt que de retourner une donnée qui n'a pas de sens ? Dans un certain sens, la plupart des gens seront d'accord. Bien entendu, le message d'erreur lui-même est plutôt surprenant :

```

,       birth
*
ERROR at line 4:
ORA-00979: not a GROUP BY expression

```

Il semble suggérer que le problème provient du fait que la colonne `birth` dans la liste du `SELECT` n'est pas incluse dans la clause `GROUP BY`. A son tour, cela soulève la question de savoir si le problème serait résolu en incluant la colonne `birth` dans la clause `GROUP BY`.

Le fait d'ajouter la colonne `birth` dans la clause `GROUP BY` permet certainement de se débarrasser du message d'erreur. Cependant, beaucoup d'utilisateurs débutants dans le SQL ne comprendraient pas que cela produit une requête différente. Le `GROUP BY species` original produit un groupe pour chaque espèce, alors que `GROUP BY species, birth` crée un groupe pour chaque combinaison de valeurs `species` et `birth` (ne correspondant sûrement pas à ce qui est voulu).

D'un autre côté, on ne peut pas attendre du SGBD qu'il sache ce à quoi on pensait en incluant `birth` dans la liste du `SELECT` en premier lieu. Donc, bien que le message d'erreur puisse sembler surprenant, il est toujours préférable que de retourner des données sans aucun sens, en silence.

Sommes-nous vraiment obligés de supporter cela ? La réponse est « non » !

## 6 - Inclure ONLY\_FULL\_GROUP\_BY dans le sql\_mode de MySQL

Désormais, MySQL est également capable de détecter ce problème, et il est parfaitement possible de faire rejeter la requête précédente par MySQL et ainsi éviter qu'il ne renvoie des données sans aucun sens. C'est faisable en incluant `ONLY_FULL_GROUP_BY` dans le `sql_mode`.

Comme la plupart des réglages de serveur, on peut préciser le `sql_mode` en passant l'argument de ligne de commande `--sql-mode` à l'exécutable du serveur MySQL (`mysqld`), ou on peut l'inclure dans un fichier d'options. Par exemple, inclure la ligne suivante dans le fichier d'options activera `ONLY_FULL_GROUP_BY` au démarrage du serveur :

```
sql_mode=ONLY_FULL_GROUP_BY
```

Depuis MySQL 4.1, il est également possible d'ajuster le `sql_mode` en cours d'exécution par le biais de la syntaxe `SET`. De cette façon, le `sql_mode` peut être ajusté globalement ou au niveau de la session. Ce dernier est le plus utile, parce qu'il permet à chacun d'ajuster au mieux le `sql_mode` pour une application sans affecter les autres applications qui tournent sur le serveur (certaines applications exigent les paramètres par défaut et peuvent ne plus fonctionner avec un `sql_mode` particulier).

Le fragment de code suivant montre comment inclure `ONLY_FULL_GROUP_BY` dans le `sql_mode` en cours d'exécution :

```
mysql> SET sql_mode := CONCAT('ONLY_FULL_GROUP_BY, ', @@sql_mode);
Query OK, 0 rows affected (0.00 sec)
```

La variable serveur `@@sql_mode` contient une chaîne de caractères (éventuellement vide) des paramètres du `sql_mode`, séparés par une virgule. La fonction `CONCAT` ajoute en début de `sql_mode` `ONLY_FULL_GROUP_BY`, quoi que la chaîne contienne déjà. Remarquez la virgule juste après `ONLY_FULL_GROUP_BY`. Si la valeur de la variable `@@sql_mode` est une chaîne vide, la valeur issue de l'expression `CONCAT` se terminera par une virgule, ce qui est autorisé (les virgules finales sont supprimées dans le processus).

Si on tente d'exécuter de nouveau notre requête, elle échoue avec un message d'erreur :

```
mysql> SELECT species
-> , MIN(birth) -- birthdate of oldest pet per species
-> , MAX(birth) -- birthdate of youngest pet per species
-> , birth -- birthdate of ... uh oh...!
-> FROM menagerie.pet
-> GROUP BY species;
ERROR 1055 (42000): 'menagerie.pet.birth' isn't in GROUP BY
```

Le message d'erreur indique qu'on n'a pas inclus la colonne `birth` dans la clause `GROUP BY`. Désormais, MySQL se comporte de la même manière qu'Oracle avec cette requête.

## 7 - Que dit « la » norme SQL ?

Dans les parties précédentes, nous avons vu qu'Oracle et MySQL réagissent de façon très différente à la clause **GROUP BY**. Mais que dit la norme ? Comment la clause **GROUP BY** est-elle supposée se comporter ?

En introduction de cet article, j'ai clamé que j'allais briser un mythe populaire qui prétend que... la norme SQL impose que les colonnes référencées dans la liste du **SELECT** apparaissent également dans la clause **GROUP BY**, à moins que ces colonnes apparaissent exclusivement dans une fonction d'agrégation.

Maintenant, je ne prétends pas être un expert de la norme SQL (ISO/IEC 9075). En fait, j'ai noté à plusieurs reprises que le volume de documentation, de même que le vocabulaire très soutenu ne me permettent pas d'avoir une vue claire de la question. Mais essayons tout de même.

Dans la version de 1992 de la norme, le paragraphe 7.9-7 stipule :

*"Si T est une table regroupée, alors toute <colonne référence> dans chaque <expression> qui référence une colonne de T doit référencer un groupement de colonnes ou être spécifié dans un <ensemble de fonctions de spécification>."*

Comme je le disais, je ne suis pas un spécialiste de la question, mais tel que je le lis, cela se résume à :

*"Les requêtes qui contiennent une clause GROUP BY peuvent inclure une colonne référencée dans le SELECT uniquement si la colonne apparaît dans la clause GROUP BY, ou si cette colonne apparaît dans une fonction d'agrégation."*

Désormais, au paragraphe 7.12-15 de la norme de 2003, on trouve ceci :

*"Soient T une table groupée et G l'ensemble des colonnes groupées sur T. Pour toute <expression> contenue dans la <liste du SELECT>, toute référence de colonne qui référence une colonne de T doit référencer une colonne C qui dépend fonctionnellement de G ou qui doit être contenue dans un argument agrégé d'un <ensemble de fonctions de spécification> dont la requête d'agrégation est QS."*

La version de la norme de 1999 contient une règle identique. Ce qu'il est important de retenir ici, c'est que les 2 versions qui succédèrent à la norme de 1992 ne requièrent plus explicitement que toutes les colonnes non agrégées dans la liste du **SELECT** soient présentes dans la clause **GROUP BY**. A la place, il faut que toute colonne non agrégée qui apparaît dans la liste du **SELECT** soit **fonctionnellement dépendante** de la clause **GROUP BY**.

## 8 - Dépendances fonctionnelles

Que peut bien vouloir dire « fonctionnellement dépendant » dans les normes de 1999 et 2003 ? La réponse à cette question est également définie dans la norme. Malheureusement, on ne peut pas l'illustrer par un exemple simple, parce que la définition formelle de ce que constitue exactement une dépendance fonctionnelle, selon la norme, est plutôt vaste et compliqué.

Par chance, le concept de dépendance fonctionnelle peut être illustré simplement de manière un peu moins formelle. Supposons que l'on ait deux expressions, **A** et **B**. **B** est dépendant fonctionnellement de **A** si **B** a une valeur et une seule pour une valeur particulière de **A**. Regardons l'extrait de code suivant :

```
mysql> SELECT @A:=1 AS A
        -> , @B:=@A + 1 AS B;
+----+-----+
| A | B |
+----+-----+
| 1 | 2 |
+----+-----+
```

Ici, la colonne **B** est dépendante fonctionnellement de la colonne **A**. La valeur de **B** peut être déduite de la valeur de **A** de manière très étroite, à savoir en ajoutant 1 à toute valeur de **A**. On peut calculer la valeur de **B** quelle que soit la valeur de **A**, et pour toute valeur de **A**, la valeur correspondante de **B** sera toujours la même.

La notion de dépendance fonctionnelle peut aussi s'appliquer à plusieurs colonnes :

```
SELECT CONCAT(A,B) C
FROM   uneTable
```

La colonne **C** est définie par le résultat de la fonction **CONCAT(A,B)**. Connaissant les valeurs de **A** et de **B**, on peut calculer le résultat de **CONCAT(A,B)**. Bien entendu, pour un couple donné de valeurs de **A** et **B**, le résultat de **CONCAT(A,B)** sera toujours le même. Ainsi, **C** est dépendant fonctionnellement du couple de colonnes **A** et **B**. Notez qu'il ne suffit pas simplement de connaître la fonction de calcul de **B** à partir de **A**. Prenons cet exemple :

```
mysql> SELECT @A:=1          AS A
-> ,      @B:=@A + RAND() AS B;
```

Ici, nous connaissons le moyen d'obtenir la valeur de **B** à partir de la valeur de **A** : on prend la valeur de **A** et on lui ajoute la valeur renvoyée par la fonction **RAND()**. Toutefois, la fonction **RAND()** retourne une valeur différente à chaque fois qu'on l'appelle. De ce fait, la valeur de **B** sera différente à chaque fois, et on peut donc en conclure que **B** n'est pas dépendant fonctionnellement de **A**. Ainsi, il n'y a pas une valeur unique de **B** pour une même valeur de **A**, donc **B** ne dépend pas fonctionnellement de **A**.

## 8-A - Dépendance fonctionnelle et normalisation

Le terme de dépendance fonctionnelle est également employé au regard de la normalisation. Une partie du processus de normalisation consiste à déterminer les dépendances fonctionnelles entre différents groupes de colonnes.

La normalisation nécessite que chaque table possède au moins une clé. Une clé est une colonne ou un groupe de colonnes utilisé pour identifier chaque enregistrement dans une table. Par définition, si on dispose d'une clé, toutes les colonnes non-clé sont dépendantes fonctionnellement de la clé. Une autre manière de considérer la question consiste à imaginer que l'on recherche une ligne en utilisant une clé. Si la valeur de la clé existe, on obtient une ligne et une seule. Par définition, toute colonne de cette ligne possède une valeur, et donc la valeur de chaque colonne non-clé peut être déduite de la clé.

Les dépendances fonctionnelles entre un groupe de colonnes composant la clé et n'importe quel autre groupe de colonnes sont autorisées, alors que les dépendances fonctionnelles entre deux groupes de colonnes qui ne font pas partie de la clé sont éliminées lors du processus de normalisation (ce qui est fait en séparant les groupes de colonnes qui mettent en avant la dépendance fonctionnelle dans une nouvelle table, en faisant de ces groupes de colonnes une clé de la nouvelle table).

## 8-B - Dépendances fonctionnelles et GROUP BY

Nous venons juste de voir que les versions 1999 et 2003 de la norme SQL imposent que les colonnes qui sont contenues dans la liste du **SELECT** soient fonctionnellement dépendantes du groupe défini dans la clause **GROUP BY**. En d'autres termes, si l'on sait qu'une colonne contient une et une seule valeur pour une combinaison donnée de valeurs des colonnes apparaissant dans la clause **GROUP BY**, on peut mettre cette colonne dans la liste du **SELECT** sans la faire apparaître dans une fonction d'agrégation.

Nous avons également vu que si on dispose d'une clé (primaire ou unique), toutes les colonnes qui ne sont pas incluses dans la clé sont fonctionnellement dépendantes de cette clé. Cela signifie que, si on inclut les colonnes clé dans la clause **GROUP BY**, on peut référencer n'importe quelle colonne dans la liste du **SELECT**, même si elles apparaissent en-dehors d'une fonction d'agrégation.

L'exemple suivant, tiré de la base de données exemple **sakila**, permet d'illustrer notre propos :

```
mysql> SELECT  film_id      -- clé primaire
-> ,          title        -- colonne non-clé
-> ,          COUNT(*)     -- une ligne par groupe
-> FROM      sakila.film
-> GROUP BY  film_id;     -- group by sur la clé primaire
```

```
+-----+-----+-----+
| film_id | title                | COUNT(*) |
+-----+-----+-----+
| 1 | ACADEMY DINOSAUR    | 1 |
+-----+-----+-----+
```

```

[...]
| 1000 | ZORRO ARK | 1 |
+-----+-----+
1000 rows in set (0.05 sec)

```

Ici, on requête la table `film`. La clé primaire de la table `film` est seulement composée de la colonne `film_id`. La clause `GROUP BY` contient la colonne `film_id`. En résultat, la requête retourne un ensemble de groupes, chacun d'eux résumant une seule ligne. Bien évidemment, puisqu'il n'y a qu'une seule ligne par groupe, il ne peut y avoir qu'une seule valeur dans chacune des autres colonnes de la table `film`. C'est pourquoi on peut se permettre d'inclure n'importe quelle colonne dans la liste du `SELECT` sans aucun risque. Pour cette raison, il est parfaitement juste d'inclure la colonne `film_title` dans la liste du `SELECT`.

Bien entendu, le `GROUP BY` dans la requête précédente n'a aucun sens d'un point de vue logique. Puisque la clé primaire de la table `film` n'est composée que de la colonne `film_id`, on sait déjà qu'il ne peut y avoir qu'une seule ligne pour une valeur donnée de `film_id`. Toutefois, cela devient intéressant quand on inclut d'autres tables dans la requête. Prenons l'exemple suivant :

```

mysql> SELECT      f.film_id
-> ,              f.title
-> ,              COUNT(fa.actor_id)
-> FROM          film      f
-> LEFT JOIN    film_actor fa
-> ON          f.film_id = fa.film_id
-> GROUP BY    f.film_id;
+-----+-----+-----+
| film_id | title           | COUNT(*) |
+-----+-----+-----+
| 1       | ACADEMY DINOSAUR | 10       |
.
.
[...on ne montre pas les 998 lignes...]
.
.
| 1000   | ZORRO ARK       | 3        |
+-----+-----+-----+
1000 rows in set (0.02 sec)

```

Ici, nous avons ajouté un `LEFT JOIN` pour calculer le nombre d'acteurs par film. Cette fois-ci, la clause `GROUP BY` sur la colonne `film_id` prend tout son sens : on obtient maintenant un groupe d'acteurs jouant un rôle dans chaque film. Dans le même temps, on sait que toutes les colonnes de la table `film` sont fonctionnellement dépendantes de la colonne `film_id`. Chaque groupe retourné par la clause `GROUP BY` correspond à une ligne et une seule de la table `film`, ce qui signifie que pour chaque groupe, il n'y a qu'une seule valeur dans chaque colonne de la table `film`. Donc, il n'y a aucun risque à référencer les colonnes de la table `film` dans la liste du `SELECT`, même si on ne les utilise pas dans une fonction d'agrégation.

Il est nécessaire de prendre conscience que l'on ne peut pas référencer n'importe quelle colonne dans la liste du `SELECT` : on ne peut référencer que les colonnes qui dépendent fonctionnellement de la colonne `film_id` de la table `film`. Cela veut dire qu'il est faux de référencer directement n'importe quelle colonne de la table `film_actor` dans la liste du `SELECT` : on doit alors passer par une fonction d'agrégation.

L'exemple précédent expose un motif. La table `film` agit comme un **maître** et la table `film_actor` se comporte comme un **détail**. Le motif maître-détail est très courant : commande et objets commandés, vendeur et produits, pays et villes sont tous des exemples de ce motif.

## 9 - Pourquoi donc voudrais-je faire cela ?

Je l'espère, j'ai été capable d'expliquer dans quelles circonstances il n'y a pas de risque, dans les requêtes avec `GROUP BY`, de référencer directement des colonnes dans la liste du `SELECT`. On pourrait pourtant se demander quels en sont les avantages et les inconvénients. Je veux dire, pouvoir le faire ne signifie pas qu'on doive le faire, non ?

## 9-A - Inconvénients ?

Eh bien, il y a certainement des raisons pour toujours écrire une clause **GROUP BY** complète.

Tout d'abord, la plupart des SGBDR n'autorisent qu'une clause **GROUP BY** complète, donc dans ce cas il n'y a pas réellement de choix. Cependant, dans MySQL, ce choix est possible tant qu'on n'utilise pas **ONLY\_FULL\_GROUP\_BY** dans le **sql\_mode**.

Une autre raison de toujours écrire la clause **GROUP BY** en entier vient du fait que d'autres développeurs pourraient ne pas comprendre les circonstances exactes de l'utilisation d'une clause **GROUP BY** partielle. Le plus souvent, ils ont passé beaucoup de temps à apprendre aveuglément à répéter toutes les colonnes de la liste du **SELECT** dans la clause **GROUP BY**, et ils signaleront généralement qu'il est faux de ne pas appliquer cette règle.

Bien sûr, il est impossible de faire la distinction entre une requête qui omet volontairement des colonnes dans la clause **GROUP BY**, et une autre dans laquelle on a oublié de les inclure. Quand l'intention est de toujours écrire une clause **GROUP BY** complète, il est facile de vérifier si les colonnes référencées dans la clause **GROUP BY** correspondent aux colonnes dans la liste du **SELECT**.

J'ai entendu certaines personnes argumenter que le fait d'inclure seulement la clé dans la clause **GROUP BY** peut provoquer des problèmes si on change la définition de la clé. Personnellement, je pense que c'est un argument bidon. Quand vous étudiez un changement de définition de clé, vous allez de toute manière devoir reconsidérer toutes vos requêtes, parce que toutes vos jointures vont devoir également refléter cette modification. Je veux juste dire que changer quelques **GROUP BY** par-ci, par-là est le cadet de vos soucis quand vous étudiez le changement de définition d'une clé. Un autre argument que j'ai entendu est qu'il est, d'une manière ou d'une autre, "plus clair", "plus propre" et "plus joli" de répéter toutes les colonnes référencées dans la liste du **SELECT** au niveau de la clause **GROUP BY**. Personnellement, je pense que c'est également un argument bidon. En fin de compte, il s'agit de questions de point de vue.

## 9-B - Avantages

Personnellement, je pense qu'il est plus clair et plus joli de regrouper uniquement par colonnes clé lorsque c'est possible. J'ai précisé que ce qui est "clair" ou "joli" est une question de point de vue, donc je dois également écarter cet argument.

J'affirmerais pour ma part que les clauses **GROUP BY** complètes sont plus difficiles à maintenir. Beaucoup de modifications vont requérir deux éditions du code au lieu d'une. Bien entendu, cela peut ou pas l'emporter sur les autres avantages d'un **GROUP BY** complet.

Une clause **GROUP BY** complète peut être plus lente qu'une clause partielle. La requête suivante ramène tous les films qui ont rapporté plus de 300 \$ :

```
mysql> SELECT      f.film_id
-> ,              f.title
-> ,              sum(p.amount) sum_amount
-> FROM          film f
-> LEFT JOIN     inventory i
-> ON           f.film_id = i.film_id
-> LEFT JOIN     rental r
-> ON           i.inventory_id = r.inventory_id
-> LEFT JOIN     payment p
-> ON           r.rental_id = p.rental_id
-> GROUP BY     f.film_id
-> HAVING       sum_amount > 300;
Empty set (0.18 sec)
```

En n'utilisant que la colonne **film\_id** dans la clause **GROUP BY**, découvrir qu'il n'y a aucun film satisfaisant ce critère ne prend que 0,18 secondes. Comparons-la à la requête équivalente utilisant une clause **GROUP BY** complète :

```
mysql> SELECT      f.film_id
-> ,              f.title
-> ,              sum(p.amount) sum_amount
-> FROM          film f
-> LEFT JOIN     inventory i
-> ON           f.film_id = i.film_id
```

```

-> LEFT JOIN rental r
-> ON      i.inventory_id = r.inventory_id
-> LEFT JOIN payment p
-> ON      r.rental_id = p.rental_id
-> GROUP BY f.film_id
-> ,      f.title
-> HAVING  sum_amount > 300;
Empty set (0.51 sec)

```

Cette requête a pris presque trois fois plus de temps. Avec **EXPLAIN**, on peut afficher les plans d'exécution de ces requêtes. Sans une clause **GROUP BY** complète, on voit un plan d'exécution relativement normal :

```

mysql> EXPLAIN
-> SELECT      f.film_id
-> ,          f.title
-> ,          sum(p.amount) sum_amount
-> FROM        film f
-> LEFT JOIN   inventory i
-> ON          f.film_id = i.film_id
-> LEFT JOIN   rental r
-> ON          i.inventory_id = r.inventory_id
-> LEFT JOIN   payment p
-> ON          r.rental_id = p.rental_id
-> GROUP BY    f.film_id
-> HAVING      sum_amount > 300
-> \G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: f
      type: index
possible_keys: NULL
      key: PRIMARY
      key_len: 2
      ref: NULL
      rows: 953
      Extra:
***** 2. row *****
      id: 1
select_type: SIMPLE
      table: i
      type: ref
possible_keys: idx_fk_film_id
      key: idx_fk_film_id
      key_len: 2
      ref: sakila.f.film_id
      rows: 2
      Extra: Using index
***** 3. row *****
      id: 1
select_type: SIMPLE
      table: r
      type: ref
possible_keys: idx_fk_inventory_id
      key: idx_fk_inventory_id
      key_len: 3
      ref: sakila.i.inventory_id
      rows: 1
      Extra: Using index
***** 4. row *****
      id: 1
select_type: SIMPLE
      table: p
      type: ref
possible_keys: fk_payment_rental
      key: fk_payment_rental
      key_len: 5
      ref: sakila.r.rental_id
      rows: 1
      Extra:

```

```
4 rows in set (0.00 sec)
```

Avec un **GROUP BY** complet, on observe une différence sur la table **film** :

```
mysql> EXPLAIN
-> SELECT      f.film_id
-> ,
->             sum(p.amount) sum_amount
-> FROM        film f
-> LEFT JOIN   inventory i
-> ON          f.film_id = i.film_id
-> LEFT JOIN   rental r
-> ON          i.inventory_id = r.inventory_id
-> LEFT JOIN   payment p
-> ON          r.rental_id = p.rental_id
-> GROUP BY   f.film_id
-> ,
->             f.title
-> HAVING      sum_amount > 300
-> \G
***** 1. row *****
      id: 1
      select_type: SIMPLE
      table: f
      type: index
      possible_keys: NULL
      key: idx_title
      key_len: 767
      ref: NULL
      rows: 953
      Extra: Using index; Using temporary; Using filesort
***** 2. row *****
      id: 1
      select_type: SIMPLE
      table: i
      type: ref
      possible_keys: idx_fk_film_id
      key: idx_fk_film_id
      key_len: 2
      ref: sakila.f.film_id
      rows: 2
      Extra: Using index
***** 3. row *****
      id: 1
      select_type: SIMPLE
      table: r
      type: ref
      possible_keys: idx_fk_inventory_id
      key: idx_fk_inventory_id
      key_len: 3
      ref: sakila.i.inventory_id
      rows: 1
      Extra: Using index
***** 4. row *****
      id: 1
      select_type: SIMPLE
      table: p
      type: ref
      possible_keys: fk_payment_rental
      key: fk_payment_rental
      key_len: 5
      ref: sakila.r.rental_id
      rows: 1
      Extra:
4 rows in set (0.01 sec)
```

Au cas où vous ne l'auriez pas remarqué, la table **film** a à présent : **Extra: Using index; Using temporary; Using filesort** . Ce que je pense qu'il se passe, c'est que MySQL prend la clause **GROUP BY** mot à mot et réalise l'algorithme **GROUP BY** pour chaque expression spécifiée. MySQL implémente le **GROUP BY** en triant les lignes selon les expressions du **GROUP BY**. Dans le cas présent, l'ajout de la colonne **title** à la clause **GROUP BY** ne permet pas au serveur de

trier les lignes en mémoire, et force l'évaluation du **GROUP BY** en utilisant une table temporaire et un tri sur fichier. Il pourrait essayer de détecter qu'une clé de la table **film** est incluse dans la liste du **GROUP BY** et que la colonne titre peut être complètement ignorée lors de l'évaluation de la clause **GROUP BY**, puisqu'il y aura exactement une valeur dans la colonne **title** pour chaque **film\_id**. Mais, une fois encore, MySQL n'impose pas qu'on écrive un **GROUP BY** complet. Ainsi, si les performances sont prépondérantes, soyez astucieux et n'écrivez pas une clause **GROUP BY** complète.

## 9-C - Agrégation sur des colonnes fonctionnellement dépendantes

Nous venons juste de voir qu'il n'y a pas de risque d'inclure des colonnes dans la liste du **SELECT** tant que ces colonnes sont fonctionnellement dépendantes de la liste du **GROUP BY**. La raison en est que, tant que les colonnes dépendent fonctionnellement de la clause **GROUP BY**, il n'y aura qu'une seule valeur pour chaque résultat du groupe. Nous avons également montré que ça pouvait être une mauvaise idée d'inclure ces colonnes dans la clause **GROUP BY**, pour des questions de dégradation des performances.

Pour certains, il peut toujours être inacceptable d'inclure des colonnes fonctionnellement dépendantes dans la liste du **SELECT** sans les référencer dans la clause **GROUP BY**. Par exemple, vous pourriez utiliser un SGBD qui nécessite que toutes les colonnes qui ne sont pas référencées dans la clause **GROUP BY** soient agrégées. Dans ce cas, il est préférable d'appliquer une fonction d'agrégation aux colonnes fonctionnellement dépendantes plutôt que de les inclure dans la clause **GROUP BY**. Considérons la requête suivante :

```
mysql> SELECT      f.film_id
-> ,              MAX(f.title) AS title
-> ,              sum(p.amount) sum_amount
-> FROM          film f
-> LEFT JOIN     inventory i
-> ON            f.film_id = i.film_id
-> LEFT JOIN     rental r
-> ON            i.inventory_id = r.inventory_id
-> LEFT JOIN     payment p
-> ON            r.rental_id = p.rental_id
-> GROUP BY     f.film_id
-> HAVING        sum_amount > 300;

Empty set (0.20 sec)
```

Une fois de plus, on regroupe sur la colonne **film\_id**, qui représente la clé primaire de la table **film**. Cette fois-ci, cependant, on applique la fonction **MAX** à la colonne **title**. Nous savons qu'il n'y a qu'une seule valeur de la colonne **title** pour chaque valeur de la colonne **film\_id**, donc la fonction d'agrégation n'influencera pas le résultat. En réalité, on aurait tout aussi bien pu utiliser la fonction **MIN**.

Ces fonctions d'agrégation retourneront la même valeur correcte pour la même raison qu'il n'y a pas de risque de ne pas inclure une colonne fonctionnellement dépendante dans la clause **GROUP BY**. Techniquement, l'agrégation n'est pourtant pas nécessaire : c'est juste une astuce pour tromper le SGBD.

De toute évidence, appliquer une fonction d'agrégation sera plus lent que de ne pas le faire. Toutefois, ce sera souvent plus rapide que d'inclure la colonne fonctionnellement dépendante dans la clause **GROUP BY**.

## Conclusion

Contrairement à une croyance populaire, la norme SQL n'impose pas que les requêtes avec **GROUP BY** référencent toutes les colonnes non agrégées de la liste du **SELECT** dans la clause **GROUP BY**. A l'instar de la version de 1999 de la norme SQL, il est explicitement permis que la liste du **SELECT** référence des expressions non agrégées, du moment qu'elles dépendent fonctionnellement de la liste du **GROUP BY**.

Toute expression qui n'a qu'une seule valeur pour chaque groupe défini dans la clause **GROUP BY** dépend fonctionnellement de cette clause. On peut percevoir les dépendances fonctionnelles selon un motif de requête courant : chaque fois qu'on a une jointure entre une table maître et une table détail pour calculer des agrégats sur les lignes de détail selon chaque ligne du maître, on peut utiliser la clause **GROUP BY** sur la clé primaire ou unique du maître. Toute colonne non clé d'une ligne du maître sera dépendante fonctionnellement de la clé, et peut ainsi figurer dans la liste du **SELECT**, en dehors d'une fonction d'agrégation.

Dans MySQL, il est possible d'écrire des requêtes avec **GROUP BY** qui référencent des colonnes dans la liste du **SELECT** non incluses dans la clause du **GROUP BY**, même si ces colonnes ne dépendent pas fonctionnellement de la clause **GROUP BY**. Ce comportement n'est conforme à aucune des versions de la norme SQL. Il est possible de l'éviter en incluant **ONLY\_FULL\_GROUP\_BY** dans le paramètre **sql\_mode** du serveur, mais il peut être plus judicieux de tirer parti de cette possibilité pour écrire des clauses **GROUP BY** partielles.

En résumé :

- Il n'y a absolument aucun risque avec des clauses **GROUP BY** partielles, du moment que les colonnes non agrégées dans la liste du **SELECT** dépendent fonctionnellement de la clause **GROUP BY**.
- Un **GROUP BY** partiel peut apporter de meilleures performances, puisqu'il évite au serveur l'évaluation de toute la liste du **GROUP BY**.
- Si on ne souhaite pas écrire une clause **GROUP BY** partielle, il est préférable d'utiliser **MIN** ou **MAX** pour agréger les colonnes fonctionnellement dépendantes dans la liste du **SELECT** plutôt que de déplacer ces colonnes dans la clause **GROUP BY**.