

Formation SQL, le DQL

Formation SQL:

Le Data Query Language (DQL)

Section 1:

Introduction

Introduction

Ce module est centré sur le **DQL** (*Data Query Language*) ou **langage de requête de données**

Il s'agit d'un **sous-langage** du langage SQL. Il est, comme son parent, un **langage de requête**.

Son unique rôle est la **récupération de données** au sein d'une base. Il permet d'**interroger une BDD** pour en récupérer **les informations** mais ne permet ni la modification, ni la suppression, ni l'ajout de données. Ces fonctions seront apportées par d'autres sous-langages.

Avant-Propos

Cette section traite du langage DQL, qui ne nous permet pas d'injecter ou de supprimer nous-mêmes nos données.

Des bases de données sont donc fournies avec le cours pour permettre la manipulation du langage.

Son implémentation dépendra de l'interface utilisée.

Si vous avez des connaissances sur d'autres sous-langages du SQL, n'hésitez pas à suivre le cours avec vos propres bases de données.

Choisir sa base de données

Avant d'envoyer des requêtes, il me faut choisir dans quelle base de données je souhaite travailler, nos deux premières requêtes seront:

1. Pour afficher la liste des BDD disponibles:

```
SHOW DATABASES;
```

2. Pour utiliser celle de mon choix:

```
USE nom_de_la_bdd;
```

Voir les tables présentes dans la BDD

Une fois que je suis dans la base de données de mon choix, je peux consulter la liste des tables présentes:

- Pour MySQL/MariaDB:

```
SHOW TABLES;
```

Section 2:

Bases du DQL

SELECT... FROM...

SELECT

En SQL, la commande **SELECT** est utilisée pour **récupérer des données** spécifiques à partir d'une base de données. C'est l'une des commandes les plus fondamentales et couramment utilisées mais aussi probablement l'instruction la plus complexe du langage SQL.

C'est également la commande **centrale** du sous-langage DQL, elle nous sera indispensable dans la quasi-totalité de nos requêtes.

Elle permet de récupérer une, plusieurs ou toutes les colonnes de notre table.

Structure de la commande

La commande se décompose en deux parties **SELECT** et **FROM** sous cette forme:

```
SELECT nom_de_la_colonne  
FROM nom_de_la_table;
```

Sélectionner toutes les colonnes de ma table

Pour sélectionner toutes les colonnes d'une table, on utilise comme raccourci **l'étoile** * après notre SELECT:

```
SELECT *  
FROM nom_de_la_table;
```

Sélectionner plusieurs colonnes de ma table

De la même manière, il est possible de récupérer plusieurs colonnes en une seule requête, il suffit de les séparer d'une virgule

```
SELECT colonne1, colonne2, colonne3  
FROM nom_de_la_table;
```

Attention: On peut sélectionner plusieurs colonnes mais on évitera de sélectionner **plusieurs tables** lors d'une requête. (voir "Récupérer les colonnes de plusieurs tables")

Exercice: Récupérer des données de notre table

Essayons maintenant nous-même de récupérer les données de la table **Users** de notre base de données.

1. Dans un premier temps, récupérez l'intégralité de la table **Users**
2. Modifiez la requête pour n'afficher que les colonnes **first_name**, **last_name** et **job**

Récupérer les colonnes de plusieurs tables

Si notre BDD comporte plusieurs tables, il est possible de récupérer les colonnes de ces différentes tables, mais il sera obligatoire de préciser le nom de la table avant chaque colonne

```
SELECT table1.colonne1, table2.colonne1, table2.colonne2  
FROM table1, table2;
```

Attention: On nomme ce type de jointure une **jointure implicite**. C'est une méthode déprécié et peu lisible, on préférera l'utilisation de **jointures explicites** que nous verrons plus loin dans le cours

Filtrer les résultats:

SELECT...WHERE

La clause **WHERE**

La clause **WHERE** est une commande qui vient se greffer à notre **SELECT**, elle a **une fonction de filtre**, elle permet de ne sélectionner que les données qui correspondent au(x) filtre(s) que nous aurons choisi.

Elle prend la forme d'une **expression conditionnelle**.

*Note: Lorsque la clause **WHERE** est omise, toutes les lignes sont affichées (équivalent à **WHERE TRUE**).*

Structure de la commande

La clause **WHERE** vient se situer directement derrière la clause **FROM**:

```
SELECT nom_de_la_colonne  
FROM nom_de_la_table  
WHERE instruction_conditionnelle;
```

Pour indiquer la condition que je souhaite, je vais utiliser les **opérateurs de comparaison**

Les opérateurs de comparaison

Opérateur	Description	Exemple
=	Égalité	WHERE nom = prenom
>	Supérieur à	WHERE age > 18
<	Inférieur à	WHERE days < 20
>=	Supérieur ou égal	WHERE number >= 30
<=	Inférieur ou égal	WHERE date <= 1995
<> ou !=	Différent de	WHERE age != 21


Il suffit d'appliquer les opérateurs de comparaison sur les colonnes de notre table pour filtrer.

Exemple

Dans ma table **Users**, je ne souhaite garder que les utilisateurs qui ont moins de 30ans et afficher leur prénom, leur nom et leur âge, j'enverrai la requête suivante:

```
SELECT first_name, last_name, age  
FROM users  
WHERE age < 30;
```

Comparaison de texte

Il est également possible de comparer des données sous forme de texte, dans ce cas il faudra entourer le texte entre apostrophes ' ' 

```
SELECT first_name, last_name, age  
FROM Users  
WHERE first_name = 'Alice';
```

Note: Les guillemets fonctionnent également, mais par convention il est préférable d'utiliser les quotes (apostrophes)

Exercice: Filtrer les données de notre table avec WHERE.

Essayons maintenant nous-même de filtrer les données de la table **Users** de notre base de données.

1. Dans un première requête, récupérez tous les utilisateurs dont le métier n'est pas développeur
2. Dans une seconde requête, récupérez tous les utilisateurs dont le prénom est John.
3. Dans une dernière requête, récupérez tous les utilisateurs dont le salaire est supérieur ou égal à 3000.

CORRECTION: Filtrer les données de notre table avec WHERE.

```
SELECT first_name, last_name, job  
FROM Users  
WHERE job != 'Developer';
```

```
SELECT first_name, last_name  
FROM Users  
WHERE first_name = 'John';
```

```
SELECT first_name, last_name, salary  
FROM Users  
WHERE salary >= 3000
```

Projections et sélections

En SQL, les termes **projection** et **sélection** ont des significations différentes et spécifiques:

- **La projection** consiste à **spécifier quelles colonnes d'une table doivent être retournées** dans les résultats d'une requête. Cela signifie que vous pouvez choisir de ne récupérer que certaines colonnes, en excluant les autres.
- **La sélection** se réfère à l'**opération de filtrage des lignes en fonction de critères spécifiques**, en fonction de conditions définies dans la clause WHERE.

Projections et sélections

Bien sûr, **les deux opérations sont cumulables**:

```
SELECT first_name, last_name  
FROM users  
WHERE job = 'Developer';
```

Projection : first_name, last_name – Vous ne récupérez que ces colonnes.

Sélection : WHERE job = 'Developer' – Vous ne récupérez que les lignes où le travail de l'employé est "Developer".

Ces deux opérations sont fondamentales pour obtenir les données désirées d'une base de données relationnelle.

Cumuler les conditions:

Opérateurs logiques

AND, OR et NOT

Les opérateurs AND et OR

En plus des opérateurs de comparaison, notre instruction conditionnelle WHERE a la possibilité d'utiliser des **opérateurs logiques: AND, OR et NOT**.

Ils donnent la possibilité de filtrer en cumulant les conditions:

- L'opérateur **AND** sélectionnera les lignes qui remplissent **toutes les conditions**
- L'opérateur **OR** sélectionnera les lignes qui remplissent **au moins l'une des conditions**

Exemple: Le AND

Je souhaite récupérer toutes les lignes dont l'utilisateur s'appelle David **ET** dont le métier est docteur.

```
SELECT first_name, last_name, job  
FROM Users  
WHERE first_name = 'David' AND job = 'Doctor';
```

Les utilisateurs s'appelant David mais n'étant pas docteur ne seront pas sélectionnés. Tout comme les docteurs ne s'appelant pas David. Les deux conditions doivent être **VRAI**

Exemple: Le OR

Je souhaite récupérer toutes les lignes dont l'utilisateur s'appelle David **OU** dont le métier est docteur.

```
SELECT first_name, last_name, job  
FROM Users  
WHERE first_name = 'David' OR job = 'Doctor';
```

Les utilisateurs s'appelant David mais n'étant pas docteur seront sélectionnés. Tout comme les docteurs ne s'appelant pas David. Seule **L'UNE** des deux conditions doit être **VRAI** pour être récupérée.

Exercice: Utiliser les opérateurs OR et AND

Dans notre table **Users**, essayez de filtrer de cette manière:

1. Dans un première requête, récupérez tous les utilisateurs dont l'âge est inférieur à 30ans ou supérieur et égal à 35ans.
2. Récupérez ensuite tous les utilisateurs dont le métier est professeur et le salaire est supérieur à 2600.

CORRECTION: Filtrer les données de notre table avec WHERE.

```
SELECT first_name, last_name, age  
FROM Users  
WHERE age < 30 OR age >= 35;
```

```
SELECT first_name, last_name, age, salary  
FROM Users  
WHERE job = 'Teacher' AND salary > 2600;
```

L'opérateur NOT

L'opérateur **NOT** est un opérateur logique de négation qui est utilisé pour **inverser** une condition.

Il est généralement utilisé en combinaison avec les opérations de comparaison pour obtenir le **résultat inverse** de la condition.

Attention: Ne pas confondre avec "!=" ou "<>".

NOT est utilisé pour **inverser une condition** logique, tandis que != est utilisé pour **comparer deux valeurs** et vérifier leur différence. Ils peuvent cependant être parfois utilisés à des fins similaires.

Exemple: Le NOT

Je souhaite récupérer toutes les lignes dont l'utilisateur n'est pas né à New York.

```
SELECT first_name, last_name, birth_location  
FROM Users  
WHERE NOT birth_location = 'New York';
```

Combinaison d'opérateurs logiques

Ces trois opérateurs logiques sont **cumulables** au sein d'une seule requête. On utilisera les parenthèses `()` pour en définir l'ordre des priorités. Prenons la requête suivante:

```
SELECT first_name, last_name, birth_location, job
FROM Users
WHERE birth_location = 'New York' AND (job = 'Teacher' OR job = 'Developer');
```

Cette requête sélectionnera tous les utilisateurs dont la localisation est New York et qui sont professeurs ou développeurs.

Exercice: Récapitulatif

Créez une requête qui permet de récupérer toutes les personnes qui sont nées à New York, dont le salaire est compris entre 3000 et 3500 (compris) et qui ne sont ni docteur ni avocat.

- **Toutes les conditions doivent tenir en une seule requête**
- **Les trois opérateurs logiques : AND, OR et NOT doivent être utilisés.**

Correction: Récapitulatif

```
SELECT *  
FROM Users  
WHERE birth_location = 'New York'  
AND (salary >= 3000 AND salary <= 3500)  
AND NOT (job = 'Doctor' OR job = 'Lawyer');
```

N'hésitez pas à passer à la ligne sur les requêtes complexes pour gagner en lisibilité.

Les retours à la ligne seront de toute façon ignorés par votre SGBD

Supprimer les doublons:

DISTINCT

La clause **DISTINCT**

La clause **DISTINCT** est utilisée **sur la projection**.

Elle sert à spécifier que les résultats d'une requête doivent inclure uniquement les **valeurs uniques** d'une colonne.

En d'autres termes, elle permet **d'éliminer les doublons dans les résultats** de la requête.

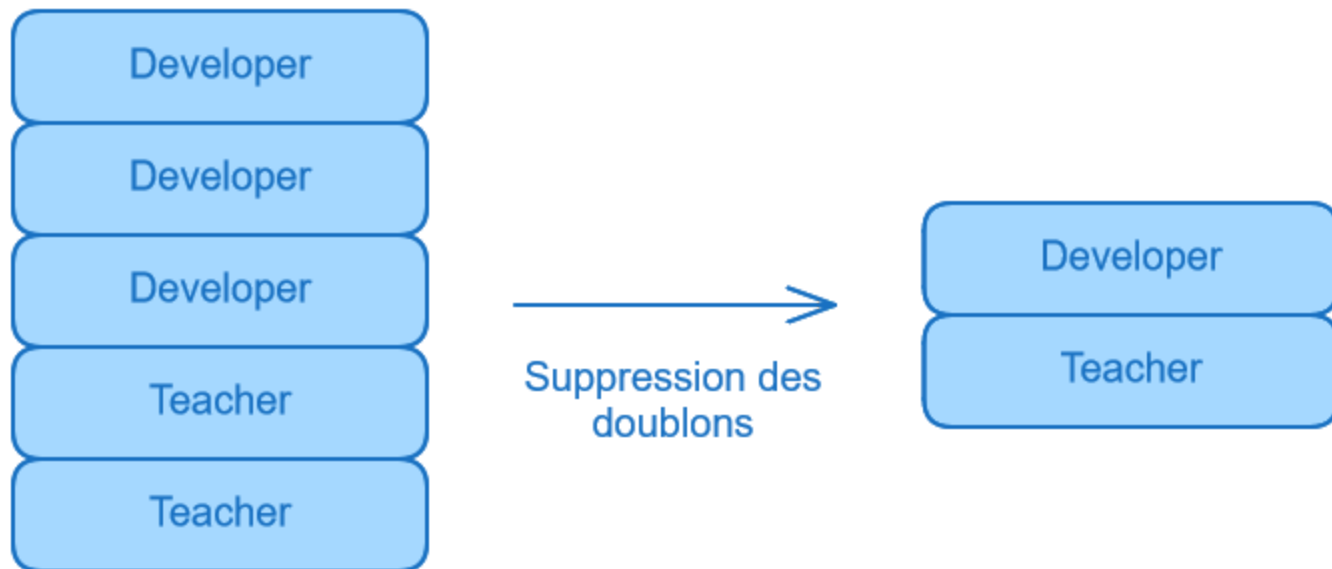
Comme la plupart des clauses, elle est compatible avec WHERE.

Syntaxe:

```
SELECT DISTINCT nom_de_la_colonne1, nom_de_la_colonne2  
FROM nom_de_la_table
```

La clause DISTINCT

Avec un `SELECT DISTINCT JOB`, j'exclus tous les métiers qui vont se répéter dans la récupération de mes datas.



Exemple

Je cherche à récupérer tous les jobs uniques qui existent pour les gens nés à New York:

```
SELECT DISTINCT job  
FROM Users  
WHERE birth_location = 'New York';
```


Valeurs uniques de plusieurs colonnes

Vous pouvez également utiliser **DISTINCT pour plusieurs colonnes**. Dans ce cas, la combinaison des valeurs des colonnes spécifiées doit être unique pour qu'une ligne soit retournée.

```
SELECT DISTINCT first_name, last_name  
FROM users;
```

Cette requête retourne toutes les combinaisons uniques de prénoms et de noms de famille dans la table users.

Nous avons un homonyme qui s'est glissé dans notre table, il y a deux "David Lee", seul le premier recensé sera affiché.

Intervalles et suites de valeurs:

IN et BETWEEN

IN et BETWEEN

En plus des opérateurs logiques OR, AND et NOT, nous pouvons affiner le filtre WHERE grâce aux opérateurs IN et BETWEEN:

- **IN** permet de filtrer les données dans une requête en utilisant des **valeurs précises**. On peut ainsi indiquer facilement une suite de valeurs.
- **BETWEEN** permet de filtrer les données dans une requête en fonction de **plages de valeurs**.

IN

Par exemple, si je recherche tous les utilisateurs qui viennent **spécifiquement** de Londres et de Paris

```
SELECT first_name, last_name, birth_location  
FROM Users  
WHERE birth_location IN ('Paris', 'London');
```

Si vous êtes attentifs, vous constaterez que cette requête est similaire à:

```
SELECT first_name, last_name, birth_location  
FROM Users  
WHERE birth_location = 'Paris' OR birth_location = 'London';
```

IN + NOT

L'opérateur IN est cumulable avec NOT également donc si je recherche tous les utilisateurs qui **NE** viennent **PAS** de Londres et de Paris:

```
SELECT first_name, last_name, birth_location
FROM Users
WHERE birth_location NOT IN ('Paris', 'London');
```

Cette requête est similaire à:

```
SELECT first_name, last_name, birth_location
FROM Users
WHERE birth_location != 'Paris' OR birth_location != 'London';
```

BETWEEN

La clause **BETWEEN** est utilisée pour récupérer une plage précise de valeurs. Par exemple si je veux récupérer une plage d'utilisateurs dont l'âge est compris entre 30 et 35 ans:

```
SELECT first_name, last_name, age  
FROM Users  
WHERE age BETWEEN 30 AND 35;
```

NOTE: Elle est inclusive, c'est à dire que les valeurs minimales et maximales sont incluses dans la récupération des données (dans notre exemple, les utilisateurs de 30 et 35ans seront inclus)

BETWEEN + NOT

Comme **IN**, **BETWEEN** peut être cumulé avec **NOT** pour exclure toute une plage de valeurs. Par exemple si je veux récupérer une plage d'utilisateurs dont l'âge n'est pas compris entre 30 et 35 ans:

```
SELECT first_name, last_name, age  
FROM Users  
WHERE age NOT BETWEEN 30 AND 35;
```

NOTE: Dans la même logique, les valeurs minimales et maximales sont également exclues (dans notre exemple, 30 et 35 sont donc exclus)

Exercice: Manipulation de données dans la table "Users"

Dans notre table **Users**, en utilisant au moins pour une requête IN et pour une autre BETWEEN:

1. Sélectionnez tous les enregistrements où le métier est "Engineer"
2. Sélectionnez les prénoms et les noms de famille des utilisateurs nés à Londres, Paris ou Berlin
3. Sélectionnez les utilisateurs dont l'âge est compris entre 25 et 35 ans
4. Sélectionnez les utilisateurs qui sont à la fois des développeurs et dont le salaire est supérieur à 2500

Correction:

```
SELECT * FROM Users  
WHERE job = 'Engineer';
```

```
SELECT first_name, last_name FROM Users  
WHERE birth_location IN ('London', 'Paris', 'Berlin');
```

```
SELECT * FROM Users  
WHERE age BETWEEN 25 AND 35;
```

```
SELECT * FROM Users  
WHERE job = 'Developer' AND salary > 2500;
```

Tri des résultats

ORDER BY

ORDER BY

Maintenant que nous savons **filtrer** efficacement, il peut être utile de **trier** également.

La clause **ORDER BY** permet de classer nos données par ordre ascendant ou descendant en ciblant une colonne.

Si elle cible une colonne qui contient du texte, elle classera donc par **ordre alphabétique**.

Si elle contient des nombres, **du plus petit au plus grand**

Syntaxe

```
SELECT first_name, age  
FROM users  
ORDER BY age;
```

Ici, les gens seront classés par âge. Bien sûr, cette clause est cumulable avec **WHERE**:

```
SELECT first_name, age  
FROM users  
WHERE age < 50  
ORDER BY age;
```

ASC, DESC

Par défaut, l'ordre de tri est croissant. Mais il est également possible de ranger par ordre décroissant grâce à la clause **DESC**:

```
SELECT *  
FROM users  
ORDER BY birth_location DESC;
```

On peut également préciser **ASC** quand on trie par ordre croissant, mais cela mène au résultat que de ne pas apporter de précision, on gagne cependant en lisibilité.

ORDER BY (avec plusieurs colonnes)

```
SELECT *  
FROM users  
ORDER BY last_name DESC, age ASC;
```

Il est tout à fait possible d'utiliser **ORDER BY** avec plusieurs colonnes. L'ordre à son importance.

Dans l'exemple ci-dessus, les utilisateurs seront classés par leur nom **de façon descendante** puis par leur âge **de façon ascendante** si leur nom est identique.

Limiter les résultats

LIMIT/OFFSET

LIMIT

La clause **LIMIT** est utilisé pour limiter le nombre de lignes retournées par une requête SELECT.

Elle permet de **contrôler la quantité de données affichées** ou récupérées dans le résultat de la requête.

La clause LIMIT est souvent utilisée en conjonction avec l'ordre ORDER BY pour **trier les résultats avant de limiter le nombre de lignes**.

Syntaxe

La clause LIMIT vient se placer tout à la fin de nos filtres, **après** le ORDER BY:

```
SELECT column1, column2, ...  
FROM table  
WHERE conditions  
ORDER BY column  
LIMIT number;
```

Exemple

```
SELECT first_name, last_name, salary  
FROM Users  
ORDER BY salary DESC  
LIMIT 5;
```

Avec cet exemple, je vais d'abord classer mes salaires du plus élevé au plus bas. Puis je ne vais garder que les 5 premiers résultats, cette requête me permet donc de récupérer **les 5 utilisateurs avec le plus gros salaire** de ma base de données

OFFSET

La clause LIMIT peut être enrichi d'un **OFFSET**. Sa fonction est de **décaler notre résultat** du nombre de lignes que l'on souhaite:

```
SELECT first_name, last_name, salary
FROM Users
ORDER BY salary DESC
LIMIT 5 OFFSET 3;
```

Si on reprend l'exemple précédent, mon résultat sera les **5 meilleurs salaires de ma table en excluant les 3 premiers**, donc, du 4ème au 8ème.

Exercice: Utilisation de **ORDER BY**, **LIMIT** et **OFFSET**

Toujours au sein de notre table Users, construisez les requêtes suivantes:

1. Sélectionnez les cinq utilisateurs les plus âgés de la table "Users", triés par ordre décroissant d'âge.
2. Affichez les enregistrements 6 à 10 triés par ordre alphabétique du prénom.
3. Sélectionnez les trois utilisateurs ayant les salaires les plus élevés de la table "Users", triés par ordre décroissant de salaire.

CORRECTION: Utilisation de ORDER BY, LIMIT et OFFSET

```
SELECT * FROM Users  
ORDER BY age DESC  
LIMIT 5;
```

```
SELECT * FROM Users  
ORDER BY first_name  
LIMIT 5 OFFSET 5;
```

```
SELECT * FROM Users  
ORDER BY salary DESC  
LIMIT 3;
```

Recherche de texte

LIKE

LIKE

Jusqu'à présent, nous avons filtrés en utilisant des valeurs exactes, notamment grâce à nos opérateurs de comparaison. Mais il est également possible de filtrer en se basant sur des **patterns de texte**, c'est l'utilité de LIKE.

```
SELECT *  
FROM Users  
WHERE birth_location LIKE 'P%';
```

Cette requête cherchera tous les utilisateurs dont la ville de naissance commencent par la lettre P, c'est le rôle du %, il s'agit d'un **caractère générique**

Caractères génériques

En MySQL, vous pouvez utiliser deux caractères génériques avec l'opérateur LIKE pour effectuer des correspondances de motifs plus flexibles lors de la recherche de chaînes de caractères :

- **%** (Pourcentage) : Représente n'importe quelle séquence de caractères, y compris aucune séquence. Par exemple, si vous utilisez `LIKE 'a%'`, cela renverra toutes les valeurs qui commencent par la lettre 'a'.
- **_** (Underscore) : Représente exactement un caractère. Par exemple, si vous utilisez `LIKE '_o%'`, cela renverra toutes les valeurs où la deuxième lettre est 'o'.

Quelques exemples de caractères génériques

- `LIKE 'a%'` : Recherche de toutes les valeurs commençant par 'a'.
- `LIKE '%o'` : Recherche de toutes les valeurs se terminant par 'o'.
- `LIKE '%or%'` : Recherche de toutes les valeurs contenant 'or' n'importe où à l'intérieur du mot.
- `LIKE '_at%'` : Recherche de toutes les valeurs où la deuxième lettre est 'a' et la troisième lettre est 't'.
- `LIKE 'a%e'` : Recherche de toutes les valeurs commençant par 'a' et se terminant par 'e'.

Exercice: Utilisation du LIKE

Dans notre table USERS, construisez les requêtes suivantes:

1. Sélectionnez les utilisateurs ayant un prénom qui commence par "D".
2. Trouvez les utilisateurs dont le nom de famille se termine par "son".
3. Identifiez les utilisateurs dont le prénom contient exactement 5 caractères.
4. Sélectionnez les utilisateurs ayant "Doctor" dans leur métier.
(caractère générique obligatoire)

CORRECTION: Utilisation du LIKE

```
SELECT *  
FROM Users  
WHERE first_name LIKE 'D%';
```

```
SELECT *  
FROM Users  
WHERE last_name LIKE '%son';
```

```
SELECT *  
FROM Users  
WHERE first_name LIKE '_____' ;
```

```
SELECT *  
FROM Users  
WHERE job LIKE '%Doctor%';
```

Recherche de valeurs IS NULL

La notion de NULL en SQL

NULL représente **une absence de valeur dans une colonne**.

Cela signifie que la valeur est indéfinie ou inconnue. Ce n'est pas la même chose qu'une chaîne vide (') ou le chiffre zéro (0) qui sont bien des valeurs réelles.

Une valeur NULL est utilisée pour marquer des données manquantes ou non renseignées.

product_id	product_name	description	price	stock
1	Laptop	Portable computer	1200.00	10
2	Smartphone	NULL	699.99	25
3	Desk Chair	Comfortable office chair	NULL	15

Une liste de produits avec certaines descriptions ou prix manquants

La clause IS NULL

La clause IS NULL permet de **vérifier si une colonne contient une valeur NULL.**

Syntaxe:

```
SELECT colonne1, colonne2, ...  
FROM table  
WHERE colonne IS NULL;
```

Exemple:

```
SELECT first_name, last_name  
FROM users  
WHERE email IS NULL;
```

Retourne prénoms et noms des users dont **la colonne email est vide.**

La clause IS NOT NULL

Rôle : La clause IS NOT NULL permet de vérifier si une colonne contient une valeur non nulle, c'est-à-dire qu'elle a été renseignée.

Exemple:

```
SELECT first_name, last_name  
FROM users  
WHERE email IS NOT NULL;
```

Résultat : Cette requête renvoie les prénoms et noms des employés dont l'adresse e-mail est renseignée.

Opérateurs arithmétiques

Les opérateurs arithmétiques

En plus des **opérateurs de comparaison** (`<`, `>`, `!=`) et des **opérateurs logiques** (`AND`, `OR`, `NOT`). On peut également appliquer les opérateurs arithmétiques en SQL.

On les trouve dans **les projections** (traitement des colonnes) et dans **les sélections** (traitement des enregistrements.)

- Additions: `+`
- Soustractions: `-`
- Multiplications: `*`
- Divisions: `/`

Exemple d'utilisation dans une projection

Pour calculer le coût total des articles en stock (prix * quantité) :

```
SELECT product_name, price * stock AS total_stock_value  
FROM Products;
```

Cette requête affichera le nom du produit et la valeur totale de son stock.

Une nouvelle colonne est donc créé temporairement pour afficher le résultat de notre requête, mais ne elle vient pas s'insérer ou modifier notre base de données. (Le DQL ne permet pas cela)

Exemple d'utilisation dans une sélection

Pour filtrer les produits dont le prix par unité est inférieur à 50 après une réduction de moitié sur le prix

```
SELECT product_name  
FROM Products  
WHERE (price / 2) < 50;
```

Cette requête retournera les produits dont le prix est inférieur à 50 après une réduction de 50%.

Création d'alias de colonnes

Lorsque **vous effectuez une projection avec des calculs** ou des expressions, la colonne résultante **n'a pas de nom explicite**.

Vous pouvez **utiliser AS** pour donner **un nom à cette colonne** dans le résultat. On nomme cela un **alias**.

```
SELECT product_name, price * quantity AS total_stock_value  
FROM Products;
```

Avec `AS total_stock_value`, la colonne résultante est renommée en `total_stock_value` dans le résultat de la requête.

Fonctions supplémentaires

Fonctions supplémentaires

Les fonctions supplémentaires des SGBD étendent les capacités des requêtes SQL en offrant des outils avancés pour manipuler et transformer les données.

Attention, ces fonctions sont donc parfois différentes dans chaque SGBD, même si la plupart sont généralement communes.

Elles permettent des **opérations avancées** sur les deux types de data que nous avons vu jusqu'à maintenant: **les chaînes de caractères (")** et **les nombres**.

Voici quelques une des fonctions de **MySQL**:

Fonctions arithmétiques supplémentaires

- **Modulo** : Renvoie le reste de la division entière entre deux nombres.
- **Racine carrée** : Calcule la racine carrée d'un nombre.
- **Puissance** : Élève un nombre à la puissance d'un autre.
- **Arrondi** : Arrondit un nombre à un certain nombre de décimales.
- **Valeur plancher (floor)** : Renvoie le plus grand entier inférieur ou égal à un nombre.
- **Valeur plafond (ceiling)** : Renvoie le plus petit entier supérieur ou égal à un nombre.

Fonctions arithmétiques (Syntaxe SQL)

Voici les syntaxes SQL pour chaque fonction arithmétique supplémentaire :

- **Modulo** : `MOD(number1, number2)`
- **Racine carrée** : `SQRT(number)`
- **Puissance** : `POW(base, exponent)` ou `POWER(base, exponent)`
- **Arrondi** : `ROUND(number, decimals)`
- **Valeur plancher (floor)** : `FLOOR(number)`
- **Valeur plafond (ceiling)** : `CEIL(number)` ou `CEILING(number)`

Exemple avec la table products

```
SELECT product_name,  
       price,  
       stock,  
       price * stock AS total_value,  
       FLOOR(price * stock) AS floored_total_value  
FROM Products;
```

- **price * stock** calcule la valeur totale du stock pour chaque produit.
- **FLOOR(price * stock)** arrondit cette valeur totale à l'entier inférieur.

Fonctions de chaînes de caractères

- **UPPER** : Convertit tous les caractères d'une chaîne en majuscules.
- **LOWER** : Convertit tous les caractères d'une chaîne en minuscules.
- **LENGTH** : Renvoie le nombre de caractères dans une chaîne.
- **LPAD** : Ajoute des caractères à gauche d'une chaîne jusqu'à atteindre une longueur spécifiée.
- **RPAD** : Ajoute des caractères à droite d'une chaîne jusqu'à atteindre une longueur spécifiée.
- **SUBSTRING** : Extrait une partie d'une chaîne à partir d'une position donnée.

Fonctions de chaînes de caractères (Syntaxe SQL)

- **UPPER** : `UPPER(string)`
- **LOWER** : `LOWER(string)`
- **LENGTH** : `LENGTH(string)`
- **LPAD** : `LPAD(string, total_length, pad_string)`
- **RPAD** : `RPAD(string, total_length, pad_string)`
- **SUBSTRING** : `SUBSTRING(string, start_position, length)`

Exemple concret avec la table **Products**

Imaginons que nous souhaitons afficher les noms de produits en majuscules, leur longueur, et une version remplie de caractères ***** à gauche pour atteindre une longueur de 15 caractères.

```
SELECT product_name,  
       UPPER(product_name) AS upper_name,  
       LENGTH(product_name) AS name_length,  
       LPAD(product_name, 15, '*') AS left_padded_name  
FROM Products;
```

- **upper_name** : LAPTOP
- **name_length** : 6
- **left_padded_name** : *****Laptop

