Université de Montréal, EBSI
SCI6306 – Bases de données documentaires
Automne 2018

Documentation
SQL & MySQL

TABLE DES MATIÈRES

STRUCTURES GÉNÉRIQUES DES PRINCIPALES REQUÊTES SQL (SELECT, DELETE, UPDATE, INSERT INTO)

Structure générique d'une requête Sélection en SQL

❶ SELECT *nom(s) de champ* ❷ FROM *nom(s) de table* ❸ WHERE *condition(s)*
❹ GROUP BY *nom(s) de champ* ❺ ORDER BY *nom(s) de champ*;

```
SELECT UCASE(nom), COUNT(suit.no_etud) AS "Nbre de cours suivis" FROM etud, suit
WHERE etud.no_etud=suit.no_etud GROUP BY nom, suit.no_etud ORDER BY COUNT(suit.no_etud) DESC;
```

❶ **Instruction SELECT** [*affichage*]
- *Obligatoire*
- Précise le ou les champs à présenter dans la table des résultats
  - Pour sélectionner tous les champs, mettre * à la place des noms de champ : `SELECT * FROM …`
  - Champs séparés par des virgules : `SELECT no_etud, nom`
  - Si un même nom de champ est utilisé dans deux tables, faire précéder le nom du champ par le nom de la table en les séparant par un point (par ex. **`etud.no_etud, suit.no_etud`**)
  - Si un nom de champ et/ou de table comporte une espace ou un caractère spécial, l'encadrer par des apostrophes ouvrantes (par ex. **`'no etud'`**)
  - Possibilité de remplacer dans la présentation des résultats un nom de champ par un *alias* plus significatif : `SELECT nom AS "Nom de l'étudiant"`
  - Des fonctions peuvent être appliquées sur les champs (par ex. fonctions statistiques comme **`AVG`** pour la moyenne)
- Prédicat possible pour éliminer des doublons des résultats
  - Prédicat par défaut (implicite) : ALL – les doublons sont conservés (`SELECT ALL nom(s) de champ`)
  - `SELECT DISTINCT nom(s) de champ` : pour éliminer les doublons dans les lignes de résultats
  - `SELECT DISTINCTROW nom(s) de champ` : pour éliminer les doublons dans les lignes de la table de données

❷ **Clause FROM** *[source des données]*
- *Obligatoire*
- Précise la ou les tables d'où proviennent les champs
  - Noms de table séparés par des virgules : `FROM cours, suit`
  - Des alias peuvent être utilisés lorsqu'une table est utilisée plus d'une fois dans une clause FROM : `FROM cours AS c1, cours AS c2`

❸ **Clause WHERE** *[conditions]*
- *Facultative*
- Précise la (les) condition(s) pour retenir des enregistrements
  - Permet entre autres de faire les jointures entre des tables (par ex. `SELECT no_etud, no_cours FROM cours, suit WHERE cours.no_cours=suit.no_cours;`)
- Les lignes retenues sont celles pour lesquelles la condition est vraie
- Principaux types de condition
  - Égalité par ex. `WHERE etud.no_etud = suit.no_etud`
  - Inégalité par ex. `WHERE nom < 'L'`
  - Différence par ex. `WHERE no_prof <> ''`
  - Caractères génériques par ex. `WHERE nom LIKE 'Asi%'`
- Plusieurs conditions peuvent être reliées par des opérateurs booléens (AND, OR) (par ex. `WHERE cours.no_cours=suit.no_cours AND suit.no_etud = '10003'`). L'opérateur NOT permet d'obtenir la négation d'une condition (par ex. `WHERE local NOT LIKE 'C%'`).

❹ **Clause GROUP BY** *[regroupement]*
- *Facultative*
- Précise le ou les champs servant à regrouper les données
- Le regroupement des lignes par rapport à un ou des champs permet d'appliquer des fonctions d'agrégation sur ce (ces) champ(s) telles que **`COUNT`**, **`AVG`**, **`MIN`**, **`MAX`** : `SELECT COUNT(no_etud) FROM etud`

❺ **Clause ORDER BY** *[tri]*
- *Facultative*
- Précise le ou les champs servant de clé(s) de tri des résultats; si plus d'une clé de tri, les séparer par une virgule : `ORDER BY note, nom`
- Par défaut, le tri est ascendant; pour un tri descendant, ajouter `DESC` après le nom de champ : `ORDER BY note DESC`

Structure générique d'une requête Suppression en SQL

❶ DELETE FROM *nom de la table* ❷ WHERE *critère(s)*;

```
DELETE FROM etud WHERE nom='Asimov, Isaac';
```

❶ **Instruction DELETE FROM** *[source des données]*
- *Obligatoire*
- Précise la table où seront supprimés des enregistrements
- *Note* : les enregistrements supprimés le sont **définitivement**

❷ **Clause WHERE** *[conditions]*
- *Obligatoire*
- Précise le ou les critères pour choisir les enregistrements à effacer

Structure générique d'une requête Mise à jour en SQL

❶ UPDATE *nom de la table* ❷ SET *valeur(s)* ❸ WHERE *critère(s);*

```
UPDATE suit SET note = note + 2 WHERE note_p;
```

❶ **Instruction UPDATE** *[source des données]*
- *Obligatoire*
- Précise la table où des modifications seront apportées

❷ **Clause SET** *[valeurs]*
- *Obligatoire*
- Précise la ou les nouvelles valeurs des champs
  - Valeurs séparées par une virgule si plusieurs champs modifiés

❸ **Clause WHERE** *[conditions]*
- *Facultative*
- Précise le ou les critères sous lequel(lesquels) des lignes sont retenues pour être modifiées

Structure générique d'une requête Ajout en SQL

❶ INSERT INTO *nom de la table* (*nom(s) de champ)* ❷ VALUES (*valeur(s)*);

```
INSERT INTO etud (no_etud, nom) VALUES ('10009', 'Bauer, Jack');
```

❶ **Instruction INSERT INTO** *[source des données]*
- *Obligatoire*
- Précise le nom de la table où un nouvel enregistrement sera créé
- Précise entre parenthèses le nom du ou des champs pour lequel(lesquels) des valeurs seront précisées
  - Champs séparés par des virgules

❷ **Clause VALUES** *[valeurs]*
- *Obligatoire*
- Précise entre parenthèses la valeur du ou des champs du nouvel enregistrement
  - Valeurs séparées par des virgules
  - Chaînes textuelles entre apostrophe

EXEMPLES DE JOINTURES

Supposons que les tables PERSONNE (table des personnes) et LOCALITE (table faisant la correspondance entre les codes postaux et les localités associées) soient structurées comme suit[1] :

Table PERSONNE

Champ	Type	Taille
NO	Numérique	2
NOM	Caractère	30
TEL	Caractère	12
CP	Caractère	7

Table LOCALITE

Champ	Type	Taille
CP	Caractère	7
LOCALITE	Caractère	20

et qu'elles aient, à un moment donné, les contenus suivants :

PERSONNE

NO	NOM	TEL	CP
1	Bissonnette, Lise	514-432-3514	H2J 1C4
2	Valjean, Jean	514-123-2292	H3V 2E5
3	Smith, Luc	514-778-7167	H3V 2E5

LOCALITE

CP	LOCALITE
H2J 1C4	Ville St-Laurent
H2J 3A8	Ville St-Laurent
H3V 2E5	Montréal

Supposons que l'on veuille obtenir le nom et le numéro de téléphone des personnes habitant Montréal. Les quatre premiers exemples ci-dessous correspondent à la construction par étape d'une requête SQL permettant d'obtenir l'information désirée.

1. Jointure sans condition

Une jointure en SQL est un énoncé SELECT dans lequel le mot-clé FROM est suivi d'au moins deux noms de table. Lorsqu'il n'y a aucune condition de sélection (clause WHERE), la jointure donne comme résultat une table qui contient toutes les combinaisons possibles des lignes provenant des tables jointes.

```
SELECT *
FROM PERSONNE, LOCALITE;
```

NO	NOM	TEL	PERSONNE.CP	LOCALITE.CP	LOCALITE
1	Bissonnette, Lise	514-432-3514	H2J 1C4	H2J 1C4	Ville St-Laurent
2	Valjean, Jean	514-123-2292	H3V 2E5	H2J 1C4	Ville St-Laurent
3	Smith, Luc	514-778-7167	H3V 2E5	H2J 1C4	Ville St-Laurent
1	Bissonnette, Lise	514-432-3514	H2J 1C4	H3V 2E5	Montréal
2	Valjean, Jean	514-123-2292	H3V 2E5	H3V 2E5	Montréal
3	Smith, Luc	514-778-7167	H3V 2E5	H3V 2E5	Montréal
1	Bissonnette, Lise	514-432-3514	H2J 1C4	H2J 3A8	Ville St-Laurent
2	Valjean, Jean	514-123-2292	H3V 2E5	H2J 3A8	Ville St-Laurent
3	Smith, Luc	514-778-7167	H3V 2E5	H2J 3A8	Ville St-Laurent

Remarquez ainsi que les champs CP des tables PERSONNE et LOCALITE n'ont pas toujours la même valeur, comme toutes les combinaisons possibles sont effectuées. À votre avis, est-ce souhaitable?

[1] Ces tables sont disponibles sur le site du cours (fichier *jointures.sql*). Vous pouvez les y récupérer pour les importer dans votre compte individuel sur phpMyAdmin pour expérimenter vous-même avec les jointures.

2. Sélection avec condition

L'ajout d'une condition (clause WHERE) permet, comme dans la sélection sans jointure (c'est-à-dire sur une seule table), de ne retenir que certaines lignes. On peut ainsi ne garder que les lignes pour lesquelles les champs CP des deux tables sont égaux.

```
SELECT *
FROM PERSONNE, LOCALITE
WHERE PERSONNE.CP = LOCALITE.CP;
```

NO	NOM	TEL	PERSONNE.CP	LOCALITE.CP	LOCALITE
1	Bissonnette, Lise	514-432-3514	H2J 1C4	H2J 1C4	Ville St-Laurent
2	Valjean, Jean	514-123-2292	H3V 2E5	H3V 2E5	Montréal
3	Smith, Luc	514-778-7167	H3V 2E5	H3V 2E5	Montréal

3. Condition additionnelle

Il est possible d'inclure plus d'une condition dans la clause WHERE pour « raffiner » la sélection, par exemple en liant les conditions avec un ET logique (AND) et ainsi ne conserver que les lignes correspondant à Montréal comme localité.

```
SELECT *
FROM PERSONNE, LOCALITE
WHERE PERSONNE.CP = LOCALITE.CP
AND LOCALITE = 'Montréal';
```

NO	NOM	TEL	PERSONNE.CP	LOCALITE.CP	LOCALITE
2	Valjean, Jean	514-123-2292	H3V 2E5	H3V 2E5	Montréal
3	Smith, Luc	514-778-7167	H3V 2E5	H3V 2E5	Montréal

4. Sélection des colonnes désirées

Finalement, on peut préciser les champs que l'on veut voir apparaître dans la table des résultats simplement en remplaçant l'astérisque (*) en début de requête (qui signifie d'afficher tous les champs) par les noms des champs désirés séparés par une virgule.

```
SELECT NO, NOM, TEL
FROM PERSONNE, LOCALITE
WHERE PERSONNE.CP = LOCALITE.CP
AND LOCALITE = 'Montréal';
```

NO	NOM	TEL
2	Valjean, Jean	514-123-2292
3	Smith, Luc	514-778-7167

C'est le résultat désiré. Notons qu'aucune colonne de la table LOCALITE n'est sélectionnée pour affichage, bien que cette table soit utilisée (et nécessaire) dans la jointure. C'est là une situation tout à fait courante. Remarquez aussi que l'on a pris soin d'inclure, dans les champs affichés, la clé primaire de la table qui permettra, s'il y a deux homonymes demeurant à Montréal, de les distinguer.

Élimination des répétitions

Lorsqu'une requête peut entraîner une réponse qui contient des répétitions (c'est-à-dire plusieurs lignes identiques dans une table), on peut faire éliminer ces répétitions avec la clause DISTINCT (prédicat) placée tout de suite après SELECT.

Remarquez, par exemple, que la requête suivante génère une table avec des lignes qui se répètent.

```
SELECT CP
FROM PERSONNE;
```

CP
H2J 1C4
H3V 2E5
H3V 2E5

Si vous ajoutez DISTINCT devant le champ à afficher, les répétitions seront de ce fait éliminées.

```
SELECT DISTINCT CP
FROM PERSONNE;
```

CP
H2J 1C4
H3V 2E5

Deux fois la même table dans une jointure

On peut utiliser deux fois la même table dans une jointure. On fait alors suivre chacune des deux occurrences par un *alias* (c'est-à-dire une chaîne de caractères qui représente de manière unique chacune des occurrences par exemple P1 pour la première instance de la table PERSONNE et P2, pour la deuxième instance). On utilise alors les alias au lieu du nom de la table ailleurs dans l'énoncé SELECT.

```
SELECT DISTINCT P1.CP
FROM PERSONNE AS P1, PERSONNE AS P2
WHERE P1.CP = P2.CP
AND P1.NO <> P2.NO;
```

P1.CP
H3V 2E5

Exercice : Décrivez en mots ce que fait cette dernière requête.

Réponse : La requête permet d'obtenir la liste des différents codes postaux (prédicat DISTINCT) partagés (condition P1.CP = P2.CP) entre au moins deux personnes différentes (condition P1.NO <> P2.NO)

VALEUR NULL ET CHAÎNE VIDE [2]

Il est possible de définir certains champs afin qu'ils acceptent les valeurs NULL (données absentes) – par ex. pour les champs facultatifs. De plus, les champs de type texte ont aussi la particularité de pouvoir contenir des chaînes vides (données qui n'existent pas). « Valeur NULL » et « chaîne vide » sont deux réalités différentes qu'il est important de bien distinguer, en particulier parce qu'elles ont des comportements différents au niveau de la recherche.

Valeur NULL : définition et comportement en recherche

Soit la table suivante qui contient entre autres le champ VALEUR qui accepte les valeurs NULL (champ facultatif) :

Table demo_NULL

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec valeur
2	*NULL*	enregistrement 2 NULL
3	valeur enregistrement 3	enregistrement 3 avec valeur

La chose importante à retenir d'une valeur NULL est que **ce n'est pas une chaîne de caractères** ce qui entraîne des comportements particuliers avec certains opérateurs de comparaison. C'est le cas de l'opérateur « = » qui sert à vérifier si une chaîne de caractères est égale à un certain contenu textuel. Par exemple, la requête ci-dessous permet d'identifier les enregistrements dont le champ « explication » contient exactement « enregistrement 1 avec valeur » :

```
SELECT *
FROM demo_NULL
WHERE explication = 'enregistrement 1 avec une valeur';
```

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec une valeur

Le mot-clé ici à retenir est que cet opérateur compare des chaînes de caractères. Or, une valeur NULL n'étant pas une chaîne de caractères, l'utilisation de « = » avec une valeur NULL ne retournera jamais rien. Par exemple, la requête SQL ci-dessous ne repère aucun enregistrement :

```
SELECT *
FROM demo_NULL
WHERE valeur = NULL;
```

Il faut donc faire très attention si on veut repérer des champs ayant une valeur NULL, un besoin légitime dans bien des contextes, à ne pas utiliser l'opérateur « = ». L'opérateur à utiliser en ce cas est « IS » :

```
SELECT *
FROM demo_NULL
WHERE valeur IS NULL;
```

no	valeur	explication
2	*NULL*	enregistrement 2 NULL

Si ce que l'on désire repérer est l'inverse, soit les champs dont la valeur n'est pas NULL, vous pouvez utiliser l'opérateur « IS NOT » :

[2] Ces tables sont disponibles sur le site du cours (fichier *vide_null.sql*). Vous pouvez les y récupérer pour les importer dans votre compte individuel sur phpMyAdmin et expérimenter vous-même avec le comportement des valeurs NULL et des champs vides.

```
SELECT *
FROM demo_NULL
WHERE valeur IS NOT NULL;
```

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec une valeur
3	valeur enregistrement 3	enregistrement 3 avec une valeur

Il est aussi possible d'appliquer la négation (NOT) à l'ensemble de la comparaison ainsi : `WHERE NOT (valeur IS NULL)`.

La fonction ISNULL peut aussi être utilisée pour repérer les valeurs NULL : `WHERE ISNULL(valeur)`.

Il y a donc plusieurs manières équivalentes pour repérer les valeurs NULL ou les valeurs non NULL. L'important est d'éviter la seule qui ne fonctionne pas (l'opérateur « = » ainsi que son « opposé » « <> »)!

En sus de l'opérateur « = » qu'il faut éviter avec les valeurs NULL, il faut savoir que les opérateurs « < », « <= », « > » et « >= » utilisés avec des chaînes de caractères pour repérer des valeurs plus petites ou plus grandes que certains contenus, ne fonctionneront pas non plus avec des valeurs NULL. Une valeur NULL n'étant pas une chaîne de caractères, elle n'est ni plus petite (ni plus grande) qu'une chaîne de caractères. Ainsi, bien qu'on puisse à première vue s'attendre à ce que la requête ci-dessous repère tous les enregistrements, ce n'est pas le cas. Les enregistrements dont le champ « valeur » est NULL ne sont pas repérés comme la valeur NULL de l'enregistrement 2 n'est ni « <"A" », ni « >= "A" » comme ce n'est pas une chaîne de caractères :

```
SELECT *
FROM demo_NULL
WHERE valeur < "A"
OR valeur >= "A";
```

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec une valeur
3	valeur enregistrement 3	enregistrement 3 avec une valeur

Valeur NULL *versus* Chaîne vide

Pour les champs textuels, en sus de pouvoir accepter ou non les valeurs NULL, on peut aussi retrouver des chaînes vides. La distinction entre la signification d'une valeur NULL dans un champ et d'une chaîne vide est subtile :

- La valeur NULL signifie que la valeur est absente (c'est-à-dire non saisie pour différentes raisons comme, par exemple, un oubli ou par manque de temps).
- La chaîne vide, quant à elle, correspond à une valeur qui n'existe pas (par exemple, un champ « plaque d'immatriculation » pour une personne n'ayant aucun véhicule motorisé).

Par exemple, imaginons un champ « cellulaire » contenant le numéro de téléphone cellulaire des employés d'une organisation et les deux cas suivants :

- John Smith, n'aimant pas donner ses informations personnelles sans raison valable, ne veut pas donner son numéro de cellulaire qui ne sert que pour des fins privées.
- Son voisin de cubicule, Adams Apple, est un réfractaire notoire à l'utilisation des cellulaires pour des raisons écologiques et n'en possède donc pas.

Dans un cas comme dans l'autre, il n'y aura pas de numéro de cellulaire dans la base de données pour ces deux individus mais pour des raisons très différentes. Pour le premier, c'est une donnée absente; pour le deuxième, une donnée qui n'existe pas.

S'il est important, pour les besoins de l'organisation, de pouvoir distinguer ces deux cas, il faut donc prévoir pour ce champ à la fois la possibilité de valeurs NULL (cas de John Smith) et de chaînes vides (cas d'Adams Apple). Des requêtes SQL pourraient ainsi repérer l'un ou l'autre des cas. Par contre, il faut s'assurer que les personnes responsables de la saisie comprennent bien la différence et fassent la saisie en conséquence; de même pour les personnes qui feront l'interrogation de la base de données.

Chaîne vide : comportement en recherche

Soit la table suivante qui contient un champ de type texte qui permet les chaînes vides (*valeur*) :

Table demo_VIDE

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec une valeur
2		enregistrement 2 VIDE
3	valeur enregistrement 3	enregistrement 3 avec une valeur

La bonne nouvelle concernant la chaîne vide est qu'en tant que chaîne de caractères, elle se comporte donc « normalement » avec les différents opérateurs dans les requêtes SQL (« = », « < », « <= », « > », « >= »). Une chaîne vide, à la fois à la saisie et dans les requêtes, est représentée par deux guillemets simples qui se suivent (') ou deux guillemets doubles qui se suivent ("). Pour identifier les champs avec des chaînes vides, on procède donc ainsi :

```
SELECT *
FROM demo_VIDE
WHERE valeur = '';
```

```
SELECT *
FROM demo_VIDE
WHERE valeur = "";
```

no	valeur	explication
2		enregistrement 2 VIDE

Une chaîne vide étant une chaîne de longueur 0, elle se trouve ainsi être plus petite que toutes les chaînes de longueur 1 et plus. Par exemple :

```
SELECT *
FROM demo_VIDE
WHERE valeur < 'A';
```

no	valeur	explication
2		enregistrement 2 VIDE

Ainsi, la requête ci-dessous repèrera cette fois toutes les notices comme, pour tous les enregistrements, le champ « valeur » contient des chaînes de caractères!

```
SELECT *
FROM demo_VIDE
WHERE valeur < "A"
OR valeur >= "A";
```

no	valeur	explication
1	valeur enregistrement 1	enregistrement 1 avec valeur
2		enregistrement 2 VIDE
3	valeur enregistrement 3	enregistrement 3 avec valeur

PRINCIPAUX TYPES DE CHAMP SOUS MYSQL

Type	MySQL	Explications
Texte	CHAR	• Chaînes de caractères alphanumériques • Longueur **fixe** entre 0 et 255 caractères ○ « fixe » = ajout d'espaces à droite lors de l'enregistrement pour atteindre la longueur définie (espaces enlevées lorsque les données sont extraites)
	VARCHAR	• Pour enregistrer des chaînes de caractères alphanumériques • Longueur **variable** entre 0 à 65 535 caractères (quelque 30 pages de texte!) ○ « variable » = seuls les caractères saisis sont enregistrés (sans ajout d'espaces)
	ENUM	• Liste de **valeurs textuelles prédéfinies** parmi lesquelles **zéro ou une** valeur est retenue • Maximum de 65 535 valeurs en théorie, 3 000 en pratique; pas plus de 255 champs ENUM et SET distincts dans une table • **Avantages** : permet de contrôler la saisie ainsi que de sauver de l'espace de stockage comme les données sont enregistrées sous forme numérique
	SET	• Liste de valeurs textuelles prédéfinies parmi lesquelles **zéro ou plusieurs valeurs** peuvent être retenues (les valeurs sont enregistrées avec une **virgule** comme séparateur). • Maximum de 64 valeurs; pas plus de 255 champs ENUM et SET distincts dans une table • **Avantages** : permet de contrôler la saisie ainsi que de sauver de l'espace de stockage comme les données sont enregistrées sous forme numérique
Date/Heure	DATE	• **Date seulement** sans heure en format AAAA-MM-JJ • Valeurs entre 1000-01-01et 9999-12-31
	DATETIME	• **Date et heure** en format AAAA-MM-JJ HH:MM:SS • Valeurs entre 1000-01-01 00:00:00 et 9999-12-31 23:59:59
	TIME	• **Heure** en format HH:MM:SS • Peut servir pour représenter l'heure ainsi qu'un intervalle de temps entre deux événements • Valeurs entre -838:59:59 à 838:59:59
	YEAR	• **Année** soit en format AAAA (si défini de longueur 4) ou en format AA (si défini de longueur 2) • Valeurs entre 1901 et 2155, ou 0000 (longueur 4)

Type	MySQL	Explications
Numérique	TINYINT	• **Très petites** valeurs entières • Si « signé » : entre -128 et 127 • Si « non signé » : entre 0 et 255 • Occupe 1 bit d'espace
	SMALLINT	• **Petites** valeurs entières • Si « signé » : entre -32 768 et 32 767 • Si « non signé » : entre 0 et 65 535 • Occupe 2 bits d'espace
	MEDIUMINT	• Valeurs entières de **grandeur moyenne** • Si « signé » : entre -8 388 608 et 8 388 608 • Si « non signé » : entre 0 et 16 777 215 • Occupe 3 bits d'espace
	INT	• Valeurs entières **relativement grandes** • Si « signé » : entre -2 147 483 648 et 2 147 483 647 • Si « non signé » : entre 0 et 4 294 967 295 • Occupe 4 bits d'espace
	BIGINT	• **Très grandes** valeurs entières • Si « signé » : entre -9 223 372 036 854 775 808 et -9 223 372 036 854 775 807 • Si « non signé » : entre 0 et 18 446 744 073 709 551 615 • Occupe 8 bits d'espace
	FLOAT	• Nombres réels (c'est-à-dire avec décimales) • *Attributs facultatifs* : FLOAT(M,D) où M = nombre total de chiffres et D = nombre de chiffres après la virgule, par exemple 99,999 pour FLOAT(5,3) • Occupe 4 bits d'espace
	DOUBLE	• Nombres réels (c'est-à-dire avec décimales) • *Attributs facultatifs* : DOUBLE(M,D) où M = nombre total de chiffres et D = nombre de chiffres après la virgule, par exemple 99,999 pour DOUBLE(5,3) • Occupe 8 bits d'espace (donc plus précis que FLOAT)
Logique	BOOLEAN	• Synonyme de TINYINT(1) • 0 est considéré comme Faux, les valeurs >0 sont considérés comme Vrai

PRINCIPALES EXPRESSIONS SOUS MYSQL

Introduction

Il existe deux endroits où une **expression** peut survenir dans un énoncé SELECT en SQL :

1- **Avant la clause FROM**. Cela signifie alors qu'on ne veut pas simplement faire afficher un champ tel quel, mais plutôt une certaine expression calculée à partir d'un ou de plusieurs champs. Voici un exemple d'un tel usage d'une expression :

```
SELECT UCASE(NOM) FROM ETUD;
```

On peut bien sûr demander l'affichage d'une ou plusieurs expressions et d'un ou plusieurs champs dans le même énoncé SELECT :

```
SELECT NO_PROF, UCASE(NOM), BUREAU FROM PROF;
```

2- **Dans la clause WHERE**. En réalité, l'ensemble d'une clause WHERE est simplement une expression, qui peut être plus ou moins complexe selon les cas, et qui est souvent composée de sous-expressions reliées entre elles par des opérateurs booléens (le plus souvent, AND : le « et » logique).

La particularité d'une expression constituant une clause WHERE est qu'elle doit « retourner » une valeur de type booléen (ou « logique »). Une valeur booléenne (ou « logique ») est l'une des deux valeurs « VRAI » ou « FAUX ».

Le fait que la clause WHERE d'un énoncé SELECT doive retourner une valeur booléenne veut donc dire qu'en « calculant » la valeur de l'expression pour chacune des lignes de la table à laquelle s'applique l'énoncé, on doit obtenir à chaque fois une valeur finale de « VRAI » ou « FAUX ». Notez que certaines valeurs intermédiaires de l'expression peuvent ne pas être booléennes; l'important est que la valeur *finale* soit booléenne.

Supposons que NOM soit le nom d'un champ (de type caractère) d'une table à laquelle on applique un énoncé SELECT. Voici un exemple d'expression retournant une valeur booléenne :

```
NOM LIKE 'ber%'
```

Il s'agit d'une forme de comparaison entre deux chaînes de caractères, utilisant l'opérateur LIKE. Cette comparaison donne la valeur VRAI si la chaîne de gauche est conforme au patron de recherche donné par la chaîne de droite. Dans le patron de recherche, le caractère « % » est un caractère générique qui représente 0, 1 ou plusieurs caractères quelconques; il représente donc en fait la troncature. Notez que les valeurs comparées elles-mêmes sont de type caractère, mais le résultat final de l'expression est soit VRAI, soit FAUX; l'expression donne donc bien un résultat de type booléen. En cette qualité, elle pourrait être utilisée comme clause WHERE dans un SELECT :

```
SELECT * FROM ETUD WHERE NOM LIKE 'ber%'
```

La sélection repêchera donc exactement les lignes de ETUD dans lesquelles le champ NOM commence par « ber » (Berriman, Bernier, Bernard, Bergeron, etc.). L'opérateur LIKE est discuté plus en détail dans la section sur les OPÉRATEURS DE RELATION ci-dessous.

Une expression qui retourne une valeur booléenne s'appelle (très justement, d'ailleurs) une *condition*. C'est pourquoi on réfère souvent à la clause WHERE d'un énoncé SELECT comme à la *condition de sélection* de l'énoncé.

Il est à noter que les expressions survenant *avant* la clause FROM dans un énoncé SELECT ne sont pas soumises, comme la clause WHERE, à des restrictions sur le type de valeurs qu'elles peuvent retourner.

Forme générale d'une expression

Une **expression** en général est constituée d'éléments des huit types suivants :

1. des constantes,
2. des noms de champ,
3. des noms de variable,
4. des opérateurs de transformation,
5. des opérateurs de relation,
6. des noms de fonctions,
7. des opérateurs booléens,
8. des parenthèses.

Les parenthèses exceptées, chacun des éléments qui composent une expression « retourne » une valeur, laquelle est déterminée selon des règles propres à chaque type d'élément. Les plus importantes de ces règles sont présentées ci-dessous.

Lorsque MySQL évalue une expression, il détermine d'abord la valeur retournée par les constantes, les noms de champ et les noms de variable, puis celle retournée par les opérateurs et les fonctions (comme en arithmétique élémentaire), jusqu'à ce que toute l'expression soit évaluée. La valeur ainsi obtenue est ce qu'on appelle la « valeur retournée par l'expression ». Comme en arithmétique, les parenthèses peuvent influencer l'ordre d'évaluation des opérateurs et des fonctions.

Les différentes valeurs qui peuvent être retournées par une expression ont un « type », tout comme les champs d'une table ont un type. Les types que peut avoir une valeur sont : caractères, numérique, date et logique. Tel que dit précédemment, une « condition » est simplement une expression retournant une valeur logique (c'est-à-dire booléenne) : VRAI ou FAUX.

Voici quelques mots d'explications sur les huit types d'éléments que l'on peut rencontrer dans une expression et sur les règles qui gouvernent l'évaluation des éléments de chacun de ces types :

1. **CONSTANTES**. Une constante peut être un nombre, **sans guillemet** (par ex. 1, 2, 25); une chaîne de caractères où il **faut** taper les guillemets (par ex. 'A', 'Carmel, Lucie'); une date qui doit être mise entre guillemets (par ex. '1990-01-01'); ou une valeur logique : true pour Vrai et false pour Faux. Une constante se retourne « elle-même » comme valeur. Par exemple, la constante 'ABC' retourne comme valeur la chaîne de caractères 'ABC' (qui est bien sûr de type caractère).

   Les guillemets doubles sont la plupart du temps acceptés au lieu des guillemets simples, mais *pas partout*. C'est pourquoi nous avons systématiquement utilisé les guillemets simples dans nos exemples. Pour inscrire un guillemet simple comme caractère dans une chaîne, il suffit de le doubler (l'inscrire deux fois, et non inscrire un guillemet double), par exemple : 'L''hiver', ou le faire précéder d'une barre oblique inversée, par exemple 'L\'hiver'.

2. **NOMS DE CHAMP**. Il s'agit des noms de champ de la table à laquelle s'applique l'énoncé. Par exemple, dans un énoncé travaillant sur la table ETUD, les noms de champ sont NO_ETUD, NOM, ADRESSE et DAT_NAIS. Pour chaque ligne sur laquelle l'énoncé travaille, la valeur retournée par un nom de champ est le contenu du champ correspondant dans la ligne en question. Un champ numérique retourne une valeur numérique, un champ logique une valeur logique, un champ date une valeur date, et un champ caractère, une valeur caractère.

Si un énoncé implique un nom de champs identique entre deux tables (par exemple, pour une jointure) qu'il peut y avoir ambiguïté sur la provenance d'un champ, on fait précéder le nom du champ du nom de la table d'où il provient suivi d'un point (e.g. ETUD.NO_ETUD).

3. **NOMS DE VARIABLES**. Les noms de variable sont analogues aux constantes, mais au lieu de retourner toujours la même valeur, elles retournent une valeur définit, par exemple, par l'utilisateur. Les noms de variables sont précédés par un arobas (@).

4. **OPÉRATEURS DE TRANSFORMATION**. Il s'agit d'opérateurs permettant d'effectuer certains calculs sur des valeurs. Les opérateurs +, -, * et /, appliqués à des valeurs numériques, correspondent aux quatre opérations arithmétiques usuelles.

5. **OPÉRATEURS DE RELATION**. Les opérateurs de relation sont similaires aux opérateurs de transformation, mais retournent toujours une valeur logique. Il s'agit par exemple d'opérateurs comme = (égal à), <> (différent de), <= (plus petit ou égal à), >= (plus grand ou égal à). Les opérateurs de relation retournent la valeur VRAI ou FAUX dépendant que la comparaison à laquelle ils correspondent est vraie ou fausse avec les valeurs qui leur sont fournies. Ainsi, l'expression 1=2 retourne la valeur FAUX, alors que 1<>2 retourne VRAI. L'expression 'ABC'<'B' a la valeur VRAI.

   Un opérateur de relation utile pour les champs de type texte est l'opérateur LIKE. Cet opérateur recherche une chaîne de caractères dans un ou des champs de type caractère, et offre des possibilités de masquage et de troncature, avec les caractères '_' (masquage d'un seul caractère) et '%' (troncature, utilisable à gauche, à droite ou à l'intérieur d'une chaîne).

   Un autre opérateur de relation important est le IN. Utilisé le plus souvent en conjonction avec une sous-requête, il permet de vérifier si une valeur apparaît dans une liste (par ex. NO_ETUD IN (10001, 10002)).

6. **NOMS DE FONCTIONS**. Il s'agit de fonctions permettant de calculer certaines valeurs à partir d'autres valeurs[3]. Dans une condition de sélection, les valeurs sur lesquelles on appliquera des fonctions seront surtout des valeurs provenant de champs de la table à laquelle s'applique l'énoncé SELECT. Les valeurs sur lesquelles une fonction doit opérer sont données entre parenthèses après le nom de la fonction, et séparées entre elles par des virgules s'il y en a plus d'une. On appelle ces valeurs les « arguments » de la fonction. Le nombre d'arguments que l'on doit fournir à une fonction, de même que leur type, dépendent de la fonction elle-même, et sont indiqués dans la documentation de la fonction. La valeur retournée par une fonction est le résultat d'un calcul effectué par la fonction sur les valeurs spécifiques qu'on lui fournit en arguments.

   Les fonctions **UCASE** et **LCASE** prennent toutes deux en argument une valeur de type caractère et retournent respectivement la chaîne de caractères en majuscules et en minuscules. À titre d'exemple, l'expression LCASE('Géographie-102') retourne la valeur 'géographie-102'. Évidemment, si le champ TITRE d'une des lignes d'une table à laquelle on applique un énoncé SELECT contient la chaîne 'Géographie-102', alors l'expression LCASE(TITRE) retournera aussi la valeur 'géographie-102' pour cette ligne.

   La fonction **CONCAT** permet de concaténer (c'est-à-dire de « coller ensemble ») les valeurs de différentes expressions pour un enregistrement (valeurs d'un champ, chaîne de texte fixe, résultat d'une autre fonction, etc.). À titre d'exemple, l'expression CONCAT(NO_ETUD," / ", UCASE(NOM)) retournera '10004 / ASIMOV, ISAAC' pour la ligne correspondant à cet étudiant.

   La fonction **GROUP_CONCAT** permet de concaténer (c'est-à-dire de « coller ensemble ») les valeurs des différents enregistrements regroupés par une clause GROUP BY. À titre d'exemple,

[3] D'autres exemples que ceux donnés peuvent être trouvés à l'URL https://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html.

l'expression GROUP_CONCAT(ETUD.NO_ETUD separator ', ') pour un regroupement par cours retournera la liste concaténée de tous les numéros d'étudiants suivant le même cours comme '10004, 10002'.

Les fonctions **CURRENT_DATE()**, **CURRENT_TIME()**, **CURRENT_TIMESTAMP()** sont des fonctions SANS ARGUMENT, ce qui fait qu'elle apparaît toujours sous la forme « CURRENT_DATE() » (avec les parenthèses et rien dedans) dans une expression. Un synonyme de CURRENT_TIMESTAMP() est NOW(). Ces fonctions retournent une valeur de type date, respectivement égales à la date courante du jour, à l'heure courante du jour, à la date et l'heure courante du jour, telles que réglées au niveau du serveur. On peut utiliser ces fonctions comme n'importe quelle valeur de type date dans une expression.

La fonction **DATE_FORMAT** est une fonction permettant de changer le format d'un champ de type « date/heure ». Elle prend la forme : DATE_FORMAT(« nom du champ », « format désiré ») où « format désiré » est une combinaison des différentes possibilités d'affichage du jour, du mois, de l'année et de l'heure, combinaison indiquée entre guillemets simples ou doubles. Les principales possibilités sont[4] :

`%d`	affiche le jour en format numérique à 2 chiffres (e.g. 01, 31)
`%a`	affiche le jour en format textuel abrégé (lun., mar., etc.)
`%W`	affiche le jour en format textuel complet (lundi, mardi, etc.)
`%m`	affiche le mois en format numérique à 2 chiffres (e.g. 01, 12)
`%b`	affiche le mois en format textuel abrégé (janv, fév, etc.)
`%M`	affiche le mois en format textuel complet (janvier, février, etc.)
`%y`	affiche l'année en format numérique à 2 chiffres (e.g. 99, 01)
`%Y`	affiche l'année en format numérique à 4 chiffres

Exemples : soit le champ date_naissance avec comme valeur « 10 novembre 2009 »

- DATE_FORMAT(date_naissance,'%Y') affichera « 2009 »
- DATE_FORMAT(date_naissance,'%y') affichera « 09 »
- DATE_FORMAT(date_naissance,'%W %d %M %Y') affichera « mardi 10 novembre 2009 »

Toute chaîne de caractère dans le format désiré qui ne correspond pas à un des éléments de date ou d'heure sera indiqué tel quel. Par exemple, DATE_FORMAT(date_naissance,'%W, le %d %M %Y') affichera « mardi, le 10 novembre 2009 ».

Le système vous retourne les noms des mois ou des jours en anglais et vous les voudriez en français? Ajoutez la ligne suivante avant votre énoncé SELECT pour modifier la langue utilisée :

```
SET lc_time_names = 'fr_FR';
```

La fonction **DATEDIFF** permet de calculer la différence entre deux dates en nombre de jours.

[4] Une liste complète est disponible à l'URL http://www.w3schools.com/sql/func_date_format.asp.

La fonction **EXISTS**, utilisée avec une sous-requête, permet de tester si le résultat de la sous-requête est vide ou non.

La fonction **IF** permet de sélectionner une de deux valeurs en fonction d'une condition. Cette fonction comporte trois arguments : IF(*condition*, *valeur si vrai*, *valeur si faux*). Par exemple, l'expression « IF (prix > 1000, 'Cher', 'Abordable') » retournera la chaîne 'Cher' si prix est supérieur à 1000 et 'Abordable' autrement.

La fonction **ISNULL** permet de vérifier si un champ (ou une valeur quelconque) est NULL. Par exemple, « IsNull(local) » retournera la valeur booléenne VRAI si le champ « local » contient la valeur NULL, et FAUX autrement.

La fonction **IFNULL** permet de vérifier si un champ (ou une valeur quelconque) est NULL et, si c'est le cas, d'afficher autre chose que NULL. Par exemple, « IfNull(local,'Aucun local assigné') » retournera la chaîne de caracètre 'Aucun local assigné' si le champ « local » contient la valeur NULL, et le local autrement.

La fonction **ROUND** permet d'arrondir une valeur numérique. Cette fonction demande deux arguments : (1) le nom du champ, et (2) le nombre de décimales souhaité. Par exemple, « round(note,0) » permet d'arrondir le contenu du champ « note » à un nombre entier (c'est-à-dire sans décimales).

7. **OPÉRATEURS BOOLÉENS**. Les opérateurs booléens sont similaires aux opérateurs de transformation, mais opèrent toujours sur des valeurs logiques, et retournent toujours également une valeur logique. Il s'agit de AND, OR et NOT. Ces opérateurs permettent de combiner plusieurs valeurs logiques selon les règles de la logique booléenne.

8. **PARENTHÈSES**. Les parenthèses sont utilisées pour modifier l'ordre d'évaluation des expressions. Les différents opérateurs dans Access ont une priorité relative qui détermine lesquels sont exécutés en premier, en l'absence de parenthèses. On peut mémoriser ces priorités relatives, et n'utiliser les parenthèses que lorsqu'elles sont absolument nécessaires. Une autre approche, plus simple, consiste à toujours utiliser des parenthèses dès qu'il y aurait ambiguïté possible, même s'il pouvait s'avérer qu'elles ne soient pas nécessaires. Il est toujours permis d'utiliser autant de niveaux de parenthèses que l'on veut.

   Ainsi, l'expression :

```
(LCASE(CONCAT('A','B')))
```

   est tout à fait valide, et retourne la valeur caractère 'ab'