Rédigé par : Jimmy Paquereau

Fiche de révisions - SQL

1. Les clauses SQL

Les mots-clefs **SELECT**, **FROM**, **WHERE**, **GROUP BY**, **HAVING** et **ORDER BY** sont appelés des **clauses**.

Important ! Dans tout ce qui suit, n'oubliez pas de préfixer les champs par le nom de la table (exemple : NomTable.NomChamp) lorsqu'il y a ambiguïté, à savoir lorsque votre requête conduit à la présence de deux champs portant le même nom.

SELECT *
Permet d'afficher tous les champs disponibles de toutes les tables « sélectionnées »

SELECT champ1, champ2, …,champN
SELECT champ1 AS Alias1, champ2 AS Alias2, …,champN AS AliasN
Permet d'afficher 1 ou plusieurs champs parmi les champs disponibles.

FROM UneTable1, UneTable2, …, UneTableN
FROM UneTable1 AS Alias1, UneTable2 AS Alias2, …, UneTableN AS AliasN
Permet de préciser les tables à utiliser.

WHERE Conditions
Permet de préciser les lignes à conserver ou retirées de la « sélection » (de la projection). Les conditions sont des expressions booléennes (voir fiche sur l'algorithmique) portant sur les champs (les colonnes) des lignes. On parle de **restriction**.

GROUP BY UneTable.champ1, UneTable.champ2, UneTable.champN
Permet de regrouper des lignes les unes avec les autres.
1ère remarque : conséquence, il est impossible de regrouper deux lignes selon une colonne *champX* si ces deux lignes n'ont pas la même valeur dans la colonne *champX* (à méditer !).
2ème remarque : GROUP BY s'utilise avec des **agrégats**, i.e. des fonctions effectuant un calcul sur chaque groupe de lignes : COUNT(*), COUNT(unChamp), COUNT(DISTINCT unCham), AVG(unChamp), SUM(unCham). Par exemple, COUNT(*) , pour chaque groupe, le nombre de lignes regroupées.
3ème remarque : en règle générale, les champs (hors agrégats) figurant dans la clause SELECT doivent figurer dans la clause GROUP BY.

GROUP BY UneTable.champ1, UneTable.champ2, UneTable.champN
Permet de regrouper des lignes les unes avec les autres.
1ère remarque : conséquence, il est impossible de regrouper deux lignes selon une colonne *champX* si ces deux lignes n'ont pas la même valeur dans la colonne *champX* (à méditer !).
2ème remarque : GROUP BY s'utilise avec des **agrégats**, i.e. des fonctions effectuant un calcul sur chaque groupe de lignes : **COUNT(*)**, **COUNT(unChamp)**, **COUNT(DISTINCT unCham)**, **AVG(unChamp)**, **SUM(unChamp)**, **MIN(unChamp)**, **MAX(unChamp)**. Par exemple, COUNT(*) , pour chaque groupe, le nombre de lignes regroupées.
3ème remarque : en règle générale, les champs (hors agrégats) figurant dans la clause SELECT doivent figurer dans la clause GROUP BY.
4ème remarque : une fois la clause GROUP BY exécutée, à un regroupement correspond une unique ligne. Autrement dit, gars à ne pas mettre n'importe quoi dans la clause HAVING. Les champs ne figurant pas dans la clause SELECT ne peuvent en règle générale plus être utilisés tels quels.

HAVING conditions
Permet d'effectuer une restriction à la manière d'un WHERE, mais une fois la clause GROUP BY exécutée.
1ère remarque : les conditions figurant dans la clause HAVING portent en générales sur les agrégats. N'hésitez pas à utiliser l'alias précisé dans la clause SELECT (si vous en avez précisé un bien sûr).
Exemple : COUNT(*) > 5.
2ème remarque : il est absolument hors de question de voir apparaître des conditions comportant des agrégats dans la clause WHERE ! En effet, c'est impossible, lorsque le WHERE est exécuté, les agrégats n'ont pas encore été calculés, la clause GROUP BY s'exécutant après la clause WHERE.

ORDER BY champ1, champ2 ASC, champ3 DESC, …
Permet de trier les lignes (résultat de la requête) en fonction des champs. Le ASC (*ascendant*) permet de trier par ordre croissant, DESC (*descendant*) par ordre décroissant. Ne rien mettre équivaut à trier par ordre croissant (ASC).

LIMIT nombreLignes OFFSET premiereLigne
Permet de ne retourner qu'une partie des lignes, à savoir permet de retourner « nombreLignes » lignes à partir de la ligne « premiereLigne ». Attention ! La première ligne est la ligne 0.

2. Opérateurs et fonctions

Les opérateurs et fonctions sont volontiers utilisés dans la clause WHERE (mais pas nécessairement).

Les opérateurs logiques (attention au priorité de calcul, voir fiche algorithmiques) :
Condition1 **AND** Condition2
Condition1 **OR** Condition2
NOT Condition1

Les opérateurs de comparaison (non exhaustif) :
Champ1 = Champ2
Champ1 <> Champ2 ← Notation SQL de ≠ (différent de)
Champ1 > Champ2 et ← Respectivement <
Champ1 >= Champ2 ← Respectivement <=
Champ1 **IS NULL** et Champ **IS NOT NULL** ← Teste si la valeur du champ est nulle (resp. non nulle)
Champ1 **LIKE** ''CH%'' Champ1 **LIKE** ''%CHE'' ← Teste si la valeur du champ commence par ''CH'' (resp. se termine par ''CHE'')
Champ1 **BETWEEN** … **AND** …
Champ1 **BETWEEN** #JJ/MM/AAAA# AND #JJ/MM/AAAA# ← Si Champ1 est une date
Champ1 **IN** (''Valeur1'', ''Valeur2'', …) ← Teste si Champ1 est égal à l'une des valeurs dans la liste
Champ1 **IN** (**SELECT** … **FROM** … **WHERE** …) ← Idem, mais la liste est retournée par une requête

Les fonctions (non exhaustif, propre à chaque SGBD) :
YEAR(Champ1) ← Retourne l'année correspondant à la date Champ1
MONTH(Champ1) ← Retourne l'année correspondant à la date Champ1
DAY(Champ1), MINUTE(Champ1), HOUR(Champ1), SECOND(Champ1)
NOW ou NOW() ← Retourne la date courante (date actuelle, aujourd'hui)

Les fonctions SQL, multiples en pratique (voire, on peut en créer), peuvent très bien être utilisées dans la clause WHERE avec les opérateurs de comparaison usuel =, >=, BETWEEN…
Exemple : YEAR(Champ1) BETWEEN 2010 AND 2016).

3. Sous-requêtes

Rien de très original, une sous-requête est une requête dans une requête. Dès que l'on utilise des sous-requêtes, il est vivement conseillé d'utiliser des ALIAS dès que requête et sous-requêtes utilisent les mêmes tables, voire s'utilisent mutuellement.

Dans nombre de cas, on peut éviter les sous-requêtes. On les utilise essentiellement pour récupérer des statistiques.

Exemple : je veux récupérer la moyenne générale, au semestre 1 de cette année, des élèves de la classe n°7. Il me faut calculer la moyenne générale de chaque élève (sous-requête) puis calculer la moyenne de ces moyennes (à méditer). Cela pourrait donner une requête s'apparentant à la suivante :

```
SELECT AVG(Moyenne)
FROM (
        SELECT AVG(Note * Coefficient) AS Moyenne
        FROM Notes, Etudiant
        WHERE Notes.NumEtudiant = Etudiant.NumEtudiant
        AND Notes.Semestre = 1
        AND Notes.Year = YEAR(NOW())        ← Allez, Un brin de folie !
        AND Etudiant.NumClasse = 7
        GROUP BY Notes.NumEtudiant          ← Remarquez l'obligation de préfixer par Notes.
)
```

Autre exemple : on veut connaître l'élève ou les élèves (s'il y a des *exæquos*) qui a ou ont eu la meilleure note de l'année 2016. Ici, l'idée, c'est d'une part de chercher la meilleure note de l'année 2016, d'autre part de chercher celui ou ceux qui l'a ou l'ont eue.

```
SELECT DISTINCT Etudiant.Prenom, Etudiant.Nom   ← Remarquez la nécessité d'ajouter DISTINCT
FROM Notes AS N1, Etudiant                        afin d'éviter un doublon (duplicata, i.e. 2x le même
WHERE N1.NumEtudiant = Etudiant.NumEtudiant       étudiant*)
AND N1.Year = 2016
AND N1.Note = (
        SELECT Max(N2.Note)
        FROM Notes As N2                        ← L'alias lève ici tout ambiguïté (dans le doute…)
        WHERE N2.Year = 2016
)
```

** Il peut y avoir deux fois le même étudiant dans le cas où la meilleure note de l'année est par exemple 18 et où un même étudiant a obtenu deux fois 18 au cours de l'année.*

4. Jointures

On a vu des exemples ci-dessus. Il s'agit de jointures dites SQL1, à savoir des « pseudo-jointures ». Ces « pseudo-jointures » procèdent comme suit :
- en résumé, il s'agit d'un produit cartésien suivi d'une restriction. Explicitons.
- un produit cartésien consiste à mettre en tête-à-tête tous les éléments d'un ensemble (les lignes d'une table) avec tous les éléments d'un ensemble (les lignes de la même table ou d'une autre table).
- la restriction évoquée consiste à ne conserver que les lignes qui vont bien ensemble, à savoir celles pour lesquelles **Table1.clefEtrangere = Table2.clefPrimaire**.

Illustration : on se donne deux tables, Product(Num, Label, #Category) et Category(Num, Label)
Remarquez qu'on a appelé Category la clef étrangère de Product pointant sur Category.Num !

Product		
Num	Label	Category
1	Produit 1	1
2	Produit 2	1
3	Produit 3	2

Category	
Num	Label
1	Categorie 1
2	Categorie 2

Produit cartésien de Product et Category (c'est-à-dire FROM Product, Category) :

Product.Num	Product.Label	Product.Category	Category.Num	Category.Label
1	Produit 1	1	1	Categorie 1
1	Produit 1	1	2	Categorie 2
2	Produit 2	1	1	Categorie 1
2	Produit 2	1	2	Categorie 2
3	Produit 3	2	1	Categorie 1
3	Produit 3	2	2	Categorie 2

Nombre de lignes = Nombre de « Product » x Nombre de « Category »
Ce qu'on note mathématiquement : Card(Product x Category) = Card(Product) x Card(Category)

Finalement, après restriction (WHERE Product.Category = Category.Num), on obtient le résultat d'une jointure, à savoir qu'il ne reste que les lignes ou l'égalité susvisée est vérifiée (lignes ayant deux cases sévèrement encadrées ci-dessus et ci-dessous) :

Product.Num	Product.Label	Product.Category	Category.Num	Category.Label
1	Produit 1	1	1	Categorie 1
2	Produit 2	1	1	Categorie 1
3	Produit 3	2	2	Categorie 2

5. Opérateurs ensemblistes

Les opérateurs ensemblistes sont : UNION, EXCEPT et INTERSECT. Il permette de « mettre en relation » plusieurs requêtes. On retiendra que : **les deux requêtes doivent retourner un résultat de même nature : même nombre de colonnes et même type de colonnes !**

1	Produit 1	**UNION**	3	Produit 3	=	1	Produit 1
2	Produit 2		4	Produit 4		2	Produit 2
						3	Produit 3
						4	Produit 4

1	Produit 1	**EXCEPT**	1	Produit 1	=	2	Produit 2
2	Produit 2		3	Produit 3		4	Produit 4
3	Produit 3						
4	Produit 4						

1	Produit 1	**INTERSECT**	1	Produit 1	=	1	Produit 1
2	Produit 2		3	Produit 3		3	Produit 3
3	Produit 3						
4	Produit 4						

6. Requêtes paramétrées

Les requêtes paramétrées, qu'est-ce ? Tout d'abord, c'est du « made in Microsoft Access », puisque ça n'existe nulle part ailleurs. Cela consiste à poser une question à un utilisateur exécutant une requête Access, plus exactement, cela revient à lui demander de saisir une valeur et à utiliser la valeur qu'il a saisie, typiquement pour effectuer une comparaison.

Exemple : on veut afficher une fiche client (sous-entendu, les données d'un Client, contenue dans la table Client, et dont l'utilisateur nous précisera le numéro).

SELECT *
FROM Client
WHERE Client.NumClient = [Saisir un n° client :]

Ici, la « question » posée est « Saisir un n° client ». A l'exécution de la requête, une boîte de dialogue (petite fenêtre) apparaît. Celle-ci contient le message « Saisir un n° client ». L'utilisateur saisit un numéro X quelconque. Notre condition devient alors Client.NumClient = X (avec X la valeur saisie).

7. Quelques conseils

- Gardez bien en tête l'ordre dans lequel s'exécute une requête :
FROM > WHERE > GROUP BY > HAVING > SELECT

- Gardez bien en tête l'ordre dans lequel s'écrit une requête :
SELECT … FROM … WHERE … GROUP BY … HAVING … ORDER BY …

- A chaque fois, afin de rédiger une « requête SELECT » procédez comme suit :
a : je me demande où sont mes informations. Autrement dit, je me demande quelles sont les tables dont j'ai besoin. Je commence à rédiger ma clause FROM ;
b : je me demande comment relier mes tables entre elles (jointures). J'ajoute, le cas échéant, les tables manquantes à la clause FROM ;
c : j'ajoute immédiatement les jointures sans réfléchir (jointures ou « pseudo-jointure »), à savoir table1.clefPrimaire = table2.clefEtrangere AND …
d : j'ajoute les restriction manquantes, à savoir, je me demande : dois-je éliminer des lignes ?
e : ça y est ! Je peux m'occuper du reste.

- Bien vérifier que la question, le problème posé, ne puissent pas se décomposer en plusieurs problèmes. Si tel est le cas, penser à rédiger une requête par problème et, le cas échéant, utiliser une sous-requête.

- Si l'on cherche « le nombre de » ou « combien y a-t-il … », il y a du « SELECT COUNT(…) » dans l'air.

- Si l'on cherche « la moyenne de » ou quelque chose « en moyenne », il y a du « SELECT AVG(…) » dans l'air.

8. Requêtes INSERT INTO, UPDATE, DELETE FROM et autres

CREATE TABLE : ce sont les requêtes permettant la création de tables.
ALTER TABLE : ce sont les requêtes permettant la modification de tables existantes.
CREATE VIEW : ce sont des requêtes permettant de créer des vues, à savoir des requêtes se comportant comme des tables.
CREATE FUNCTION : sans doute les plus intéressantes, ce sont les « requêtes » permettant de créer des procédures stockées et de dépasser le cadre du SQL (Transact-SQL et PL-SQL).
CREATE TRIGGER : sans doute les plus intéressantes avec CREATE FUNCTION, ce sont les « requêtes » permettant de veiller au respect de contrainte d'intégrité complexes telles que **les contraintes d'associations**.
GRANT : requêtes permettant de conférer des droits aux utilisateurs d'une base de données.
CREATE INDEX, CREATE CONSTRAINT, DROP TABLE….

INSERT INTO UneTable **VALUES** ('Une chaîne', 1.555, #01/12/2015, 3, …) ;
INSERT INTO UneTable (champ1, champ2, …, champN) **VALUES** (valeur1, valeur2, …, valeurN) ;
INSERT INTO UneTable (champ1, champ2, …, champN) **VALUES** (valeur1, valeur2, …, valeurN), (valeur1, valeur2, …, valeurN), …, (valeur1, valeur2, …, valeurN) ;

Requête permettant d'insérer une ou plusieurs lignes dans une table. Bien noter que, si vous ne précisez pas les champs (champ1, champ2, …), vous êtes sensés les utiliser tous et dans le bon ordre ! Il vaut parfois mieux écrire un peu… Conseil : dans le doute, écrivez-les.

UPDATE UneTable **SET** UnChamp = UneValeur **WHERE** Conditions	**UPDATE** UneTable **SET** UnChamp1 = UneValeur1, UnChamp2 = UneValeur2 **WHERE** Conditions	**UPDATE** UneTable **SET** UnChamp1 = UnChamp1 * 2, **WHERE** Conditions
Requête permettant de modifier une ou plusieurs colonnes d'une ou plusieurs lignes d'une table. Bien noter qu'on peut : - modifier plusieurs colonnes d'un seul tenant ; - modifier un champ en fonction de lui-même ou encore d'un autre.		

DELETE FROM UneTable **WHERE** Conditions	**DELETE** UneTable1.* **FROM** UneTable1, UneTable2 **WHERE** jointure **AND** conditions
Requête permettant de supprimer une ou plusieurs lignes. Bien retenir que, même si, beaucoup de SGBD ne prennent pas en compte ce genre de DELETE, et même si quasiment personne ne fait des DELETE comme ça en pratique, vous devez savoir faire un de DELETE comme celui ci-dessus, à droite.	

9. Culturel

SGBD (Système de Gestion de Base de Données) : un SGBD est un logiciel permettant d'assurer le stockage et l'interrogation de données. Le SQL est le langage d'interrogation de données utilisé en pratique pour interroger des SGBDR (Système de Gestion de Base de Données Relationnelles). On notera que, dans les cas simples, un SGBD peut être localisé sur un serveur unique. Il y a des cas plus complexes :
- un SGBD peut être réparti sur plusieurs serveurs ;
- un SGBD peut être répliqué (base de données dupliquées de manière non triviale sur plusieurs serveurs) ;
- etc.

Les cas évoqués ci-dessus induisent de nombreuses problématiques techniques, dont certaines font à ce jour encore l'objet de travaux de recherche.
On notera également qu'il existe un autre type de SGBD très en vogue, en particulier lorsque le nombre d'utilisateurs devient grand (des milliers, voire des millions). Ces SGBD sont les SGBD nosql. Comme leur nom l'indique, ces SGBD ne s'interrogent pas en SQL.

BASES DE DONNEES ET MATHEMATIQUE : tables, SQL, opérateurs ensemblistes… Derrière nombre de concept que nous avons étudiés se cachent des concepts plus génériques, plus théoriques. Tous ces concepts dérivent entre autre de branches mathématiques : la théorie des ensembles et l'algèbre relationnel (d'où le terme de « base de données relationnelle »). Les règles à respecter pour avoir des bases de données relationnelles « fiables » ont également été théorisées au travers en outre des formes normales de Boyce-Codd.

SQL (Structured Query Language) : le SQL n'est pas à proprement parler un langage de programmation. On le classe parmi les **L4G** (langages de 4ème génération), à savoir des langages de programmation à usage spécifique, en l'occurrence un langage d'interrogation de données.

NATURAL JOIN : vraie jointure (SQL2), elle produit le même résultat que celle vous connaissez. Elle est essentiellement bien plus rapide et sa syntaxe épurée (Table1 NATURAL JOIN Table2, c'est tout) ! Ne croyez surtout pas que la puissance actuelle des ordinateurs nous permette de tout le temps d'utiliser des requêtes telles que celles que vous connaissez (avec WHERE). Sur des gros volumes de données, cela peut être prohibitif/rédhibitoire. Certaines requêtes, dont le résultat est produit en quelques secondes avec NATURAL JOIN, mettraient des heures voire des années à être exécutées avec une « pseudo-jointure ».

INNER JOIN : vraie jointure (SQL2), mais non naturelle, elle permet entre autre de produire le même résultat que les jointures avec WHERE. Sa syntaxe est plus lourde que celle du NATURAL JOIN et moins intuitive que celle des jointures avec WHERE. Il est plus lent que le NATURAL JOIN mais nettement plus rapide qu'une jointure avec WHERE.