Formation SQL, le DQL

m2iformation.fr

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.
Une base de données est donc fourni avec le cours pour permettre la manipulation du langage, elle se prénomme **dql_bdd.sql**
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;
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

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

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

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 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.

```sql
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
```

Les opérateurs 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.

```sql
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.

```sql
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.

```sql
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 si elles sont différentes. Ils peuvent cependant être parfois utilisés à des fins similaires.

Exemple: Le NOT

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

```
SELECT first_name, last_name, location
FROM Users
WHERE NOT 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, location, job
FROM Users
WHERE 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

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

DISTINCT

La clause DISTINCT

La clause **DISTINCT** est utilisée pour 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 que nous verrons par la suite, elle est compatible avec WHERE, voici sa **syntaxe**:

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

Exemple

Je cherche à récupérer tous les jobs uniques qui existent à la localisation New York:

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

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**.
- **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, location
FROM Users
WHERE location IN ('Paris', 'London');
```

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

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

Cette requête est similaire à:

```
SELECT first_name, last_name, location
FROM Users
WHERE location != 'Paris' OR 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:

```sql
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;
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

ORDER BY

ORDER BY

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)

```sql
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.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

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

```sql
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;
```

Les fonctions d'agrégations

Les fonctions d'agrégations

Les **fonctions d'agrégations** en SQL nous permettent de réaliser des calculs sur les valeurs de notre base de données et d'en **retourner un résultat**.

Elles sont nombreuses et dépendent du SGBD, nous verrons ici les plus utilisées:

MIN() - Retourne la valeur minimale d'une colonne.

MAX() - Retourne la valeur maximale d'une colonne.

COUNT() - Compte le nombre de lignes d'une colonne

SUM() - Calcule la somme des valeurs d'une colonne

AVG() - Calcule la valeur moyenne d'une colonne

MIN()/MAX()

Les fonctions MIN() et MAX() sont les fonctions d'agrégations les plus faciles à utiliser, comme leur nom l'indique, elles retournent simplement la plus grande ou la plus petite valeur d'une colonne:

```
SELECT MIN(age)
FROM users;

SELECT MAX(age)
FROM users;
```

Les fonctions d'agrégations **s'appliquent sur les colonnes**, donc directement après le SELECT. Le filtre du WHERE s'applique **AVANT** l'agrégation

LES ALIAS

Comme nos fonctions d'agrégations nous retournent des résultats externes à notre BDD, il peut être bon de les renommer pour savoir à quoi ils correspondent. C'est là qu'entre en jeu les **ALIAS**.
Ils permettent de **RENOMMER** le nom d'un résultat.
Les alias fonctionnent grâce à la clause **AS**:

```
SELECT MIN(age) AS developer_min_age
FROM users
WHERE job = "Developer";
```

Ici j'ai renommé mon résultat en "developer_min_age"

Particularités

Les fonctions MIN() et MAX() fonctionnent également sur **les données en forme de textes** (String), dans ce cas la valeur retournée est basée sur **l'ordre alphabétique**:

```
SELECT MAX(last_name) AS last_name
FROM users;
```

Ici, je récupérerai le nom de famille qui est le dernier par ordre alphabétique.

COUNT()

La fonction COUNT() est elle aussi assez simple d'utilisation, elle renvoie **le nombre de lignes** qui correspondent à notre critère.

```
SELECT COUNT(*) AS total_users
FROM users;
```

Ici, cette requête me permet de savoir le **nombre total d'utilisateurs** que comporte ma table.

Note: La fonction COUNT() ne compte pas les lignes dont la valeur est NULL, attention donc à votre choix de colonne.

SUM()

La fonction d'agrégation **SUM()** nous permet de calculer **la somme des différentes lignes d'une colonne**. Elle ne fonctionne donc qu'avec des nombres

```
SELECT SUM(salary) AS total_salary_without_devs
FROM users;
WHERE job != 'Developer';
```

Grâce à cette requête, je connais **le montant total des salaires** de tous mes utilisateurs en excluant celui des développeurs.

AVG()

La fonction d'agrégation **AVG()** fonctionne dans la même logique que SUM(). Mais elle retournera la moyenne de toutes les valeurs plutôt que la somme.

```
SELECT AVG(salary) AS average_salary_devs
FROM users;
WHERE job = 'Developer';
```

Cette requête me retournera la moyenne des salaires des développeurs.

Exercice: Utilisation des fonctions d'agrégations

Dans ma table Users, trouvez les requêtes suivantes **en utilisant à chaque fois des alias**:

1. Quel est le salaire minimum parmi tous les utilisateurs ?
2. Quel est l'âge maximum parmi les utilisateurs ayant le métier "Engineer" ?
3. Trouvez le salaire moyen des utilisateurs dont le métier est "Teacher".
4. Trouvez le montant total des salaires de tous les utilisateurs.

CORRECTION: Utilisation des fonctions d'agrégations

```sql
SELECT MIN(salary) AS MinSalary
FROM Users;

SELECT MAX(age) AS MaxAgeEngineer
FROM Users
WHERE job = 'Engineer';

SELECT AVG(salary) AS AvgSalaryTeacher
FROM Users
WHERE job = 'Teacher';

SELECT SUM(salary) AS TotalSalaries
FROM Users;
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

GROUP BY

GROUP BY

La clause GROUP BY en SQL est utilisée pour **regrouper les lignes de données en fonction des valeurs d'une ou plusieurs colonnes.**
Elle permet de créer des **groupes distincts de lignes** qui partagent des valeurs similaires dans les colonnes spécifiées.
La clause GROUP BY est **quasiment toujours** utilisée avec des **fonctions d'agrégation** (comme SUM(), AVG(), COUNT(), etc.) pour effectuer des calculs sur chaque groupe de données plutôt que sur l'ensemble complet de données.

Syntaxe

```
SELECT birth_location, SUM(salary)
FROM Users
GROUP BY birth_location;
```

Cet exemple sert à calculer la somme des salaires pour chaque lieu de naissance distinct dans la table "Users".

- `SELECT birth_location, SUM(salary)` : Cela indique que nous voulons sélectionner la colonne "birth_location" (lieu de naissance) et la somme de la colonne "salary" (salaire).
- `GROUP BY birth_location` : Cela signifie que les lignes avec le même lieu de naissance seront regroupées ensemble.

Exemple

```sql
SELECT birth_location, AVG(salary)
FROM Users
GROUP BY birth_location
ORDER BY AVG(salary) DESC;
```

Cette requête sert à calculer la moyenne des salaires pour chaque lieu de naissance distinct dans la table "Users" et à les trier par ordre décroissant en fonction de la moyenne des salaires.

`ORDER BY AVG(salary) DESC`: Cette clause ordonne les résultats en fonction de la moyenne des salaires, triée de manière décroissante (DESC). Cela signifie que les lieux de naissance avec les moyennes de salaires les plus élevées seront affichés en premier.

HAVING

HAVING

La clause HAVING est plus simple, elle agit de la même façon qu'un WHERE mais au sein d'un groupement généré par un GROUP BY. Elle doit donc être située après celui-ci:

```
SELECT birth_location, AVG(salary) AS average_salary
FROM Users
GROUP BY birth_location
HAVING AVG(salary) > 3000;
```

Ici, j'applique mon filtre en disant que je ne veux que les groupes de lieux dont le salaire moyen est supérieur à 3000.

Remarque

L'exemple précédent est différent de:

```sql
SELECT birth_location, SUM(salary) AS total_salary
FROM Users
WHERE salary > 3000
GROUP BY birth_location;
```

En effet, ici, j'applique le filtre avant de grouper par lieux de naissance et avant la somme de mes salaires. Je dois donc bien réfléchir si je veux filtrer avant ou après et utiliser WHERE et HAVING en conséquences (les deux sont bien sûr cumulables)

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.

```sql
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

```sql
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%';
```

Sous-requêtes (Subqueries)

Sous-requêtes

En SQL, il est également possible d'imbriquer les requêtes les unes dans les autres pour gagner en précision. On nomme ce concept **la sous-requête**.
Il n'y a pas de limites à cela mais il faut éviter d'entrer dans des niveaux de complexité trop haut pour garder de la lisibilité.

Syntaxe

```sql
SELECT first_name, last_name, salary
FROM Users
WHERE salary > (SELECT AVG(salary) FROM Users);
```

Dans cet exemple, la sous-requête `(SELECT AVG(salary) FROM Users)` calcule la moyenne des salaires de tous les utilisateurs. La requête principale sélectionne ensuite les utilisateurs ayant un salaire supérieur à cette moyenne

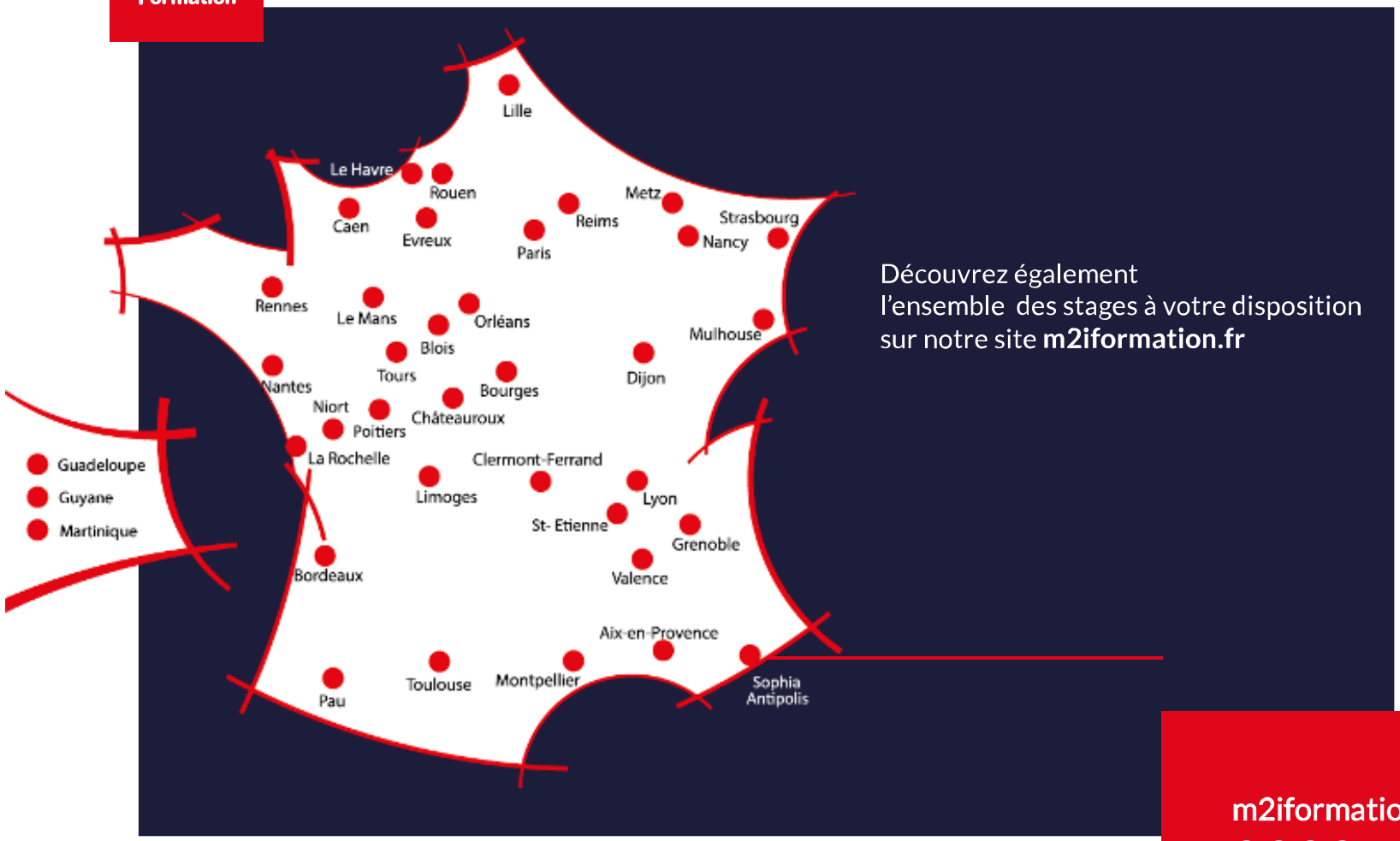
Autre exemple

```sql
SELECT first_name, last_name, salary
FROM Users
WHERE salary = (SELECT MAX(salary) FROM Users);
```

Dans cet exemple, la sous-requête `(SELECT MAX(salary) FROM Users)` calcule le salaire maximum parmi tous les utilisateurs. La requête principale sélectionne ensuite les utilisateurs ayant ce salaire maximum.

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr