Licence 2 – Informatique
Info3Bb – Introduction aux bases de données

# Travaux Pratiques

Annabelle Gillet
Annabelle.Gillet@u-bourgogne.fr

Révision : 11 septembre 2025


## Table des matières

## Connexion au SGBD

Un SGBD PostgreSQL est hébergé sur le serveur `kafka.iem`. Chaque étudiant dispose de sa propre base de données, dont le nom est identique au login IEM. Le mot de passe par défaut est également identique au login. Vous pouvez accéder à votre base de données soit via un terminal et le client `psql`, soit via DBeaver.

Il est possible de se connecter à la base de données via `psql` de la manière suivante :

```
psql -h kafka.iem -U [login IEM] [login IEM]
```

en remplaçant `[login IEM]` par votre login utilisé dans les salles machines. Les requêtes peuvent être saisies directement dans l'invite de commande et doivent être terminées par un point-virgule.

Pour lancer DBeaver, vous pouvez taper la commande suivante dans un terminal :

```
/usr/gide/dbeaver/dbeaver
```

De manière optionnelle, vous pouvez également ajouter le chemin vers l'exécutable de DBeaver dans vos variables d'environnement afin de faciliter son lancement. Dans ce cas, il faut ajouter le chemin dans le `PATH` (si vous souhaitez rendre cet ajout persistant, il faut ajouter la ligne commençant par `export` dans le fichier `.bashrc` à la racine de votre home) :

```
export PATH=/usr/gide/dbeaver/:$PATH
```

Il suffit ensuite de taper `dbeaver` dans un terminal pour pouvoir le lancer.

Pour ajouter une connexion au serveur avec DBeaver, il faut suivre la procédure suivante :

1. Cliquer sur `Nouvelle connexion`
2. Sélectionner `PostgreSQL` comme type de nouvelle connexion
3. Remplir l'interface de la manière suivante, en remplaçant `[login IEM]` par votre login (ce qui n'est pas évoqué n'a pas besoin d'être modifié) :
   — Champ `Host` : `kafka.iem`
   — Champ `Database` : `[login IEM]`
   — Champ `Nom d'utilisateur` : `[login IEM]`
   — Champ `Mot de passe` : `[login IEM]`
   — Si vous paramétrez DBeaver sur votre machine, il faut un `Local client` en version 16

Une fois que la connexion est établie, il est possible d'accéder à l'interface d'écriture de requêtes en faisant un clic droit sur la base (ou sur un schéma) et en sélectionnant `Editeur SQL`.

# 1 Interrogation

**Exercice 1.** Construction de la base de données

Cet exercice sert à mettre en place la base de données qui servira à tester les requêtes SQL des exercices suivants. Le schéma de la base de données est indiqué dans la figure 1.

1. Récupérer les fichiers `creation_schema.sql` et `insertion_donnees.sql` disponibles sur Plubel
2. Charger les données dans votre base à l'aide des commandes suivantes, en remplaçant `[login]` par votre login des salles machines IEM :

```
psql -h kafka.iem -U [login] -f creation_schema.sql [login]
psql -h kafka.iem -U [login] -f insertion_donnees.sql [login]
```

3. Se connecter au client `psql`, puis tester la bonne importation de la base de données grâce à la commande `\dt dvd.*`. Cette commande affiche toutes les tables du schéma `dvd`, et devrait retourner le résultat suivant :

```
              Liste des relations
 Schéma |      Nom        | Type  | Propriétaire
--------+-----------------+-------+--------------
 dvd    | acteur          | table | [login]
 dvd    | adresse         | table | [login]
 dvd    | categorie       | table | [login]
 dvd    | client          | table | [login]
 dvd    | film            | table | [login]
 dvd    | film_acteur     | table | [login]
 dvd    | film_categorie  | table | [login]
 dvd    | inventaire      | table | [login]
 dvd    | location        | table | [login]
 dvd    | magasin         | table | [login]
 dvd    | pays            | table | [login]
 dvd    | ville           | table | [login]
(12 lignes)
```

**Exercice 2.** Premières requêtes

Ces requêtes s'appliquent sur une seule table et ne font pas appel à des agrégations. Comme les tables sont contenues dans le schéma `dvd`, il faut soit préfixer le nom des tables par le nom du schéma (par exemple `dvd.categorie`), soit se positionner dans le schéma grâce à la commande `set schema 'dvd';`. Vous pouvez vous aider de la *cheat sheet* disponible sur Plubel.

1. Afficher tout le contenu de la table `categorie`.

```
SELECT * FROM dvd.categorie;
```

2. Retourner les titres de films qui ont une durée supérieure à 180min.

```
SELECT titre FROM dvd.film WHERE duree > 180;
```

3. Afficher uniquement la ligne correspondant à la toute première location de dvd contenue dans la base de données.

```
SELECT * FROM dvd.location ORDER BY data_location LIMIT 1;
```

4. Afficher uniquement la ligne correspondant à la dernière location de dvd contenue dans la base de données.

```
SELECT * FROM dvd.location ORDER BY data_location DESC LIMIT 1;
```

5. Afficher le titre des films ainsi que leur description pour tous les films qui contiennent le mot "shark" (avec ou sans majuscule) dans leur description.

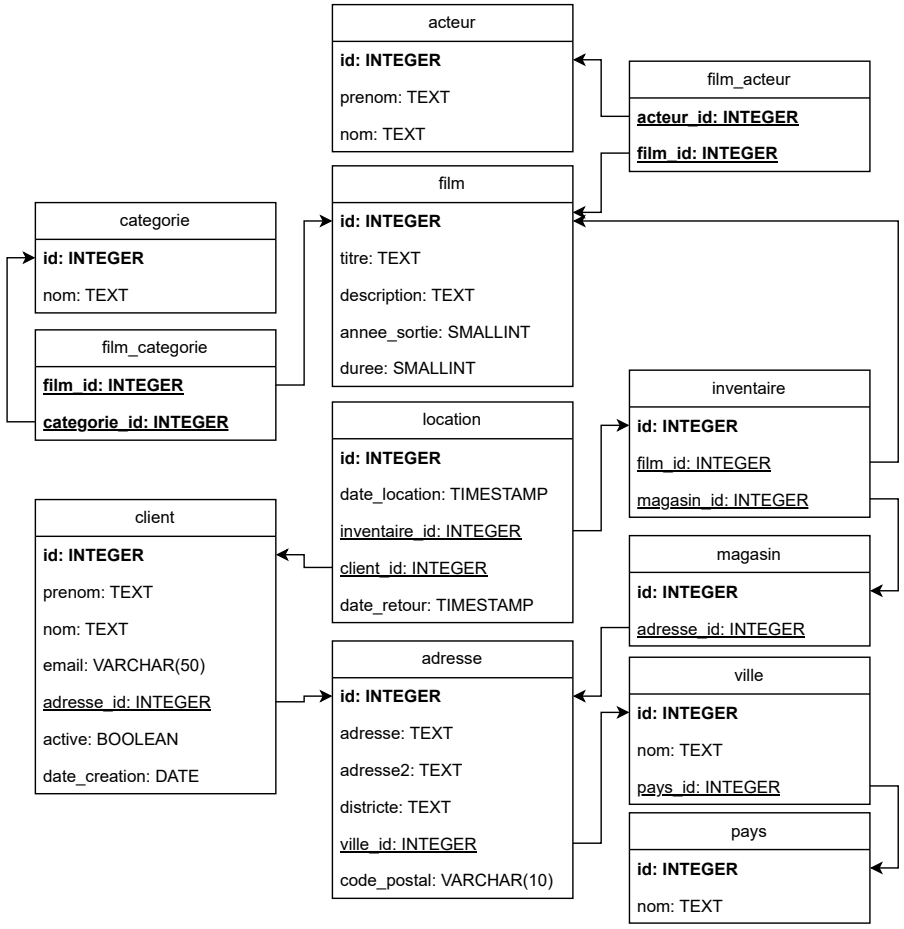


FIGURE 1 – Schéma de la base de données utilisée pour les TP

```
SELECT titre, description
FROM dvd.film
WHERE description ILIKE '%shark%';
```

6. Afficher le titre des films ainsi que leur description pour tous les films qui contiennent le mot "shark" (avec ou sans majuscule) dans leur description, mais pas "shark tank" (toujours avec ou sans majuscule).

```
SELECT titre, description
FROM dvd.film
WHERE description ILIKE '%shark%'
    AND description NOT ILIKE '%shark tank%';
```

## Exercice 3. Jointures simples

Ces requêtes visent à associer plusieurs tables pour pouvoir répondre à des requêtes plus complexes. On se concentre dans un premier temps uniquement sur les jointures INNER JOIN.

1. Afficher le nom des catégories des films ayant l'id 989, 963 et 514.

```
SELECT film_id, nom
FROM dvd.categorie c
    INNER JOIN dvd.film_categorie fc ON c.id = fc.categorie_id
WHERE fc.film_id IN (989, 963, 514);
```

2. Donner la liste des locations de dvd effectuées par le client ayant l'id 1.

```
SELECT *
FROM dvd.client c
    INNER JOIN dvd.location l ON l.client_id = c.id
WHERE c.id = 1;
```

3. Donner le titre des films loués par le client ayant l'id 1.

```
SELECT nom, prenom, titre
FROM dvd.client c
    INNER JOIN dvd.location l ON l.client_id = c.id
    INNER JOIN dvd.inventaire i ON l.inventaire_id = i.id
    INNER JOIN dvd.film f ON i.film_id = f.id
WHERE c.id = 1;
```

4. Sans afficher de doublon, donner les catégories de films dans lesquelles chaque acteur a joué, sous la forme nom – prénom – nom de catégorie.

```
SELECT DISTINCT a.nom, prenom, c.nom
FROM dvd.categorie c
    INNER JOIN dvd.film_categorie fc ON c.id = fc.categorie_id
    INNER JOIN dvd.film_acteur fa ON fa.film_id = fc.film_id
    INNER JOIN dvd.acteur a ON a.id = fa.acteur_id;
```

5. Trouver l'id du magasin localisé au Canada.

```
SELECT *
FROM dvd.magasin m
    INNER JOIN dvd.adresse a ON m.adresse_id = a.id
    INNER JOIN dvd.ville v ON v.id = a.ville_id
    INNER JOIN dvd.pays p ON p.id = v.pays_id
WHERE p.nom ILIKE 'Canada';
```

## Exercice 4. Agrégations

Ces requêtes font appel au mécanisme d'agrégation afin de calculer de nouveaux résultats à partir d'un groupe de tuples.

1. Retourner le nombre total de films contenus dans la base de données.

```
SELECT COUNT(*) FROM dvd.film;
```

2. Donner la durée moyenne de location d'un dvd.

```
SELECT AVG(date_retour - date_location)
FROM dvd.location;
```

3. Donner l'id des clients qui ont loué au moins 5 dvd.

```
SELECT c.id, COUNT(*)
FROM dvd.client c INNER JOIN dvd.location l ON c.id = l.client_id
GROUP BY c.id
HAVING COUNT(*) >= 5;
```

4. Donner le nombre de dvd qui ont été loués mais pas encore rendus.

```
SELECT COUNT(*)
FROM dvd.location
WHERE date_retour IS NULL;
```

5. Donner le nombre de films pour chaque catégorie. On souhaite afficher le résultat sous la forme nom de la catégorie – nombre de films.

```sql
SELECT c.nom, COUNT(*)
FROM dvd.categorie c
    INNER JOIN dvd.film_categorie fc ON c.id = fc.categorie_id
GROUP BY c.nom;
```

6. Pour chaque acteur (on affichera uniquement son id), donner la durée totale des films dans lesquels il a joué.

```sql
SELECT a.id, SUM(duree)
FROM dvd.film AS f
    INNER JOIN dvd.film_acteur AS fa ON f.id = fa.film_id
    INNER JOIN dvd.acteur AS a ON a.id = fa.acteur_id
GROUP BY a.id;
```

# 2 Création

L'entreprise de location de dvd souhaite ajouter un programme de fidélité pour ses clients. Pour cela, des modifications sont à apporter au niveau du schéma de la base de données utilisé au TP précédent. Il est conseillé de garder les commandes exécutées dans un script. En cas d'erreur majeure, il est possible de rétablir la base de données dans son état initial en supprimant le schéma dvd (`DROP SCHEMA dvd CASCADE;`), puis en réexécutant les scripts initiaux de Plubel.

**Exercice 5.** Création d'une table simple

La première étape de l'ajout du programme de fidélité consiste à créer une table `fidelite` qui contiendra des informations sur les différents niveaux de fidélité.

1. Créer la table `fidelite` dans le schéma `dvd`. Cette table doit contenir les attributs suivants :
   — `rang`, de type TEXT, qui est la clé primaire de la table. Cet attribut indique le nom du rang de fidélité ;
   — `points`, de type SMALLINT, indiquant le nombre de points nécessaires pour passer à ce rang de fidélité (une location de dvd = 1 point) ;
   — `reduction`, de type NUMERIC, indiquant la réduction à appliquer lorsqu'un client du rang correspondant loue un dvd.
2. Insérer les données suivantes dans la table nouvellement créée :
   — `rang` = novice – `points` = 0 – `reduction` = 0.00
   — `rang` = amateur – `points` = 5 – `reduction` = 0.10
   — `rang` = intermédiaire – `points` = 10 – `reduction` = 0.15
   — `rang` = avancé – `points` = 20 – `reduction` = 0.20

```
CREATE TABLE dvd.fidelite (
    rang TEXT PRIMARY KEY,
    points SMALLINT,
    reduction NUMERIC
);
INSERT INTO dvd.fidelite VALUES
    ('novice', 0, 0.00),
    ('amateur', 5, 0.10),
    ('intermédiaire', 10, 0.15),
    ('avancé', 20, 0.20);
```

**Exercice 6.** Ajout de contraintes

On souhaite maintenant lier la table `fidelite` à la table `client`.

1. Ajouter une colonne `rang_fidelite` de type TEXT dans la table `client`.
2. Modifier la valeur de cette colonne pour tous les clients afin qu'elle corresponde au niveau de fidélité le plus bas (novice).
3. Ajouter une contrainte de clé étrangère sur la colonne `rang_fidelite` afin qu'elle référence la colonne `rang` de la table `fidelite`.
4. Modifier la valeur de cette colonne pour les clients ayant loué au moins 5 dvd pour qu'elle corresponde au niveau amateur. Répéter cette opération pour les clients ayant loué au moins 10 dvd (intermédiaire) et 20 dvd (avancé). Indice : vous pouvez utiliser une requête imbriquée ou une clause WITH pour d'abord calculer le nombre de locations de chaque client, puis n'appliquer la mise à jour que pour les clients concernés par le niveau de fidélité. Vous pouvez vous appuyer sur la question 3 de l'exercice 4.

```
ALTER TABLE dvd.client ADD COLUMN rang_fidelite TEXT;

UPDATE dvd.client SET rang_fidelite = 'novice';
```

```sql
ALTER TABLE dvd.client ADD CONSTRAINT fk_fidelite FOREIGN KEY (rang_fidelite) REFERENCES
    dvd.fidelite(rang);

UPDATE dvd.client SET rang_fidelite = 'amateur' WHERE id IN (
    SELECT c.id
    FROM dvd.client c INNER JOIN dvd.location l ON l.client_id = c.id
    GROUP BY c.id
    HAVING COUNT(*) >= 5
);

UPDATE dvd.client SET rang_fidelite = 'intermédiaire' WHERE id IN (
    SELECT c.id
    FROM dvd.client c INNER JOIN dvd.location l ON l.client_id = c.id
    GROUP BY c.id
    HAVING COUNT(*) >= 10
);

UPDATE dvd.client SET rang_fidelite = 'avancé' WHERE id IN (
    SELECT c.id
    FROM dvd.client c INNER JOIN dvd.location l ON l.client_id = c.id
    GROUP BY c.id
    HAVING COUNT(*) >= 20
);
```

**Exercice 7.** Impact des clés étrangères

Le nom des rangs de fidélité est jugé non pertinent par la direction et doit être modifié.

1. Mettre à jour tout d'abord la ligne correspondant au rang novice de la table `fidelite` afin de le renommer en bronze, sans modifier les autres valeurs. Cette modification est-elle possible ?
2. Consulter l'encadré 1 afin d'identifier une solution possible pour pouvoir faire cette mise à jour tout en conservant la contrainte de clé étrangère. Modifier la contrainte de clé étrangère afin de rendre possible la mise à jour.
3. Modifier également la valeur des autres rangs de la manière suivante :
   — novice → bronze ;
   — amateur → argent ;
   — intermédiaire → or ;
   — avancé → platine.

```sql
ALTER TABLE dvd.client DROP CONSTRAINT fk_fidelite;

ALTER TABLE dvd.client ADD CONSTRAINT fk_fidelite FOREIGN KEY (rang_fidelite) REFERENCES
    dvd.fidelite(rang) ON UPDATE CASCADE;

UPDATE dvd.fidelite SET rang = 'bronze' WHERE rang = 'novice';
UPDATE dvd.fidelite SET rang = 'argent' WHERE rang = 'amateur';
UPDATE dvd.fidelite SET rang = 'or' WHERE rang = 'intermédiaire';
UPDATE dvd.fidelite SET rang = 'platine' WHERE rang = 'avancé';
```

Encadré 1 – Contrainte de clé étrangère avancée

Une contrainte de clé étrangère peut contenir des directives avancées spécifiant des actions à effectuer en cas de suppression ou de mise à jour de l'attribut référencé [a]. Ces directives sont spécifiées au moment de la création de la clé étrangère de la manière suivante :

```
CREATE TABLE mon_schema.ma_table (
    mon_attribut SMALLINT,
    CONSTRAINT mon_attribut_fk FOREIGN_KEY(mon_attribut) REFERENCES
        mon_schema.autre_table(attribut_reference) ON UPDATE CASCADE
)
```

Pour indiquer une action à réaliser lors d'une mise à jour, la directive `ON UPDATE` est utilisée. La directive `ON DELETE` est utilisée dans le cas d'une suppression. Les actions associées peuvent être les suivantes :

- NO ACTION : valeur par défaut. Ne réalise pas d'action particulière et produit une erreur si la modification ou suppression viole la contrainte ;
- RESTRICT : similaire à NO ACTION, la différence réside dans le moment de vérification de la contrainte ;
- CASCADE : propage la modification ou la suppression, soit en modifiant la valeur des attributs référençant l'attribut modifié, soit en supprimant les lignes référençant l'attribut supprimé ;
- SET NULL : applique la valeur NULL pour les attributs de la clé étrangère (ou un sous-ensemble de ces attributs)
- SET DEFAULT : applique la valeur par défaut pour les attributs de la clé étrangère (ou un sous-ensemble de ces attributs).

*a.* https://www.postgresql.org/docs/current/sql-createtable.html#SQL-CREATETABLE-PARMS-REFERENCES

# 3 Interrogation avancée

## Exercice 8.

1. Donner les 10 films les plus populaires (qui sont le plus loués).

```
SELECT titre
FROM dvd.film f
    INNER JOIN dvd.inventaire i ON i.film_id = f.id
    INNER JOIN dvd.location l ON l.inventaire_id = i.id
GROUP BY titre
ORDER BY COUNT(*) DESC
LIMIT 10;
```

2. Donner les 10 films qui sont les plus appréciés (qui sont gardés en location le plus longtemps).

```
SELECT titre
FROM dvd.film f
    INNER JOIN dvd.inventaire i ON i.film_id = f.id
    INNER JOIN dvd.location l ON l.inventaire_id = i.id
GROUP BY titre
ORDER BY AVG(date_retour - date_location) DESC
LIMIT 10;
```

3. Donner en une requête la concaténation des 2 listes précédentes (10 films les plus loués et 10 filmes conservés le plus longtemps).

```
(SELECT titre
FROM dvd.film f
    INNER JOIN dvd.inventaire i ON i.film_id = f.id
    INNER JOIN dvd.location l ON l.inventaire_id = i.id
GROUP BY titre
ORDER BY COUNT(*) DESC
LIMIT 10)
UNION
(SELECT titre
FROM dvd.film f
    INNER JOIN dvd.inventaire i ON i.film_id = f.id
    INNER JOIN dvd.location l ON l.inventaire_id = i.id
GROUP BY titre
ORDER BY AVG(date_retour - date_location) DESC
LIMIT 10);
```

4. Donner la liste des clients dont la durée moyenne de location d'un dvd est supérieure à la durée moyenne des tous les clients + 1 jour. Indice : il est possible d'ajouter 1 jour à un intervalle en utilisant `[duree] + interval '1' day`

```
WITH duree_location AS (
    SELECT AVG(date_retour - date_location) AS moyenne
    FROM dvd.location
)
SELECT c.id, AVG(l.date_retour - l.date_location), AVG(moyenne)
FROM dvd.client c
    INNER JOIN dvd.location l ON l.client_id = c.id,
    duree_location
GROUP BY c.id
HAVING AVG(l.date_retour - l.date_location) > (AVG(moyenne) + interval '1' day);
```

## Exercice 9.

1. Afficher le top 5 des acteurs (nom et prénom) ayant joué dans le plus de films, ainsi que le nombre de films dans lesquels ils ont joué. Attention, des homonymes peuvent être présents.

```sql
WITH acteur_popularite AS (
    SELECT acteur.id AS acteur_id, COUNT(*) AS nb_films
    FROM dvd.acteur AS acteur INNER JOIN dvd.film_acteur AS film ON acteur.id =
        film.acteur_id
    GROUP BY acteur.id
)
SELECT nom, prenom, nb_films
FROM dvd.acteur INNER JOIN acteur_popularite ON acteur_id = id
ORDER BY nb_films DESC
LIMIT 5;

SELECT nom, prenom, COUNT(*)
FROM dvd.acteur acteur INNER JOIN dvd.film_acteur film ON acteur.id = film.acteur_id
GROUP BY acteur.id, nom, prenom
ORDER BY COUNT(*) DESC;
```

2. Donner la liste des clients qui n'ont jamais loué de dvd. Proposer une version de cette requête avec une jointure uniquement (identifier le type de jointure adapté), et une version utilisant uniquement la table `client` dans le FROM.

```sql
SELECT *
FROM dvd.client c LEFT JOIN dvd.location l ON l.client_id = c.id
WHERE l.id IS NULL;

SELECT *
FROM dvd.client
WHERE id NOT IN (SELECT DISTINCT client_id FROM dvd.location);
```

3. Donner la somme payée par chaque client pour ses locations de dvd, en considérant qu'une location coûte 2€. Ne pas oublier de prendre en compte la réduction liée au niveau de fidélité des clients. Pour simplifier, on considérera que le niveau de fidélité de chaque client est valable pour toutes ses locations, sans prendre en considération le côté progressif.

```sql
WITH tarif AS (
    SELECT id AS client_id, (2 * (1 - reduction)) AS tarif
    FROM dvd.client c INNER JOIN dvd.fidelite f ON c.rang_fidelite = f.rang
)
SELECT c.id, SUM(tarif)
FROM dvd.client c
    INNER JOIN tarif t ON t.client_id = c.id
    LEFT JOIN dvd.location l ON c.id = l.client_id
GROUP BY c.id;
```

4. Pour chaque magasin, donner le nombre de clients qui ont une adresse dans le même pays, et le nombre de clients qui ont une adresse dans un pays différent. Retourner une seule ligne par magasin contenant toutes ces informations. Attention à ne pas compter de clients en doublon. Indice : le nombre de clients du même pays et le nombre de clients venant d'un pays différent peuvent être décomposés en 2 sous-requêtes, éventuellement dans une clause WITH.

```sql
WITH nb_client_meme_ville AS (
    SELECT m.id AS magasin_id, COUNT(DISTINCT c.id) AS nb
    FROM dvd.client c
        INNER JOIN dvd.adresse ca ON c.adresse_id = ca.id
        INNER JOIN dvd.location l ON l.client_id = c.id
        INNER JOIN dvd.inventaire i ON i.id = l.inventaire_id
        INNER JOIN dvd.magasin m ON m.id = i.magasin_id
        INNER JOIN dvd.adresse ma ON m.adresse_id = ma.id
        INNER JOIN dvd.ville cv ON cv.id = ca.ville_id
        INNER JOIN dvd.ville mv ON mv.id = ma.ville_id
    WHERE cv.pays_id = mv.pays_id
    GROUP BY m.id
```

```
), nb_client_ville_different AS (
    SELECT m.id AS magasin_id, COUNT(DISTINCT c.id) AS nb
    FROM dvd.client c
        INNER JOIN dvd.adresse ca ON c.adresse_id = ca.id
        INNER JOIN dvd.location l ON l.client_id = c.id
        INNER JOIN dvd.inventaire i ON i.id = l.inventaire_id
        INNER JOIN dvd.magasin m ON m.id = i.magasin_id
        INNER JOIN dvd.adresse ma ON m.adresse_id = ma.id
        INNER JOIN dvd.ville cv ON cv.id = ca.ville_id
        INNER JOIN dvd.ville mv ON mv.id = ma.ville_id
    WHERE cv.pays_id != mv.pays_id
    GROUP BY m.id
)
SELECT m.id, mv.nb AS meme_ville, dv.nb AS ville_different
FROM dvd.magasin m
    LEFT JOIN nb_client_meme_ville mv ON mv.magasin_id = m.id
    LEFT JOIN nb_client_ville_different dv ON dv.magasin_id = m.id;
```

5. Pour chaque magasin et chaque catégorie de film, donner le nombre de films de cette catégorie possédés par le magasin.

```
SELECT m.id, c.nom, COUNT(*)
FROM dvd.categorie c
    INNER JOIN dvd.film_categorie fc ON c.id = fc.categorie_id
    INNER JOIN dvd.film f ON fc.film_id = f.id
    INNER JOIN dvd.inventaire i ON i.film_id = f.id
    INNER JOIN dvd.magasin m ON m.id = i.magasin_id
GROUP BY m.id, c.nom
ORDER BY COUNT(*) DESC;
```

6. Donner pour chaque magasin les 2 catégories de films qu'il possède le plus. Est-il possible de répondre à cette question avec un LIMIT ?

```
WITH tmp AS (
    SELECT m.id, c.nom, COUNT(*), rank() OVER (PARTITION BY m.id ORDER BY COUNT(*) DESC)
        AS rang
    FROM dvd.categorie c
        INNER JOIN dvd.film_categorie fc ON c.id = fc.categorie_id
        INNER JOIN dvd.film f ON fc.film_id = f.id
        INNER JOIN dvd.inventaire i ON i.film_id = f.id
        INNER JOIN dvd.magasin m ON m.id = i.magasin_id
    GROUP BY m.id, c.nom
    ORDER BY COUNT(*) DESC
)
SELECT *
FROM tmp
WHERE rang <= 2;
```

# 4 Utilisation de la BD depuis un programme

En Java, la plupart des systèmes de gestion de bases de données sont accessibles via l'API JDBC (Java DataBase Connectivity). Cette API propose des interfaces pour de connecter et interagir avec une base de données. Pour accéder à un SGBD spécifique, il est nécessaire d'inclure son pilote lors de l'exécution du code, qui propose une implémentation spécifique des interfaces disponibles avec JDBC, bien souvent sous forme de `jar`. Le pilote pour le SGBD PostgreSQL est disponible sur l'espace du cours de Plubel.

### Exercice 10. Initialisation de la connexion

Dans cet exercice, l'objectif est d'établir et de valider la connexion à votre base de données à travers les étapes suivantes.

1. Récupérer le `jar` du pilote PostgreSQL sur Plubel.
2. Faire une classe Java simple avec un `main`, qui servira dans un premier temps à regrouper les lignes de code nécessaires pour se connecter à la base de données.
3. Importer les classes indispensables à l'ouverture d'une connexion :

```
import java.sql.*;
```

4. Charger le pilote grâce à la ligne suivante :

```
Class.forName("org.postgresql.Driver");
```

5. Créer un objet de type `Connection` de la manière suivante :

```
Connection connexion = DriverManager.getConnection("[url]", "[login]", "[password]");
```

   Les valeurs `[login]` et `[password]` sont à remplacer par les valeurs correspondant à celles permettant de se connecter à votre base de données. L'`[url]` doit indiquer le protocole de connexion, son adresse, éventuellement son port, et le nom de la base à laquelle se connecter. Pour PostgreSQL, elle doit être de la forme suivante : `jdbc:postgresql://kafka.iem/[bd]`, `[bd]` étant à remplacer par le nom de votre base (identique à votre login).

6. Ajouter le code suivant, qui permet d'envoyer une requête à la base de données et de consulter son contenu, afin de tester la connexion nouvellement créée :

```
Statement statement = connexion.createStatement();
ResultSet resultSet = statement.executeQuery("select version()");
while (resultSet.next()) {
    System.out.println(resultSet.getString(1));
}
resultSet.close();
statement.close();
```

   Penser à bien gérer les exceptions qui peuvent être levées lors de l'établissement de la connexion et des interactions avec la base de données (voir encadré 2).

7. Compiler et exécuter votre code grâce aux lignes suivantes (si le jar n'est pas au même emplacement que la classe, son chemin est à indiquer en plus de son nom) :

```
javac NomClasse.java
java -cp postgresql-42.7.3.jar:. NomClasse
```

   Si aucune erreur n'apparaît et qu'un message commençant par `PostgreSQL 16.3` s'affiche, cela signifie que la connexion est correctement établie.

### Exercice 11. `Statement` et `ResultSet`

Pour interagir avec la base de données, JDBC propose une classe `Statement` qui peut être obtenue à partir d'un objet `Connection` grâce à la méthode `createStatement()`.

ENCADRÉ 2 – Gestion des exceptions

La plupart des appels à des méthodes de l'API JDBC peuvent lever une exception de type `SQLException` (par exemple, en cas de mauvais paramètres de connexion, d'indisponibilité de la base de données ou d'erreur dans une requête envoyée). Il est donc nécessaire de gérer ces erreurs dans le programmes. En Java, cela peut se faire à l'aide des blocs `try/catch`.

```
try {
    // Code qui peut lever une exception
} catch (SQLException e) {
    // En cas d'exception levée, l'exécution du code contenu dans le bloc try
    // est interrompue, et le code contenu dans le bloc catch est exécuté.
} finally {
    // Le bloc finally est optionnel, il permet d'exécuter son bloc de code dans tous
    // les cas (si une exception a été levée ou non). Il peut par exemple servir à
    // fermer des ressources (comme un Statement ou un ResultSet).
}
```

Par exemple, dans le cas de l'exécution d'une requête, le code du `try/catch` peut se présenter de la façon suivante :

```
Statement statement = null;
ResultSet resultSet = null;
try {
    Statement statement = connexion.createStatement();
    ResultSet resultSet = statement.executeQuery("select version()");
    while (resultSet.next()) {
        System.out.println(resultSet.getString(1));
    }
} catch (SQLException e) {
    e.printStackTrace(); // Permet d'afficher l'erreur dans la console
    // Le code du bloc catch peut être plus sophistiqué, et peut permettre
    // par exemple de corriger la valeur d'une variable si l'exécution s'est mal passée
} finally {
    if (resultSet != null) { // Peut être null si l'exception s'est produite avant
    // le retour de la méthode executeQuery
        resultSet.close();
    }
    if (statement != null) { // Peut être null si l'exception s'est produite
    // pendant l'appel à la méthode createStatement
        statement.close();
    }
}
```

À partir de Java 8, pour les ressources qui implémentent l'interface `AutoClosable`, une syntaxe simplifiée du `try` permet de fermer automatiquement les ressources sans faire appel au bloc `finally`. Cette syntaxe permet aussi d'éviter la création des variables contenant les ressources en dehors du `try/catch`.

```
// Toutes les ressources ouvertes dans la parenthèse du try seront fermées
// automatiquement à la sortie du try/catch
try(Statement statement = connexion.createStatement();
    ResultSet resultSet = statement.executeQuery("select version()")) {
    while (resultSet.next()) {
        System.out.println(resultSet.getString(1));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

1. La méthode `executeQuery()` d'un `Statement` prend en paramètre une requête sous forme de `String`, et retourne un `ResultSet` contenant le résultat de la requête (qui ne doit pas se terminer par un point-virgule). Créer un `Statement` à partir de la `Connection` ouverte dans l'exercice précédent, et exécuter la requête suivante avec JDBC :

```
SELECT id, nom
FROM dvd.categorie
```

2. Un `ResultSet` peut être vu comme un itérateur, permettant de parcourir les lignes retournées par la requête. La méthode `next()` permet de passer à la ligne suivante et retourne un booléen ayant la valeur `true` s'il a pu passer à la ligne suivante, ou `false` s'il n'y a plus de ligne à consulter. Le parcours d'un `ResultSet` se présente donc souvent de la manière suivante :

```
while (resultSet.next()) {
    // Traitement sur le ResultSet
}
```

   Mettre en place cette boucle pour le parcours du `ResulSet` récupéré à la question précédente.

3. Un `ResultSet` dispose de méthodes permettant de récupérer la valeur d'un attribut à partir du nom de la colonne ou de son numéro dans l'ordre d'apparition dans le SELECT. Il est toujours préférable d'utiliser le nom de la colonne plutôt que son numéro, afin d'éviter les erreurs de référencement lorsque la requête est complexe ou qu'elle est modifiée ultérieurement. Il existe une méthode pour chaque type qui peut être récupéré (par exmple `getString("colonne")`, `getDate("colonne")`, `getInt("colonne")`). La liste complète des méthodes peut être consultées sur la javadoc [1]. Dans votre boucle `while`, récupérer l'id et le nom de la catégorie avec les types correspondants, et les afficher avec `System.out.println`.
4. Un `Statement` dispose également d'une méthode `executeUpdate()`, permettant d'effectuer des opérations de modification de la base de données (création de tables, insertion de tuples, etc.). Cette méthode prend en paramètre la requête de création, et retourne un `int`, spécifiant le nombre de lignes modifiées lors de l'exécution de la requête. Insérer une nouvelle ligne dans la table `pays`, et vérifier que la méthode `executeUpdate()` a bien retourné 1 (la valeur ajoutée n'a pas forcément besoin de correspondre à un pays existant).
5. Utiliser la méthode `executeUpdate()` pour supprimer la ligne que vous venez d'ajouter.

## Exercice 12. `PreparedStatement`, un `Statement` paramétré

Avec JDBC, il est également possible de préparer une requête dont certains paramètres restent à spécifier ultérieurement. Cela présente un fort intérêt lorsqu'une requête va être exécutée régulièrement avec un paramètre différent (par exemple, un utilisateur voulant accéder aux informations le concernant lui uniquement). On utilise dans ce cas un `PreparedStatement` à la place d'un `Statement`.

1. La requête est définie lors de la création du `PreparedStatement` et non plus lors de l'utilisation des méthodes `executeQuery()` ou `executeUpdate()`. Chaque paramètre à spécifier sera indiqué par un `?`, puis sa valeur sera complétée à l'aide d'une méthode `setXXX()` (XXX étant à remplacer par le type du paramètre [2]), qui prend en paramètres l'index du paramètre à remplacer et sa valeur.

```
PreparedStatement preparedStatement = connexion.prepareStatement("SELECT * FROM
    schema.table WHERE nom = ? AND id = ?");
preparedStatement.setString(1, "nom1");
preparedStatement.setInt(2, 10);
```

   Préparer de cette manière une requête permettant de récupérer les acteurs en fonction d'un nom passé en paramètre.

1. https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html
2. https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html

2. Les méthodes `executeQuery()` et `executeUpdate()` sont également disponibles pour les `PreparedStatement`. Exécuter la requête de la question précédente après avoir fixé une valeur pour le paramètre, puis afficher le contenu du `ResultSet` récupéré.

**Exercice 13.** Création d'une classe utilitaire

Afin de faciliter l'utilisation de JDBC dans la suite de votre cursus universitaire, vous pouvez créer une classe utilitaire qui servira à initier la connexion à une base de données. Cette classe pourra ensuite évoluer en fonction des besoins (par exemple, créer une classe singleton pour définir une unique connexion pour toute une application, ou un *pool* de connexions pour gérer efficacement les ressources).

1. Définir les variables `LOGIN`, `PASSWORD` et `URL` en tant que constantes de la classe.
2. Définir une méthode `static` permettant de récupérer un objet de type `Connection`.
3. Définir une méthode `static` prenant en paramètre un objet de type `Connection` afin de fermer cette dernière dans votre classe.
4. Vous pouvez également mettre à disposition d'autres méthodes que vous jugez utiles (par exemple, une méthode prenant en paramètre le `String` d'une requête et retournant le `ResultSet` correspond, afin de décharger l'utilisateur de votre classe des étapes de création de `Statement`).