Rapport : CineNet

Théo Raoul et Mathusan Selvakumar

27 Mai 2024

1 Introduction

Pour ce projet, nous avons choisi d'utiliser des données réelles pour créer une base de données de films, d'acteurs, de réalisateurs, d'événements et de studios. Cette approche permet une plus grande authenticité et fidélité à la réalité. Cependant, certaines interactions avec les utilisateurs, les projections d'événements et d'autres éléments ont été générés dynamiquement à l'aide de Python. Pour garantir la qualité et la reproductibilité des données, nous avons utilisé la même graine de génération pour l'ensemble du processus.

2 Modèle Entité-Association

Le diagramme Entité-Association pour CineNet est fourni en un seul fichier PDF séparemment. Les noms des associations sont omis pour des raisons de lisibilité.

3 Schéma Relationnel

UserRoles (type, name, description)

Users (id, last_name, first_name, username, email, password_hashed, birth_date, role_type)
Users[role_type] ⊆ UserRoles[type]

Countries (country_code, name)

Cities (city_code, country_code, name)
Cities[country_code] ⊆ Countries[country_code]

UserLocation (user_id, city_code)
UserLocation[user_id] ⊆ Users[id], UserLocation[city_code] ⊆ Cities[city_code]

Friendship (initiator_id, recipient_id, date)
Friendship[initiator_id] ⊆ Users[id], Friendship[recipient_id] ⊆ Users[id]

Following (follower_id, followed_id, date)
Following[follower_id] ⊆ Users[id], Following[followed_id] ⊆ Users[id]

Categories (id, name, description)

Posts (id, user_id, date, content, parent_post_id, category_id)
Posts[user_id] ⊆ Users[id], Posts[parent_post_id] ⊆ Posts[id], Posts[category_id] ⊆ Categories[id]

Tags (id, name)

PostTags (tag_id, post_id)
PostTags[tag_id] ⊆ Tags[id], PostTags[post_id] ⊆ Posts[id]

Reactions (user_id, post_id, emoji_unicode)
Reactions[user_id] ⊆ Users[id], Reactions[post_id] ⊆ Posts[id]

Events (id, name, date, city_code, organizer_id, capacity, ticket_price)
Events[city_code] ⊆ Cities[city_code], Events[organizer_id] ⊆ Users[id]

Participation (user_id, event_id, type_participation)
Participation[user_id] ⊆ Users[id], Participation[event_id] ⊆ Events[id]

Genres (id, name, parent_genre_id)
Genres[parent_genre_id] ⊆ Genres[id]

Studios (id, name)

Movies (id, title, duration, release_date)

MovieGenres (movie_id, genre_id)
MovieGenres[movie_id] ⊆ Movies[id], MovieGenres[genre_id] ⊆ Genres[id]

MovieStudios (studio_id, movie_id)
MovieStudios[studio_id] ⊆ Studios[id], MovieStudios[movie_id] ⊆ Movies[id]

People (id, last_name, first_name, birth_date)

PeopleRoles (id, name)

MovieCollaborators (people_id, movie_id, role_id)
MovieCollaborators[people_id] ⊆ People[id], MovieCollaborators[movie_id] ⊆ Movies[id], MovieCollaborators[role_id] ⊆ PeopleRoles[id]

Screenings (event_id, movie_id, screening_time)
Screenings[event_id] ⊆ Events[id], Screenings[movie_id] ⊆ Movies[id]

UserEventRatings (user_id, event_id, rating)
UserEventRatings[user_id] ⊆ Users[id], UserEventRatings[event_id] ⊆ Events[id]

UserMovieRatings (user_id, movie_id, rating)
UserMovieRatings[user_id] ⊆ Users[id], UserMovieRatings[movie_id] ⊆ Movies[id]

MovieRecommendation (user_id, movie_id, score_recommendation)
MovieRecommendation[user_id] ⊆ Users[id], MovieRecommendation[movie_id] ⊆ Movies[id]

CompletedEventRecommendation (user_id, event_id, score_recommendation)
CompletedEventRecommendation[user_id] ⊆ Users[id], CompletedEventRecommendation[event_id] ⊆ Events[id]

ScheduledEventRecommendation (user_id, event_id, score_recommendation)
ScheduledEventRecommendation[user_id] ⊆ Users[id], ScheduledEventRecommendation[event_id] ⊆ Events[id]

PostRecommendation (user_id, post_id, score_recommendation)
PostRecommendation[user_id] ⊆ Users[id], PostRecommendation[post_id] ⊆ Posts[id]

4 Les Choix et Limites de la Modélisation

Pour le projet CineNet, nous avons opté pour une base de données centrée exclusivement sur les films, excluant les séries télévisées. Cette décision vise à simplifier la modélisation et la gestion de la base de données. Les films sont des entités relativement simples à modéliser et suffisamment nombreux pour permettre une analyse pertinente. De plus, ils sont des objets culturels très populaires, souvent utilisés comme références dans les discussions sur le cinéma.

Table `UserRoles`

Bien que la table `UserRoles` puisse sembler peu intéressante à première vue, elle a été conçue pour permettre des permissions variées selon les différents types d'utilisateurs. Un utilisateur lambda a un rôle moins important qu'un studio de production, par exemple. Un nouvel attribut `permission` permettrait d'implémenter des fonctionnalités plus avancées selon le type d'utilisateur.

Tables `Cities`, `Countries` et `UserLocations`

Les utilisateurs peuvent indiquer leur ville de résidence, et les tables `Cities`, `Countries` et `UserLocations` facilitent la localisation des utilisateurs. Connaître la ville de résidence permet de créer des requêtes intéressantes, telles que la recherche d'événements ayant lieu dans une ville spécifique.

Table `Friendships`

Un utilisateur peut être ami avec un autre, mais une seule direction est créée dans la table `Friendships`. Cela peut sembler être une limitation, car il faut vérifier les deux sens pour confirmer une amitié. Toutefois, cette approche permet de différencier celui qui a initié la demande et la date de la demande.

Table `Following`

Un utilisateur peut suivre un autre utilisateur sans que ce dernier ne le suive en retour, permettant ainsi une asymétrie dans les relations de suivi.

Table `Categories`

Les catégories de forums sont limitées au premier niveau, à l'instar des subreddits sur Reddit. Il n'est pas possible d'avoir un sous-forum dans un autre, ce qui simplifie la structure et l'utilisation.

Table `Posts`

Une publication est signée par un utilisateur (l'auteur) qui possède un nom d'utilisateur et un mot de passe crypté pour une connexion sécurisée. Une publication peut ou non être une réponse à une autre publication et peut ou non appartenir à une catégorie. Ne pas avoir de catégorie signifie que la publication est générale.

Système de Mots-Clés

Grâce aux tables `Tags` et `PostTags`, il est possible de faire des recherches rapides en utilisant des titres de films, des noms d'acteurs ou des genres cinématographiques comme mots-clés. Les recherches plus avancées, telles que l'affichage des sous-genres pour un genre donné, doivent être effectuées au niveau de l'application en utilisant les tags et en comparant avec la table des genres, puis en récupérant récursivement les sous-genres.

Réactions aux Publications

Les utilisateurs peuvent réagir aux publications avec des emojis représentant différentes appréciations.

Table `Events`

Les utilisateurs peuvent indiquer leur intérêt ou leur participation effective à un événement. Les événements peuvent avoir trois états : `Scheduled` (Prévu), `Completed` (Fini), ou `Cancelled` (Annulé).

5 Contraintes

Les contraintes d'intégrité et les contraintes externes sont des conditions essentielles imposées sur les données afin de garantir leur cohérence, leur validité et leur intégrité au sein de la base de données CineNet. Voici les principales catégories de contraintes appliquées dans ce projet, avec des exemples de code SQL pour justifier leur utilisation :

5.1 Contraintes de Cardinalité

Les contraintes de cardinalité définissent le nombre minimal et maximal d'occurrences d'une entité pouvant être associée à une occurrence d'une autre entité. Elles permettent de préciser les relations entre les entités et de garantir la cohérence des associations.

— **Amitié (Friendship)** : Une amitié entre deux utilisateurs doit être unique, et un utilisateur ne peut pas être ami avec lui-même. Cela garantit que chaque relation d'amitié est bidirectionnelle et unique.

```
CREATE TABLE Friendship (
    initiator_id integer NOT NULL REFERENCES Users(id),
    recipient_id integer NOT NULL REFERENCES Users(id) CONSTRAINT friendship_check
        CHECK (initiator_id != recipient_id),
    date timestamp NOT NULL CONSTRAINT friendship_date_check CHECK (date <=
        CURRENT_TIMESTAMP),
    PRIMARY KEY (initiator_id, recipient_id)
);
```

— **Participation à un événement (Participation)** : Un utilisateur peut participer ou être intéressé par plusieurs événements, et chaque événement peut avoir plusieurs participants. Cela permet de modéliser des interactions multiples entre utilisateurs et événements.

```
CREATE TABLE Participation (
    user_id integer NOT NULL REFERENCES Users(id),
    event_id integer NOT NULL REFERENCES Events(id),
    type_participation varchar(255) NOT NULL CONSTRAINT check_type_participation
        CHECK (type_participation IN ('Interested', 'Participating')),
    PRIMARY KEY (user_id, event_id)
);
```

5.2 Dépendance Fonctionnelle

Les contraintes de dépendance fonctionnelle assurent que certaines colonnes d'une table dépendent uniquement d'autres colonnes de la même table. Elles permettent de maintenir la cohérence interne des tables et d'éviter les anomalies de mise à jour.

— **Genres de films (MovieGenres)** : Chaque genre de film doit être unique, et les sous-genres dépendent des genres principaux. Cela garantit une hiérarchie claire des genres et sous-genres de films.

```
CREATE TABLE Genres (
    id integer PRIMARY KEY,
    name varchar(255) UNIQUE NOT NULL,
    parent_genre_id integer REFERENCES Genres(id)
);
```

5.3 Contraintes de Clé Étrangère

Les contraintes de clé étrangère garantissent que les valeurs d'une colonne dans une table existent dans une colonne de la table référencée, assurant ainsi l'intégrité référentielle entre les tables.

— **Localisation des utilisateurs (UserLocations)** : Chaque utilisateur doit avoir une ville de résidence qui doit exister dans la table des villes. Cette contrainte lie chaque utilisateur à une localisation valide.

```
CREATE TABLE UserLocations (
    user_id integer NOT NULL REFERENCES Users(id),
    city_code integer NOT NULL REFERENCES Cities(city_code),
    PRIMARY KEY (user_id, city_code)
);
```

— **Organisation des événements (Events)** : Chaque événement doit être organisé par un utilisateur existant et doit se dérouler dans une ville existante. Cela garantit que les événements sont toujours associés à des utilisateurs et des lieux valides.

```
CREATE TABLE Events (
    id integer PRIMARY KEY,
    name varchar(255) NOT NULL,
    date timestamp NOT NULL,
    city_code integer NOT NULL REFERENCES Cities(city_code),
    organizer_id integer NOT NULL REFERENCES Users(id),
    capacity integer NOT NULL CONSTRAINT check_capacity CHECK (capacity > 0),
    ticket_price numeric NOT NULL CONSTRAINT check_ticket_price CHECK (ticket_price
        >= 0)
);
```

5.4 Contraintes de Validité

Les contraintes de validité assurent que les données respectent certains critères définis pour maintenir leur exactitude et leur cohérence.

— **Événements (Events)** : Les capacités des événements doivent être supérieures à zéro et les prix des billets ne peuvent pas être négatifs. Cela garantit que les informations sur les événements sont logiques et réalistes.

```
CREATE TABLE Events (
    id integer PRIMARY KEY,
    name varchar(255) NOT NULL,
    capacity integer NOT NULL CONSTRAINT check_capacity CHECK (capacity > 0),
    ticket_price numeric NOT NULL CONSTRAINT check_ticket_price CHECK (ticket_price
        >= 0)
);
```

— **Réactions aux publications (Reactions)** : Les utilisateurs peuvent réagir aux publications avec des emojis spécifiques représentant différentes appréciations. Cela permet de standardiser les réactions possibles aux publications.

```
CREATE TABLE Reactions (
    user_id integer NOT NULL REFERENCES Users(id),
    post_id integer NOT NULL REFERENCES Posts(id),
    emoji varchar(10) NOT NULL,
    PRIMARY KEY (user_id, post_id, emoji)
);
```

Conclusion

Les contraintes d'intégrité et externes sont cruciales pour la qualité des données de CineNet, assurant leur cohérence et validité.

6 Requêtes

6.1 Requête SQL pour la Récupération des Utilisateurs Participant à un Événement Planifié

Cette requête SQL implique au moins trois tables pour récupérer des événements. Elle sélectionne le nom d'utilisateur des utilisateurs participant à un événement planifié dans une ville et un pays spécifiques où ils sont situés. La requête reçoit deux paramètres : le nom de l'événement et le nom de la ville.

```
SELECT
    U.username,
    E.date
FROM
    Users U
    JOIN UserLocations UL ON U.id = UL.user_id
    JOIN Cities C ON UL.city_code = C.city_code
    JOIN Countries CO ON C.country_code = CO.country_code
    JOIN Participation P ON U.id = P.user_id
    JOIN Events E ON P.event_id = E.id
WHERE
    E.status = 'Scheduled'
    AND P.type_participation = 'Participating'
    AND E.name = :eventname
    AND CO.name = :countryname
;
```

6.2 Requête SQL pour la Récupération des Utilisateurs Suivant un Utilisateur Spécifique

Cette requête SQL implique une auto-jointure de la table `Users` pour récupérer les utilisateurs qui suivent un autre utilisateur. Elle sélectionne le nom d'utilisateur des utilisateurs qui suivent un utilisateur spécifique. La requête reçoit un paramètre : le nom d'utilisateur.

```sql
SELECT
    A.username AS Follower,
    B.username AS Followed
FROM
    Users A
    JOIN FOLLOWING F ON A.id = F.follower_id
    JOIN Users B ON F.followed_id = B.id
WHERE
    B.username = :username;
```

6.3 Requête SQL pour la Récupération des Utilisateurs Suivant un Nombre Minimum d'Utilisateurs

Cette requête SQL implique une sous-requête corrélée pour récupérer les utilisateurs qui suivent un autre utilisateur. Elle sélectionne le nom d'utilisateur des utilisateurs qui suivent un nombre minimum d'autres utilisateurs. La requête reçoit un paramètre : le nombre minimum d'utilisateurs suivis.

```sql
SELECT
    username,
(
        SELECT
            COUNT(*)
        FROM
            FOLLOWING F
        WHERE
            F.follower_id = U.id) AS Following_Count
FROM
    Users U
WHERE (
    SELECT
        COUNT(*)
    FROM
        FOLLOWING F
    WHERE
        F.follower_id = U.id) >= :minfollowingcount;
```

6.4 Requête SQL pour le Calcul de la Popularité Moyenne des Utilisateurs Suivis par Pays

Cette requête SQL implique une sous-requête dans le FROM. Elle sélectionne le nom du pays et la popularité moyenne des utilisateurs suivis par pays.

```sql
SELECT
    CO.name AS Country_Name,
    AVG(PopularityScores.Popularity) AS Average_Popularity
FROM
    Countries AS CO
    JOIN (
        SELECT
            COUNT(*) AS Popularity,
            UL.city_code
        FROM
            FOLLOWING AS F
            JOIN Users AS U ON F.followed_id = U.id
            JOIN UserLocations AS UL ON U.id = UL.user_id
        GROUP BY
            UL.city_code) AS PopularityScores ON CO.country_code =(
        SELECT
            country_code
        FROM
            Cities
        WHERE
```

```
            city_code = PopularityScores.city_code)
GROUP BY
    CO.name;
```

6.5 Requête SQL pour la Récupération des Utilisateurs Postant Après une Date Spécifique

Cette requête SQL implique une sous-requête dans le WHERE. Elle sélectionne les nom d'utilisateurs qui ont posté après une date spécifique.

```
SELECT U.username
FROM Users U
WHERE U.id IN (SELECT user_id FROM Posts WHERE date > :date)
```

6.6 Requête SQL pour la Récupération du Nombre d'Utilisateurs par Pays

Cette requête SQL implique une fonction d'agrégation avec une clause GROUP BY et une clause HAVING. Elle sélectionne le nom de chaque pays et le nombre d'utilisateurs dans chaque pays. Elle n'inclut que les pays ayant un nombre d'utilisateurs supérieur ou égal au nombre minimum d'utilisateurs spécifié.

```
SELECT
    CO.name,
    COUNT(*) AS User_Count
FROM
    UserLocations UL
    JOIN Cities C ON UL.city_code = C.city_code
    JOIN Countries CO ON C.country_code = CO.country_code
GROUP BY
    CO.name
HAVING
    COUNT(*) >= :minusercount;
```

6.7 Requête SQL pour la Récupération du Nombre de Publications par Tag avec Réaction Emoji Spécifique

Cette requête SQL implique une fonction d'agrégation avec une clause GROUP BY et une clause HAVING. Elle sélectionne le nom du tag et le nombre de publications associées à chaque tag. Elle n'inclut que les tags ayant un nombre de publications supérieur ou égal au nombre minimum de publications spécifié et ayant la réaction emoji spécifiée.

```
SELECT
    T.name,
    COUNT(DISTINCT P.id) AS Post_Count
FROM
    Tags T
    JOIN PostTags PT ON T.id = PT.tag_id
    JOIN Posts P ON P.id = PT.post_id
    JOIN Reactions R ON R.post_id = P.id
WHERE
    R.emoji = :emoji
GROUP BY
    T.name
HAVING
    COUNT(DISTINCT P.id) >= :minpostcount;
```

6.8 Requête SQL pour le Calcul de la Moyenne du Nombre Maximum de Réactions par Publication

Cette requête SQL calcule la moyenne du nombre maximum de réactions par publication. La moyenne est calculée en groupant d'abord les réactions par post_id et en comptant le nombre de réactions par publication. Le nombre maximum de réactions par publication est ensuite calculé en groupant les comptes par post_id et en prenant le compte maximum. Enfin, la moyenne des réactions maximum par publication est calculée en prenant la moyenne des comptes maximums.

```
SELECT
    AVG(Max_Reactions) AS Average_of_Max_Reactions
FROM (
    SELECT
        post_id,
        MAX(reactions_count) AS Max_Reactions
    FROM (
        SELECT
            post_id,
            COUNT(*) AS reactions_count
        FROM
            Reactions
        GROUP BY
            post_id) AS ReactionCounts
    GROUP BY
        post_id) AS MaxCounts;
```

6.9 Requête SQL pour la Récupération des Utilisateurs avec un Type de Participation Spécifique dans des Événements avec un Statut Spécifique

Cette requête SQL implique au moins une jointure externe. Elle récupère les utilisateurs ayant un type de participation spécifié dans des événements avec un statut spécifique. Elle utilise une jointure LEFT JOIN pour inclure les utilisateurs qui peuvent ne pas avoir de dossiers de participation. La requête joint les tables Users, Participation et Events pour récupérer les informations requises.

```
SELECT
    U.username,
    P.type_participation,
    E.name
FROM
    Users U
    LEFT JOIN Participation P ON U.id = P.user_id
        AND P.type_participation = :typeparticipation
    LEFT JOIN Events E ON P.event_id = E.id
        AND E.status = :eventstatus
WHERE
    U.username IS NOT NULL
    AND P.type_participation IS NOT NULL
    AND E.name IS NOT NULL;
```

6.10 Requête SQL pour la Récupération des Utilisateurs Ayant Réagi à Toutes les Publications d'un Utilisateur Spécifié

Cette requête SQL exprime une condition de totalité avec des sous-requêtes corrélées. Elle récupère les utilisateurs qui ont réagi à toutes les publications d'un utilisateur spécifié. Elle utilise une sous-requête corrélée pour vérifier s'il n'y a aucune publication à laquelle l'utilisateur n'a pas réagi.

```
SELECT DISTINCT
    R.user_id
FROM
    Reactions R
WHERE
```

```
    NOT EXISTS (
        SELECT
            P.id
        FROM
            Posts P
        WHERE
            P.user_id = :userid
            AND NOT EXISTS (
                SELECT
                    1
                FROM
                    Reactions R2
                WHERE
                    R2.user_id = R.user_id
                    AND R2.post_id = P.id));
```

6.11 Requête SQL pour la Récupération des Utilisateurs Ayant Réagi à Toutes les Publications d'un Utilisateur Spécifié avec une Fonction d'Agrégation

Cette requête SQL exprime une condition de totalité avec une fonction d'agrégation. Elle récupère les utilisateurs qui ont réagi à toutes les publications d'un utilisateur spécifié.

```
SELECT
    R.user_id
FROM
    Reactions R
    JOIN Posts P ON R.post_id = P.id
WHERE
    P.user_id = :userid
GROUP BY
    R.user_id
HAVING
    COUNT(DISTINCT P.id) =(
        SELECT
            COUNT(*)
        FROM
            Posts
        WHERE
            user_id = :userid);
```

6.12 Requête SQL pour Récupérer les Identifiants des Publications qui Ne Sont Pas des Réponses à d'Autres Publications

Cette requête retourne les identifiants des publications qui ne sont pas des réponses à d'autres publications. En raison des valeurs NULL dans la colonne parent_post_id, une approche différente est nécessaire.

```
    SELECT
    P1.id
FROM
    Posts P1
EXCEPT
SELECT
    P2.parent_post_id
FROM
    Posts P2;
```

6.13 Requête SQL pour Récupérer les Identifiants des Publications qui Ne Sont Pas des Réponses à d'Autres Publications

Cette requête retourne les identifiants des publications qui ne sont pas des réponses à d'autres publications. En raison des valeurs NULL dans la colonne parent_post_id, une approche différente est nécessaire.

```
    SELECT DISTINCT
    P1.id
FROM
    Posts P1
WHERE
    NOT EXISTS (
        SELECT
            *
        FROM
            Posts P2
        WHERE
            P1.id = P2.parent_post_id);
```

6.14 Requête SQL pour Récupérer les Identifiants des Publications qui Ne Sont Pas des Réponses à d'Autres Publications

6.14.1 Approche Incorrecte

Cette requête retourne les identifiants des publications qui ne sont pas des réponses à d'autres publications. En raison des valeurs NULL dans la colonne parent_post_id, une approche différente est nécessaire.

```
SELECT DISTINCT
    P1.id
FROM
    Posts P1
WHERE
    P1.id NOT IN (
        SELECT
            P2. parent_post_id
        FROM
            Posts P2);
```

6.14.2 Approche Correcte

La solution consiste à utiliser une sous-requête pour filtrer les valeurs parent_post_id qui ne sont pas NULL.

```
SELECT DISTINCT
    P1.id
FROM
    Posts P1
WHERE
    P1.id NOT IN (
        SELECT
            P2. parent_post_id
        FROM
            Posts P2
        WHERE
            P2. parent_post_id IS NOT NULL);
```

6.15 Requête SQL pour Récupérer le Prochain Événement Planifié

Cette requête retourne le prochain événement planifié à partir de la table 'Events'. Elle utilise une CTE récursive pour générer une séquence de dates à partir d'aujourd'hui. La requête joint ensuite cette

séquence avec la table 'Events' pour trouver le prochain événement planifié. Le résultat inclut le nom de l'événement, la date et l'heure complètes, ainsi que le statut.

```
WITH RECURSIVE DateSeq AS (
        SELECT
            NOW()::date AS event_date -- Start from today, only the date part
        UNION ALL
        SELECT
            (event_date + INTERVAL '1 day')::date
        FROM
            DateSeq
        WHERE
            event_date <(
                SELECT
                    MAX(date)::date
                FROM
                    Events)
                -- Up to the max date only, ignoring time
    )
    SELECT
        COALESCE(E.name, 'No event scheduled') AS event_name,
        E.date AS event_full_datetime,
        E.status
    FROM
        DateSeq
        LEFT JOIN Events E ON DateSeq.event_date = E.date::date -- Compare only the date parts
    WHERE
        DateSeq.event_date >= NOW()::date
        AND E.status = 'Scheduled'
        AND E.id IS NOT NULL
    ORDER BY
        DateSeq.event_date ASC
    LIMIT 1;
```

6.16 Requête SQL pour Récupérer les 10 Meilleurs Événements avec le Plus de Participants en une Année Donnée

Cette requête retourne les 10 événements avec le plus de participants pour une année donnée. Elle utilise une fonction de fenêtre pour classer les événements par le nombre de participants en ordre décroissant. Le résultat inclut le nom de l'événement, la date et le rang. La requête filtre les événements par l'année spécifiée et ordonne les résultats par rang et date.

```
SELECT DISTINCT
    E.name,
    E.date,
    RANK() OVER (PARTITION BY EXTRACT(MONTH FROM E.date) ORDER BY COUNT(P.user_id) DESC) AS
        Rank
FROM
    Events E
    JOIN Participation P ON E.id = P.event_id
GROUP BY
    E.name,
    E.date
HAVING
    EXTRACT(YEAR FROM E.date) = :year
ORDER BY
    Rank ASC,
    E.date ASC
LIMIT 10;
```

6.17 Requête SQL pour Récupérer le Nombre de Publications par Utilisateur

Cette requête récupère le nombre de publications faites par chaque utilisateur. Le résultat inclut le nom d'utilisateur et le nombre total de publications.

```
SELECT
    U.username,
    PostCount.PostsCount
FROM
    Users U
    JOIN (
        SELECT
            user_id,
            COUNT(*) AS PostsCount
        FROM
            Posts
        GROUP BY
            user_id) PostCount ON U.id = PostCount.user_id;
```

6.18 Requête SQL pour Trouver Tous les Sous-genres d'un Genre Parent Spécifié

Cette requête utilise une CTE récursive pour trouver tous les sous-genres d'un genre parent spécifié. Le résultat inclut l'identifiant du genre, le nom et l'identifiant du genre parent.

```
WITH RECURSIVE GenresCurrent AS (
        SELECT
            id,
            name,
            parent_genre_id
        FROM
            Genres
        WHERE
            id = :pgid
        UNION ALL
        SELECT
            G.id,
            G.name,
            G.parent_genre_id
        FROM
            Genres G,
            GenresCurrent GC
        WHERE
            GC.id = G.parent_genre_id
    )
    SELECT
        *
    FROM
        GenresCurrent;
```

6.19 Requête SQL pour Récupérer les Villes Hébergeant au Moins un Événement avec des Films de Plus de 1h30

Cette requête retourne les villes qui hébergent au moins un événement avec des films de plus de 1h30. Elle calcule la durée moyenne des films projetés dans ces villes. Le résultat inclut le nom de la ville et la durée moyenne des films.

```
SELECT
    CI.name AS City_Name,
    AVG(CAST(M.duration AS real)) AS Average_Duration
FROM
```

```
    Screenings S
    JOIN Movies M ON S.movie_id = M.id
    JOIN Events E ON S.event_id = E.id
    JOIN Cities CI ON E.city_code = CI.city_code
WHERE
    M.duration > 90
GROUP BY
    CI.name;
```

6.20 Requête SQL pour Récupérer les Événements Futurs avec Plus de 10 Participants par Mois

Cette requête récupère les événements futurs ayant plus de 10 participants par mois. Le résultat inclut le nom de l'événement, la date et le nombre de participants.

```
SELECT
    E.name AS Event_Name,
    E.date AS Event_Date,
    COUNT(*) AS Participants_Count
FROM
    Events E
    JOIN Participation P ON E.id = P.event_id
WHERE
    E.date > CURRENT_DATE
GROUP BY
    E.name,
    E.date
HAVING
    COUNT(*) > 10;
```

7 Recommandations

Les méthodes utilisées pour la recommandation de films, de posts et d'évènements sont identiques sur le plan technique. En effet, les recommandations sont basées sur les goûts des utilisateurs, déterminés par les films qu'ils ont notés, les posts qu'ils ont aimés et les évènements auxquels ils ont participé.

Exemple : Recommandation de films

Tout d'abord, nous calculons la similarité entre les utilisateurs. Par exemple, les similarités entre les films sont calculées en utilisant une formule basée sur le produit scalaire des notes des utilisateurs. Cela implique de comparer les notes données par les mêmes utilisateurs pour différents films.

Dans le cas de la recommandation de films, pour assurer que les similarités calculées sont significatives, seules les paires de films notées par plus d'un utilisateur sont retenues. Cela élimine les cas où la similarité serait basée sur des données insuffisantes.

Les scores de recommandation pour chaque utilisateur et chaque film non encore noté sont ensuite calculés en combinant les similarités des films notés avec les notes données par l'utilisateur. Cela se fait via une somme pondérée des similarités et des notes. Les films déjà notés par l'utilisateur sont exclus de ce calcul pour ne recommander que des films nouveaux.

Les recommandations calculées sont ensuite insérées dans une table permanente *MovieRecommendation*. En cas de conflit (si une recommandation existe déjà pour un utilisateur et un film), la recommandation est mise à jour avec le nouveau score calculé.

8 Conclusion

Pour conclure, nous avons pu démontrer de la robutesse de notre base de données avec les 20 requêtes SQL que nous avons présentées. Ces requêtes couvrent un large éventail de cas d'utilisation, allant de la récupération de données simples à des requêtes plus complexes impliquant des jointures, des sous-requêtes, des CTE récursives et des fonctions de fenêtrage. Nous avons également fourni des solutions

alternatives pour certaines requêtes, en mettant en évidence les erreurs courantes et en proposant des approches correctes. Enfin, nous avons fourni des explications détaillées pour chaque requête, en mettant en évidence les concepts SQL sous-jacents et en expliquant les raisons de chaque étape.