Enseignant :	Louis RAYNAL	**Établissement :**	ICES
Contact :	l-raynal@ices.fr	**Année :**	2025-2026
Public :	L3 Maths et M2 Bio-Santé		
Ressources :	https://github.com/LouisRaynal/coursSQL		

Cours-TP 3 : Interrogation d'une base de données

Jusqu'à présent nous avons vu comment créer une base de données, créer/supprimer des tables et insérer/modifier/supprimer des données de tables. Nous allons maintenant voir comment interroger les données d'une base existante, c'est-à-dire récupérer des informations stockées dans ses tables. Pour cela on utilisera la commande `SELECT`.

Remarque

> Une **requête de sélection**, ou bien une **requête d'interrogation**, désignera une requête qui utilise la commande `SELECT` afin de récupérer des informations d'une base de données.

1. Interrogation de données avec `SELECT`

La commande `SELECT` est sans doute la plus compliquée du langage SQL, car elle est constituée de nombreux mots-clés appelés **clauses**. Les clauses permettent de définir les informations à récupérer lors d'une requête de sélection.

Dans le tableau ci-dessous vous trouverez les différentes clauses utilisables dans une requête de sélection, ainsi que l'utilité de chacune.

Clauses	Descriptions
`SELECT`	Détermine les colonnes à inclure dans le résultat de la requête d'interrogation
`FROM`	Identifie les tables depuis lesquelles récupérer les données
`WHERE`	Précise les lignes à conserver ou à exclure
`GROUP BY`	Groupe des lignes les unes avec les autres
`HAVING`	Précise les groupes à conserver ou à exclure
`ORDER BY`	Ordonne les lignes du résultat de la requête d'interrogation
`LIMIT`	Limite le nombre de lignes renvoyées dans le résultat de la requête
`OFFSET`	Ignore un certain nombre de premières lignes dans le résultat de la requête

La syntaxe d'une requête d'interrogation qui utiliserait toutes ces clauses est la suivante :

Code 1 – Syntaxe complète d'une requête d'interrogation

```sql
SELECT nom_cols
FROM nom_tables
WHERE condition_where
GROUP BY nom_cols_groupage
    HAVING condition_having
ORDER BY expression_ordre
LIMIT num_limit
    OFFSET num_offset
;
```

Avant de détailler chacune de ces clauses, présentons la base de données sur laquelle nous allons nous exercer.

Pratique

On s'intéresse à une base de données de films référencés sur le site IMDb (*the Internet Movie Database*) ainsi que leurs réalisateurs. Cette base se nomme `imdb` et est contenue dans le fichier `imdb.sqlite` (accessible via l'adresse `https://github.com/LouisRaynal/coursSQL` dossier `TP3`).

Remarque

Un fichier `.sqlite` est un format de fichier similaire au `.db`, qui stocke une base de données SQLite.

Pratique

Commencez par télécharger cette base de données et connectez-vous-y depuis SQLiteStudio.

La base `imdb` est composée de deux tables : `directors` et `movies`.

`directors` contient des informations sur des *réalisateurs*, stockées dans les colonnes :

- `name` - nom du réalisateur ;
- `id` - identifiant unique du réalisateur ;
- `gender` - genre du réalisateur (codé `0` pour non renseigné/autre, `1` pour femme, `2` pour homme).

`movies` stocke des informations sur des *films* dans les colonnes :

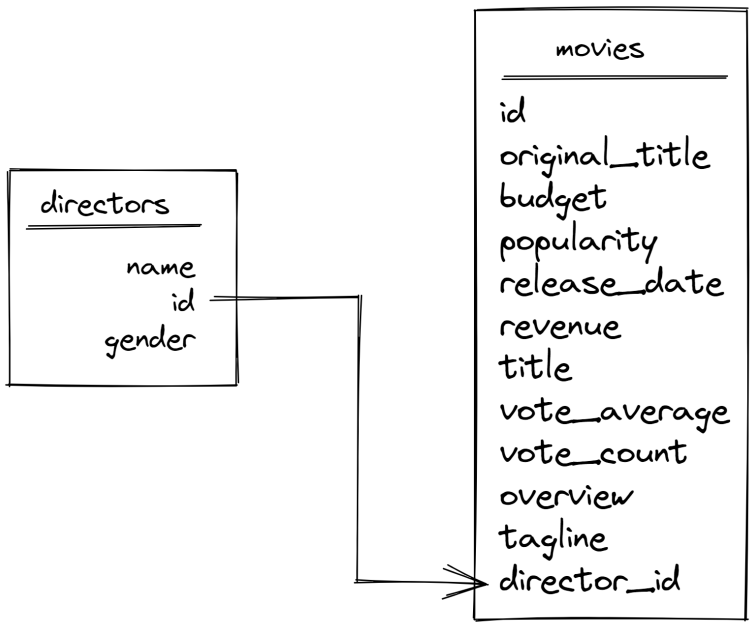
- `id` - identifiant unique du film ;
- `original_title` - titre original, dans la langue d'origine ;
- `budget` - budget en dollars ;
- `popularity` - popularité du film (mesure propre à IMDb) ;
- `release_date` - date de sortie du film ;
- `revenue` - recettes du film ;
- `title` - titre américain ;
- `vote_average` - note moyenne calculée d'après les votes des spectateurs ;
- `vote_count` - nombre de votes des spectateurs ;
- `overview` - synopsis du film ;
- `tagline` - sous-titre/slogan ;
- `director_id` - identifiant du réalisateur du film.

La colonne `id` de la table `directors` permet d'identifier de manière unique chaque ligne de cette table, c'est-à-dire chaque réalisateur. Il s'agit donc de la clé primaire de la table `directors`.

D'une manière similaire, la colonne `id` de la table `movies` est la clé primaire de cette table.

Étant donné que chaque *film* est réalisé par un *réalisateur*, il est naturel de retrouver un identifiant de réalisateur pour chaque ligne de la table `movies`, c'est-à-dire pour chaque film. L'identifiant du réalisateur `director_id` de la table `movies` est donc une clé étrangère pour cette table.

Nous pouvons maintenant dessiner le schéma relationnel correspondant à la base `imdb`, ce qui donne le schéma ci-dessous.



1.1. Clause SELECT

La clause SELECT permet de spécifier les colonnes que l'on souhaite récupérer dans le résultat d'une requête. Cette clause est donc suivie des noms de colonnes pour lesquelles nous voulons récupérer les informations. SELECT est normalement accompagnée de la clause FROM qui indique la ou les tables d'où proviennent les informations (pour le moment nous nous limitons à une table).

Code 2 – Syntaxe de base pour SELECT et FROM

```
SELECT nom_col_1, nom_col_2, nom_col_3
FROM nom_table
;
```

Pratique

Commençons par une requête interrogeant la table movies.

Code 3 – Exemple de sélection de toutes les colonnes de la table movies

```
SELECT *
FROM movies
;
```

Cette requête signifie : *sélectionne toutes les colonnes et toutes les lignes de la table* movies. Le symbole * signifie *toutes les colonnes*, ici qui proviennent de la table movies. L'absence de la clause WHERE ou LIMIT implique que toutes les lignes sont affichées comme résultat de la requête.

Nous pouvons obtenir le même résultat en explicitant le nom des colonnes.

Code 4 – Exemple de sélection de toutes les colonnes de la table `movies` (bis)

```
SELECT id, original_title, budget, popularity, release_date,
    revenue, title, vote_average, vote_count, overview, tagline, director_id
FROM movies
;
```

Il est aussi possible de récupérer uniquement certaines colonnes qui nous intéressent en spécifiant leurs noms. Par exemple pour récupérer les informations : titre de film, date de sortie, note moyenne et nombre de votes pour tous les films, on écrit la requête

Code 5 – Exemple de sélection de colonnes de la table `movies`

```
SELECT title, release_date, vote_average, vote_count
FROM movies
;
```

Remarque

L'ordre d'affichage des colonnes est toujours celui spécifié dans la requête.
Si * est utilisé, alors l'ordre est celui spécifié dans la structure de la table, lors de sa définition.

Colonnes personnalisées

En plus d'interroger des colonnes déjà existantes, il est possible d'ajouter au `SELECT` des **colonnes personnalisées** grâce à des expressions. Une expression est une notion développée un peu plus tard, mais en bref il s'agit d'une formule qui sera évaluée pour chaque ligne de la table interrogée afin d'obtenir un résultat.

Parmi des exemples de colonnes personnalisées, on trouvera :

- des chaînes de caractères, dates ou valeurs numériques. La valeur spécifiée sera répétée sur chaque ligne du résultat de la requête ;
- des opérations entre des colonnes ;
- l'évaluation de fonctions.

Pratique

La requête ci-dessous donne quelques illustrations de colonnes personnalisées.

Code 6 – Exemples de colonnes personnalisées

```
SELECT title,                   -- affiche le titre du film
    'Film',                     -- affiche 'Film' sur toutes les lignes
    vote_average * vote_count,  -- affiche le produit de deux colonnes
    ROUND(vote_average)         -- affiche l'arrondi de la note moyenne
FROM movies
;
```

Exercice : Écrire une requête interrogeant la table `movies` afin d'afficher

1. une colonne avec la valeur `42` sur toutes les lignes ;
2. une colonne qui ajoute 1 au nombre de votes de chaque film ;
3. une colonne avec le titre des films en majuscules, (utiliser la fonction `UPPER( )`).

Alias de colonne

Lors de l'affichage du résultat d'une requête de sélection, SQL génère automatiquement un nom pour chaque colonne de résultat : il s'agit de l'expression listée après la clause `SELECT`, qui a donné lieu à la colonne correspondante. Ce comportement n'est pas très pratique lorsque l'expression évaluée est très longue, comme des calculs sur des colonnes.

Il est possible de définir un nom de colonne personnalisé à l'affichage du résultat, appelé **alias de colonne**. Pour ce faire, il suffit de faire suivre l'expression évaluée par le mot-clé `AS` puis l'alias de colonne.

Pratique

Reprenons la requête précédente, mais cette fois-ci en utilisant des alias de colonne.

Code 7 – Exemple d'utilisation des alias de colonne

```sql
SELECT title,
    'Film' AS Catégorie,
    vote_average * vote_count AS "Somme des notes",
    ROUND(vote_average) AS "Note arrondie"
FROM movies
;
```

Remarque

`AS` n'est pas obligatoire pour spécifier un alias de colonne. Vous pouvez directement écrire l'alias de colonne à la suite de l'expression correspondante.

Notez aussi qu'il est possible d'utiliser un alias en plusieurs mots. Il faut l'écrire entre `" "`. Veillez simplement à utiliser le mot-clé `AS` dans ce cas-là.

Suppression des doublons de lignes

Par défaut, SQL affiche comme résultat d'une requête de sélection toutes les lignes demandées, même si des lignes figurent en double. Il est possible de supprimer les doublons de lignes grâce au mot-clé `DISTINCT`, à utiliser juste après `SELECT`. Lorsque utilisé, chaque ligne du résultat de la table n'est affichée qu'une fois.

Pratique

Si l'on sélectionne uniquement la colonne `director_id` de la table `movies`, il y aura des doublons car plusieurs films ont été réalisés par la même personne. `DISTINCT` nous permet de supprimer ces doublons.

Code 8 – Exemple d'utilisation de `DISTINCT`

```sql
-- Requête affichant 4773 lignes
SELECT director_id
FROM movies
```

```sql
;

-- Requête affichant 2349 lignes
SELECT DISTINCT director_id
FROM movies
;
```

Remarque

En SQLite il est possible d'utiliser `SELECT` sans la clause `FROM`, mais l'intérêt est très limité. Cela peut permettre d'expérimenter certaines expressions.

Code 9 – Exemple d'utilisation de `SELECT` sans `FROM`

```sql
SELECT CURRENT_DATE AS Date,
     CURRENT_TIME AS Heure,
     124+231 AS Résultat,
     ROUND(5.574)
;
```

1.2. Clause `FROM`

`FROM` permet de spécifier la ou les tables utilisées par une requête. Jusqu'à présent nous nous sommes limités à une seule table après la clause `FROM`.

Pour interroger les données de plusieurs tables dans une même requête, il est nécessaire de préciser le lien qui existe entre chaque paire de tables. C'est ce que l'on appelle faire une **jointure** entre des tables. (Le prochain cours-TP est dédié aux jointures, mais en voici un bref aperçu en pratique).

Pratique

Nous avons vu qu'il existait un lien entre les tables `directors` et `movies` grâce aux identifiants de réalisateur, qui se retrouvent à la fois dans `directors` (colonne `id` est une clé primaire pour cette table) et dans `movies` (colonne `director_id` est une clé étrangère pour cette table).

Afin d'interroger des colonnes de ces deux tables, il faut spécifier dans la clause `FROM` que l'on souhaite joindre la table `directors` à la table `movies` via les **valeurs communes** de `id` et `director_id`. Pour cela on utilise les mots-clés `INNER JOIN ... ON` de la manière suivante :

Code 10 – Exemple de jointure entre les tables `directors` et `movies`

```sql
SELECT directors.name, movies.title
FROM directors                          -- de la table directors
INNER JOIN movies                       -- jointe avec la table movies
    ON directors.id = movies.director_id
    -- sur condition d'égalité entre id de directors et director_id de movies
;
```

Alias de table

Lorsque plusieurs tables interviennent dans une requête de sélection, il est possible qu'un même nom de colonne soit utilisé dans deux tables. C'est par exemple le cas de la colonne nommée `id` dans la base `imdb`, qui se retrouve dans les tables `movies` et `directors`. Il faut alors un moyen d'identifier à quelle table une colonne fait référence.

Pour cela il y a deux possibilités :

- faire précéder le nom de la colonne, par la table de provenance : `nom_table.nom_col` ;
- définir un **alias de table** et l'utiliser à la place du nom de la table : `nom_alias.nom_col`

Un alias de table se définit dans la clause `FROM`, après le nom de la table, en utilisant le mot-clé `AS` suivi du nom de l'alias choisi.

Remarque

Il est commun d'utiliser des alias de table très compacts, afin de rendre les requêtes plus lisibles.

Comme pour un alias de colonne, le mot-clé `AS` n'est pas obligatoire, et un alias de table peut contenir plusieurs mots.

Pratique

Voici à quoi ressemble la précédente requête en utilisant des alias de table :

Code 11 – Exemple de jointure entre les tables `directors` et `movies` avec des alias de table

```
SELECT d.name, m.title
FROM directors AS d
INNER JOIN movies AS m
    ON d.id = m.director_id
;
```

Les tables `directors` et `movies` ont respectivement reçu l'alias de table `d` et `m`. Une fois défini, un alias peut être utilisé dans les autres endroits de la requête, par exemple dans les expressions qui suivent la clause `SELECT`.

1.3. Clause `WHERE`

La clause `WHERE` peut être suivie d'une ou plusieurs **conditions**, séparées par les opérateurs `AND` et `OR`. Ces conditions permettent de préciser les lignes qui seront conservées dans le résultat de la requête.

• Lorsqu'une seule condition est spécifiée, alors seulement les lignes vérifiant cette condition sont récupérées.

Pratique

Par exemple

```
WHERE movies.vote_average > 7
```

permet de ne garder que les lignes de `movies` avec une note moyenne `vote_average` supérieure à 7.

• Lorsque plusieurs conditions sont spécifiées et sont séparées par l'opérateur `AND`, alors toutes ces conditions doivent être vérifiées par une ligne pour être incluse dans le résultat de la requête.

Pratique

Par exemple

```
WHERE movies.vote_average > 7 AND movies.vote_count > 1000
```

permet de ne garder que les films avec à la fois une note moyenne `vote_average` supérieure à 7 et avec plus de 1000 votes.

• Lorsque plusieurs conditions sont spécifiées et sont séparées par l'opérateur `OR`, alors au moins une condition doit être vérifiée par une ligne pour être incluse dans le résultat de la requête.

Pratique

Par exemple

```
WHERE movies.vote_average > 7 OR movies.vote_count > 1000
```

permet de ne garder que les lignes de `movies` avec une note moyenne `vote_average` supérieure à 7, ou bien avec plus de 1000 votes (ou bien vérifiant les deux conditions).

• Lorsque trois conditions ou plus sont utilisées, et qu'elles sont séparées à la fois par les opérateurs `AND` et `OR`, il vaut mieux **utiliser des parenthèses** pour faciliter la compréhension de ce que va faire la requête et sa lecture.

Pratique

Prenons la clause `WHERE` suivie des trois conditions suivantes :

```
WHERE (movies.vote_average > 7 OR movies.vote_count > 1000)
    AND movies.release_date < '2015-01-15'
```

Les lignes incluses devrons vérifier : (avoir une note moyenne supérieure à 7 OU un nombre de votes supérieur à 1000) ET une date de sortie inférieure au 15 janvier 2015.

• Il est aussi possible d'ajouter l'opérateur de **négation** `NOT` devant une condition, afin de conserver les lignes ne vérifiant pas cette condition.

Pratique

Par exemple

```
WHERE NOT movies.vote_average > 7
```

permet de garder les lignes ne vérifiant pas la condition "avoir une note moyenne supérieure à 7". Cela est équivalent à

```
WHERE movies.vote_average <= 7
```

Attention

Lorsqu'il y a plusieurs conditions il est très facile de s'y perdre en utilisant un `NOT`.
Prenons par exemple

```
WHERE NOT (movies.vote_average > 7 OR movies.vote_count > 1000)
    AND movies.release_date < '2015-01-15'
```

Cela équivaut à

```
WHERE (movies.vote_average <= 7 AND movies.vote_count <= 1000)
    AND movies.release_date < '2015-01-15'
```

qui est plus simple à comprendre.

Souvenez-vous qu'utiliser `NOT` équivaut à prendre la condition inverse, et à transformer les `OR` en `AND`, et les `AND` en `OR`.

Construction d'une condition

Maintenant que nous savons comment plusieurs conditions sont prises en compte en SQL, nous allons nous intéresser à comment construire une condition.

Une **condition** est une **expression qui s'évalue à une valeur booléenne** (1 ou 0) lorsque utilisée dans une requête. On obtient 1 lorsque la condition est vraie, 0 sinon.

Une **expression** est une **formule composée d'un ou plusieurs éléments** syntaxiques, et **qui s'évalue à une valeur** (ou groupe de valeurs) lorsque utilisée dans une requête.

Une expression va habituellement s'évaluer pour chaque ligne de la table (ou les tables) de la requête dans laquelle elle se trouve. Par exemple, lorsque utilisée après la clause `WHERE`, l'expression `movies.vote_average <= 7` va s'évaluer à 1 pour les lignes vérifiant cette expression, et s'évaluer à 0 pour les lignes ne la vérifiant pas. (Notez que par définition, cette expression est ici une condition).

De nombreux éléments peuvent être utilisés pour constituer une expression. La table ci-dessous liste les principaux que nous rencontrerons.

Élément	Exemple
Nom de colonne	`movies.title`
Chaîne de caractères	`'Film'`
Valeur numérique	`42`
Donnée temporelle	`'2015-01-15'`
Opérateur de comparaison	`=`, `!=`, `<>`, `>`, `>=`, `<`, `<=`, `LIKE`, `IN`, `BETWEEN`, `IS`
Opérateur arithmétique	`+`, `-`, `*`, `/`
Opérateur logique	`NOT`, `AND`, `OR`
Opérateur de concaténation	`||`
Parenthèses	`( )`
Fonction SQL	`LOWER( )`
Liste d'expressions	`('Learning', ' ', 'SQL')`

Remarque

Certains de ces éléments peuvent être considérés individuellement comme une expression. C'est le cas d'une valeur numérique, d'une chaîne de caractères ou d'une donnée temporelle lorsque utilisées après une clause `SELECT`. Elles seront évaluées à leur valeur pour toutes les lignes du résultat de la requête. C'est aussi le cas d'un nom de colonne, qui sera évalué à la valeur de la colonne pour chaque ligne du résultat de la requête.

Certains éléments doivent obligatoirement être utilisés avec d'autres éléments pour former une expression. C'est le cas des opérateurs. Juste l'opérateur `+` n'est pas une expression, mais `1 + 4` est une expression qui s'évalue à `5`.

Ces éléments peuvent être combinés à souhait afin d'obtenir des expressions, dont des conditions. Nous allons maintenant explorer les différents types de conditions qui peuvent être créées.

Conditions d'égalité et d'inégalité

Une **condition d'égalité** prend la forme `expression_1 = expression_2` et permet ainsi de comparer deux expressions entre elles. Une **condition d'inégalité** utilise l'opérateur `<>` (ou `!=`), et prend la forme `expression_1 <> expression_2`.

La première expression est le plus souvent un nom de colonne et la deuxième un nombre ou une chaîne de caractères.

Pratique

Un exemple de condition d'égalité est

```
movies.title = 'Interstellar'
```

Un exemple de condition d'inégalité est

```
movies.title <> 'Interstellar'
```

Créer une condition d'égalité avec une valeur `NULL` permet d'identifier des informations non renseignées (c'est-à-dire avec un vide dans la table). Pour ce faire il faut utiliser l'opérateur `IS` suivi de `NULL`.

Pratique

La condition

```
movies.overview IS NULL
```

sera évaluée à vraie pour les lignes où le synopsis du film n'est pas renseigné.

Une condition d'inégalité avec une valeur `NULL` nécessite l'ajout de l'opérateur `NOT`.

Pratique

La condition

```
movies.overview IS NOT NULL
```

permet de s'assurer que le synopsis d'un film est bien renseigné.

Condition d'étendue

Il peut être utile de vérifier qu'une expression tombe dans un intervalle de valeurs. C'est notamment le cas lorsque l'on travaille avec des données numériques ou temporelles. On parle alors de **conditions d'étendue**.

• La première manière de faire consiste à utiliser les opérateurs `>`, `>=`, `<` ou `<=` afin de cibler l'intervalle de valeurs qui nous intéresse au moyen de deux conditions.

Pratique

Par exemple

```
movies.release_date >= '2015-01-01' AND movies.release_date <= '2015-12-31'
```

permet de cibler les films sortis en 2015.

• La deuxième manière consiste à utiliser l'opérateur `BETWEEN`, afin d'éviter l'utilisation de deux conditions. Il faut alors spécifier la borne haute puis basse de l'intervalle cible.

Pratique

Dans le même but que précédemment, nous pouvons plutôt écrire

```
movies.release_date BETWEEN '2015-01-01' AND '2015-12-31'
```

Attention

Les bornes de `BETWEEN` sont **inclusives**, c'est-à-dire que `BETWEEN` peut remplacer une condition avec l'opérateur `>=` puis une avec `<=` (pas `>` puis `<`).

Remarque

Il est aussi possible de construire des conditions d'étendue sur des chaînes de caractères, mais cet usage est plus rare.

Condition d'appartenance

Il est possible d'établir des **conditions d'appartenance** à un ensemble de valeurs. Il existe différentes manières de faire.

• Une possibilité est d'utiliser plusieurs conditions d'égalité séparées par des `OR`.

Pratique

Par exemple

```
movies.title = 'Up' OR movies.title = 'Interstellar' OR movies.title = 'Avatar'
```

mais cela devient rapidement illisible.

• Une deuxième possibilité est d'utiliser l'opérateur `IN` associé d'une liste de valeurs d'intérêt `(val_1, val_2, ... , val_k)`.

Pratique

On obtient ainsi la version plus compacte

```
movies.title IN ('Up', 'Interstellar', 'Avatar')
```

L'opérateur `IN` peut être précédé de `NOT`, afin de s'assurer de l'absence de valeurs parmi la liste spécifiée.

Pratique

Par exemple

```
movies.title NOT IN ('Up', 'Interstellar', 'Avatar')
```

Condition de correspondance de caractères

Jusqu'à présent nous avons uniquement vu des conditions qui identifiaient exactement une chaîne de caractères. On veut parfois s'assurer de la **correspondance partielle** d'une chaîne de caractères.
Par exemple, trouver tous les films qui commencent par `'Harry Potter'`, sans avoir à écrire le nom exact de chacun des films !

C'est ici qu'interviennent l'opérateur `LIKE` et les **caractères de remplacement** `_` et `%` .

- Le caractère de remplacement `_` va remplacer exactement un caractère dans une chaîne de caractères.

Pratique

Par exemple la chaîne `'_p_'` correspondra à toutes les chaînes avec exactement trois caractères dont un `p` en deuxième position.

- Le caractère de remplacement `%` va remplacer n'importe quel nombre de caractères dans une chaîne (y compris zéro caractère).

Pratique

Par exemples `'Po%'` correspondra à toutes les chaînes commençant par `Po`, `'%Po'` correspondra à toutes les chaînes se terminant par `Po`, alors que `'%Po%'` correspondra à toutes les chaînes contenant `Po`.

Une expression de correspondance de caractères doit toujours être utilisée après l'opérateur `LIKE`.

Pratique

La requête ci-dessous affiche tous les titres de films commençant par `'Harry Potter'` dans la base.

Code 12 – Exemple de condition avec correspondance de caractères `%`

```
1 SELECT movies.title
2 FROM movies
3 WHERE movies.title LIKE 'Harry Potter%'
4 ;
```

La requête ci-dessous affiche tous les titres de films composés de trois lettres et commençant par `M`.

Code 13 – Exemple de condition avec correspondance de caractères _

```
SELECT movies.title
FROM movies
WHERE movies.title LIKE 'M__'
;
```

Remarque

Selon la variante du langage SQL, la correspondance de caractères est sensible à la casse. Par défaut ce n'est le cas de SQLite. Vous pouvez si vous le souhaitez utiliser la commande

```
PRAGMA case_sensitive_like = true;
```

pour activer la sensibilité à la casse de l'opérateur `LIKE`. (C'est une commande propre à SQLite).

Astuce

Pour éviter d'avoir à gérer la sensibilité à la casse dans les paramètres, il est commun de travailler avec les fonctions `UPPER( )` ou `LOWER( )`, qui permettent respectivement de mettre une chaîne de caractères en majuscules ou en minuscules. Par exemple :

```
SELECT movies.title
FROM movies
WHERE LOWER(movies.title) LIKE 'harry potter%' -- vous n'aurez qu'à  utiliser
    des chaînes toujours en minuscules si vous utilisez LOWER( )
;
```

Pratique

Exercices :

1. En utilisant une correspondance de caractères, affichez tous les titres de films dont le nombre de caractères est exactement égal à 4.
2. En utilisant une correspondance de caractères, affichez tous les titres de films dont le nombre de caractères est 4 ou plus.
3. En utilisant une correspondance de caractères, affichez tous les titres de films dont le nombre de caractères est 3 ou moins.
4. En utilisant deux correspondances de caractères, affichez tous les titres de films dont le nombre de caractères est compris entre 2 et 3.
5. Reprenez les questions 1. à 4., mais en utilisant la fonction `LENGTH( )` qui permet de calculer la longueur d'une chaîne de caractères.
6. Affichez les informations sur le film dont la note moyenne est comprise entre 7 et 9, sorti entre le 13/01/2008 et le 17/03/2009, et avec `'Earth'` dans son synopsis.

1.4. Les clauses `GROUP BY` et `HAVING`

Il est parfois nécessaire d'effectuer des manipulations de données préalables avant d'afficher le résultat d'une requête. C'est le but de la clause `GROUP BY`.

La clause `GROUP BY` permet de **regrouper des lignes** selon les valeurs des colonnes listées après cette clause. `GROUP BY` est utilisée en association avec des **fonctions d'agrégation** qui précisent les opérations à faire sur les lignes d'un même groupe.

Un cours-TP est dédié à la clause `GROUP BY` et aux fonctions d'agrégation, mais voici un aperçu de son utilisation.

Pratique

Disons que vous voulez savoir combien il y a de réalisateurs par genre (codé 0, 1 ou 2 dans la table `directors`). Il serait fastidieux de lister tous les réalisateurs et leur genre, puis de compter à la main combien il y en a dans chaque catégorie.

On voudrait plutôt **grouper les lignes par genre**, et pour chaque groupe formé, **compter combien il y a de lignes**. Voici à quoi ressemblerait une telle requête :

Code 14 – Exemple d'utilisation de `GROUP BY`

```
SELECT gender, COUNT(*)
FROM directors
GROUP BY gender
;
```

`GROUP BY gender` indique que l'on souhaite grouper les lignes par valeur de la colonne `gender` (0, 1 ou 2). La fonction d'agrégation `COUNT( )` avec `*`, indique que l'on veut compter le nombre de lignes dans chaque groupe.

Remarque

Il existe de nombreuses autres fonctions d'agrégation permettant de faire des opérations sur un groupe de lignes. On trouve par exemple `MIN( )` pour calculer le minimum, `MAX( )` pour le maximum, `AVG( )` pour la moyenne, `SUM( )` pour la somme.

La clause `GROUP BY` peut être accompagnée de la clause `HAVING` afin de filtrer des groupes de lignes, de la même manière qu'un `WHERE` grâce à des conditions.

Pratique

Par exemple pour exclure le groupe de genre 0 (non renseigné/autre)

Code 15 – Exemple d'utilisation de `GROUP BY` et `HAVING`

```
SELECT gender, COUNT(*)
FROM directors
GROUP BY gender
    HAVING gender <> 0
;
```

Exercice : Modifiez la requête ci-dessus pour compter le nombre d'identifiants de réalisateur par genre, et ne garder que le groupe de genre égal à 1.

1.5. Clause `ORDER BY`

La clause `ORDER BY` permet d'ordonner les lignes résultant d'une requête de sélection, en se basant sur les valeurs d'une ou plusieurs colonnes ou expressions listées après cette clause. L'un des mots-clés `ASC` ou `DESC` peut être ajouté après chaque colonne/expression spécifiée, pour obtenir un tri par ordre **croissant** ou **descroissant** respectivement. En l'absence d'un de ces mots-clés, `ASC` est utilisé par défaut.

Pratique

La requête ci-dessous permet d'afficher tous les titres de films, ainsi que la note moyenne de chaque film.

Code 16 – Exemple de non-utilisation de `ORDER BY`

```
SELECT title, vote_average
FROM movies
;
```

Si maintenant vous voulez trier le résultat de la requête dans l'**ordre croissant** de la note moyenne

Code 17 – Exemple d'utilisation de `ORDER BY` par ordre croissant

```
SELECT title, vote_average
FROM movies
ORDER BY vote_average -- ASC est l'ordre par défaut
;
```

Pour un ordre décroissant de `vote_average` il faut préciser `DESC`.

Code 18 – Exemple d'utilisation de `ORDER BY` par odre décroissant

```
SELECT title, vote_average
FROM movies
ORDER BY vote_average DESC
;
```

Après `ORDER BY` plusieurs colonnes peuvent être spécifiées. Afin de trier d'abord par note moyenne décroissante, puis par ordre alphabétique croissant de titre en cas d'égalité de note, écrire

Code 19 – Exemple d'utilisation de `ORDER BY` avec plusieurs colonnes

```
SELECT title, vote_average
FROM movies
ORDER BY vote_average DESC, title ASC
;
```

Enfin, au lieu d'indiquer un nom de colonne ou une expression pour le tri, il est possible d'utiliser **le numéro de colonne/expression** comme indiqué après la clause `SELECT`. Cela est particulièrement utile pour les expressions sans alias.

Pratique

La précédente requête peut être transformée afin d'effectuer un tri selon les numéros de colonnes, de la manière suivante :

Code 20 – Exemple d'utilisation de `ORDER BY` par numéro

```
1 SELECT title, vote_average
2 FROM movies
3 ORDER BY 2 DESC, 1 ASC
4 ;
```

1.6. Clauses `LIMIT` et `OFFSET`

La clause `LIMIT` permet d'établir le **nombre maximum `m` de lignes à afficher** dans le résultat d'une requête de sélection.

```
LIMIT 10          -- affiche les 10 premières lignes
```

La clause `OFFSET` peut être utilisée à la suite de `LIMIT`, et permet de définir **combien de lignes doivent être sautées** avant de commencer à afficher les `m` lignes définies par `LIMIT`.

```
LIMIT 10 OFFSET 3 -- saute les 3 premières lignes, et renvoie les 10 suivantes, c'est-à-
    dire les lignes 4 à 13

LIMIT 3 OFFSET 20 -- saute les 20 premières lignes, et renvoie les 3 suivantes, c'est-à-
    dire les lignes 21 à 23
```

Pratique

Parmi tous les titres de films triés par ordre alphabétique croissant, la requête ci-dessous saute les 5 premiers, et affiche les 10 suivants.

Code 21 – Exemple d'utilisation de `LIMIT` et `OFFSET`

```
1 SELECT title
2 FROM movies
3 ORDER BY title ASC
4 LIMIT 10 OFFSET 5
5 ;
```

Remarque

En pratique vous voudrez d'abord utiliser `ORDER BY` avant d'utiliser `LIMIT ... OFFSET`, sinon il n'y a pas d'ordre bien défini pour le résultat de la requête et `LIMIT ... OFFSET` n'a plus trop de sens.

1.7. Ordre d'exécution des clauses

Dans une requête de sélection, **l'ordre d'exécution des clauses n'est pas l'ordre d'écriture**. Voici l'ordre dans lequel les clauses sont exécutées, et ce qui est réalisé à chaque étape :

1. `FROM` - désigne la ou les tables source de données, et les combine si besoin pour obtenir une seule grande **table de travail** sur laquelle les autres clauses vont venir s'appliquer ;
2. `WHERE` - filtre les lignes de la table de travail ;
3. `GROUP BY` - groupe des lignes de la table de travail, en se basant sur les valeurs des colonnes/expressions indiquées ;
4. `SELECT` - définit l'ensemble des colonnes à afficher (et applique les fonctions d'agrégation si besoin)
5. `HAVING` - filtre des groupes de lignes de la table (si `GROUP BY` a été utilisée)
6. `DISTINCT` - élimine les lignes dupliquées ;
7. `ORDER BY` - ordonne les lignes ;
8. `OFFSET` - saute un certain nombre de premières lignes (si `LIMIT` a été utilisée) ;
9. `LIMIT` - ne garde qu'un certain nombre de lignes dans le résultat de la requête.

2. Exercices

Pratique

Toujours sur la base `imdb`, rédigez des requêtes de sélection de données pour accomplir les tâches suivantes.

Concernant la table `directors` :

1. Affichez le contenu de toutes les colonnes de la table `directors`, ordonnez le résultat par prénom-nom croissant et n'affichez que les 500 premières lignes.
2. Affichez les genres distincts de la table `directors`, ordonnez le résultat par ordre décroissant.
3. Affichez tous les identifiants, prénoms-noms et genres des réalisateurs.
4. Même question qu'au dessus, mais nommer les colonnes de sortie en `Identifiant`, `Prénom Nom` et `Genre`, et utiliser `dir` comme alias pour `directors`.
5. De deux manières différentes, affichez uniquement le prénom-nom et genre des réalisateurs avec un genre codé `0` ou `1`.
6. Affichez les prénoms-noms des réalisateurs commençant par `Steven` et où apparaissent les lettres `ber`. Affichez le résultat en majuscules.
7. Affichez les prénoms-noms des réalisateurs où n'apparaissent pas les lettres `lee` ou `vern`. Affichez le résultat en minuscules.
8. Trouvez le réalisateur qui a un genre différent de `0`, avec un prénom contenant trois lettres dont un `i` en deuxième position et avec un nom se terminant par `ton`.

Concernant la table `movies` :

1. Affichez tous les films pour lesquels le titre original et différent du titre américain. Affichez le titre original, le titre américain puis le slogan.
2. Affichez tous les films pour lesquels le titre est le même que le slogan.
3. Affichez les titres, slogans, notes moyennes, nombres de votes et dates de sortie des 20 films avec la meilleur note moyenne et ayant reçu 50 notes ou plus. Triez par ordre décroissant les notes moyennes et par ordre croissant les titres.
4. Modifiez la requête précédente pour ne considérer dans le classement que les films avec une date de sortie en 2014, 2015 ou 2016.
5. Affichez tous les films pour lesquels les recettes sont inférieures au budget. Ajoutez une colonne nommée `Profit` correspondant à la différence recette - budget. Ne considérer que les films avec des recettes strictement supérieures à 0. Ordonnez les résultats par note moyenne décroissante.
6. Groupez les notes moyennes de telle sorte à compter combien de films ont reçu chaque note moyenne. Ordonnez le résultat par ordre décroissant du nombre de films concernés pour chaque note moyenne.
7. Modifiez la requête précédente pour utiliser la note moyenne arrondie plutôt que la note moyenne, et ne considérer que les films ayant reçu plus de 1000 votes.

Concernant les deux tables :

1. Affichez toutes les informations des tables `movies` et `directors` conjointement. Constatez que les identifiants de réalisateur issus des deux tables sont bien égaux.
2. Affichez pour chaque réalisateur son prénom-nom, le nombre de films réalisés, la note moyenne minimale et maximale. Ordonnez le résultat par nombre de films réalisés décroissant.