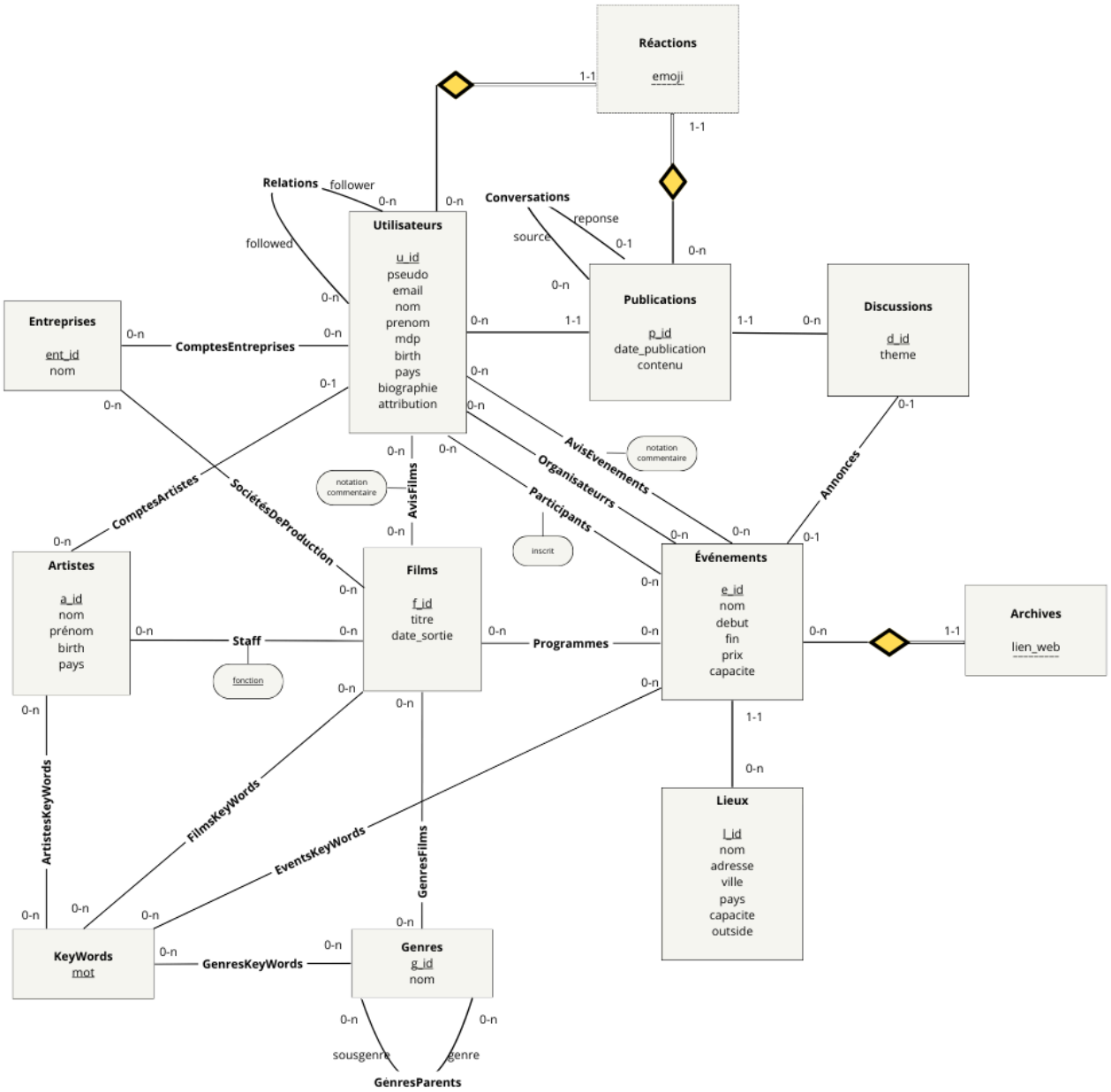
PROJET BASES DE DONNÉES (BD6)

Ayadi Jana (22114164) et Baha Manon (22004953)

I. Diagramme entités - associations



II. Schéma relationnel

Utilisateurs (u_id, pseudo, email, nom, prenom, mdp, birth, pays, biographie, attribution)
Artistes (a_id, nom, prenom, birth, pays)
Entreprises (ent_id, nom)
Films (f_id, titre, date_sortie)
Genres (g_id, nom)
Discussions (d_id, nom)
Lieux (l_id, nom, adresse, ville, pays, capacite, outside)
Evenements (e_id, nom, debut, fin, prix, #lieu, capacite)
Publications (p_id, #u_id, #d_id, date_pubilcation, contenu)
Reactions (#u_id, #p_id, emoji)
Archives (#e_id, lien_web)
AvisFilms (#f_id, #u_id, mark, commentaire)
AvisEvenements (#e_id, #u_id, mark, commentaire)
GenresFilms (#f_id, #g_id)
Staff (#f_id, #a_id, fonction)
SocietesDeProduction (#f_id, #ent_id)
Programmes (#e_id, #f_id)
Participants (#e_id, #u_id, inscrit)
Organisateurs (#e_id, #organisateur)
Annonces (#e_id, #d_id)
GenresParents (#genre, #sousgenre)
Relations (#follower, #followed)
Conversations (#source, #reponse)
ComptesArtistes (#a_id, #u_id)
ComptesEntreprises (#ent_id, #u_id)
KeyWords (mot)
ArtistesKeyWords (#a_id, #mot)
FilmsKeyWords (#f_id, #mot)
EventsKeyWords (#e_id, #mot)
GenresKeyWords (#g_id, #mot)

III. Description des tables

Dans cette partie, nous allons décrire la manière dont nous avons imaginé la base de données du site CinémaNet. Notons que nous ne détaillerons ci-dessous que les fonctionnalités non usuelles et les situations ambigües que nous avons rencontrées, le reste ne nécessitant pas d'explication particulière.

Les utilisateurs (table **Utilisateurs**) ont un rôle, qui décrit les permissions qui leur sont attribuées sur CinémaNet. Dans le code, celà se traduit de la manière suivante : Nous avons créé un type *attributions* correspondant à un enum, et chaque utilisateur possède une attribution.

Conformément aux attentes du cahier des charges, la relation Follow et la relation d'amitié sont différentes, mais néanmoins corrélées. En effet, à la manière d'Instagram, deux utilisateurs sont amis s'ils se suivent mutuellement. Il n'y a donc pas de table Amitié à proprement parler mais elle se déduit de la présence éventuelle de tuples symétriques dans la table **Follow**. Nous le vérifions dans la partie requêtes.

Évidemment, notre base de données possède plusieurs entités relatives aux professionnels du cinéma. Afin d'éviter la redondance des informations (Une même personne pouvant à la fois être acteur et réalisateur), nous avons choisi de faire une seule table **Artistes**. C'est-à-dire, de ne pas représenter individuellement les entités filles. Ces artistes sont reliés aux films dans lesquels ils ont participé par une association qui indique également la fonction qu'ils ont tenu. (table **Staff**)

Un artiste peut aussi éventuellement être rattaché à un compte utilisateur. Pour celà nous avons créé une table intermédiaire entre les tables Utilisateurs et Artistes : **ComptesArtistes**. Ainsi, les utilisateurs pourront facilement retrouver les comptes de leurs personnalités préférées.
La conception est similaire pour les Entreprises (Gaumont-Pathé…) : elles peuvent être associées aux films dont elles ont participé à la production et être associées à un compte utilisateur. (table **ComptesEntreprises**).

Notre site possède plusieurs espaces de discussions :
Si les utilisateurs veulent émettre un commentaire sur un film en particulier, ils peuvent se rendre dans l'espace **Avis** dédié aux films.
Pour des conversations plus générales, ils peuvent se rendre dans l'espace **Discussions**. Ces discussions sont organisées à la manière des discussions de Discord ou encore WhatsApp. Une discussion possède un nom et est composée de **Publications**.

Penchons-nous sur la généalogie des publications : la relation "réponse à une publication". Nous avons vite compris qu'il s'agissait d'une sorte de récursivité. Tout d'abord, nous avions eu comme idée de mettre dans la table Publications un attribut référençant l'id d'une autre publication et qui serait NULL si la publication est de premier niveau. Problème : NULL n'est pas une valeur présente dans les id de publications et l'intégrité référentielle de nos tables aurait alors été discutable. Nous avons donc construit une table qui stocke les couples (source, réponse). La table **Conversations**. Nous verrons dans les requêtes comment cette table peut être utilisée.

Intéressons nous désormais à la généalogie des genres. Pour la même raison citée dans le paragraphe sur la généalogie des publications, nous avons créé une table liant les couples (genre, sous genre). La table **GenresParents**. Cependant, il est ici compliqué de vérifier la véracité d'une telle relation. Par exemple, s'il y a les couples (x,y) et (y,z) alors il ne peut logiquement pas y avoir le couple (z,x). Il s'agit d'une contrainte externe.

Passons maintenant au dernier élément principal du site : les événements. L'un de nos grands questionnements durant ce projet s'est posé sur cette table. Nous avons hésité entre créer deux tables distinctes *Événements passés* et *Événements futurs* ou bien une seule table **Événements**. Plusieurs inconvénients sont apparus avec la première solution. Entre autres, une fois l'événement dépassé, il aurait fallu transférer automatiquement les informations de la table *Événements futurs* vers la table *Événements passé*s, ce que nous ne savions pas faire. Qui plus est, ces deux entités possédant des attributs similaires, nous avons opté pour la deuxième solution. Pour satisfaire certaines contraintes d'intégrité (inscription à des événements futurs uniquement, archives associées à des événements passés uniquement..) nous utilisons des triggers. (Nous détaillerons les triggers plus loin dans ce rapport).

Enfin, voici la manière dont nous avons pensé les mots clés. Un mot clé peut être associé à un artiste, un film, un genre et à un événement. Ainsi, si un utilisateur rentre un mot dans la barre de recherche, nous pourrions lui proposer les objets reliés à ce mot. Nous n'avons pas jugé nécessaire d'associer des mots clés aux publications car à partir d'un mot donné, il suffit de sélectionner les publications qui possèdent ce mot dans leur contenu. Nous avons créé une table générale **KeyWords**, puis les quatre tables associatives **ArtistesKeyWords**, **FilmsKeyWords**, **EventsKeyWords**, **GenresKeyWords**.

IV. Intégrité, contraintes externes, stabilité des tables et utilisation des trigger

Dans cette section, nous détaillerons les divers éléments pris en compte durant le projet pour assurer l'intégrité et la stabilité des tables. Les contraintes UNIQUE et NOT NULL ne sont pas détaillées ici mais dans le fichier des tables. Nous aborderons également les principales contraintes externes. Nous avons quelques fois utilisé les triggers, bien que hors-programme, cette fonctionnalité s'est avérée être la manière la plus propre de vérifier plusieurs restrictions que nous avions fixées.

Contraintes d'intégrité:

- La table **Programmes** ne concerne que les événements futurs. Pour celà nous avons créé une fonction de trigger sur la table Évènements puis appelé cette fonction dans un trigger sur la table Programme.

- La table **Archives** ne concerne que les événements passés. De la même manière, nous avons créé une fonction de trigger sur la table Évènements puis appelé cette fonction dans un trigger sur la table Archives.

- Nous vérifions que la capacité de l'événement est inférieure ou égale à la capacité du lieu (table **Lieux**) où se tient l'événement grâce à un trigger.

- De la même manière, il ne peut pas y avoir plus d'inscrits à un événement que la capacité spécifiée pour cet événement. Nous avons donc fait un trigger sur la table **Participants** pour vérifier cette exigence.

- Lors de la création du compte d'un artiste (table **ComptesArtistes**), on vérifie que les attributs présents dans Utilisateurs et dans Artistes sont bien cohérents grâce à un trigger.

- Dans la table **Conversations**, nous vérifions que les publications *source* et *reponse* font bien partie de la même discussion grâce à un trigger.

- Deux événements ne peuvent pas se dérouler sur le même lieu si leurs dates se superposent. Pour vérifier celà nous avons créé un trigger sur la table **Évènements**, en se servant de la fonction *daterange* de sql.

- Nous ajoutons la contrainte (grâce à un trigger) qu'un utilisateur ne peut mettre un avis à un événement (table **AvisEvenements**) seulement s' il était inscrit à l'événement et que cet événement est passé.

- La table **Conversations** est anti-symétrique et non réflexive. Les publications étant logiquement ajoutées dans un ordre chronologique, leur id (qui est serial) suit le même ordre. La contrainte CHECK (source < réponse) permet donc de satisfaire toutes ces conditions.La table **Relations** n'est ni symétrique ni antisymétrique mais elle est non-réflexive : on ne peut pas être ami avec soi-même. Nous avons vérifié ça grâce à un check.

- De manière plus évidente, nous avons vérifié avec des check que la capacité dans Lieux et Evenements est bien supérieure ou égale à 0, que le prix est bien supérieur ou égal à 0 dans Evenements. Aussi, que la date du début de l'événement est bien inférieure à la date de fin de l'événement. Enfin, nous avons vérifié que les notations dans AvisFims et AvisEvenements soient bien comprises entre 0 et 5.

- Condition que nous n'avons pas pu vérifier : la table **ParentsGenres** permet de répertorier des couples de genres (x,y) tels que y est un sous genre de x. Cependant, l'ajout des genres dans la table Genres ne suit pas la chronologie, on ne peut utiliser

pas la même méthode que pour la table Conversations. Par exemple, si on a des couples (x,y) et (y,z), il ne peut y avoir ni (y,x) ni (z,x). En revanche, nous avons bien vérifié qu'il ne peut pas couples de la forme (x,x).

Stabilité des tables :

- Nous avons ajouté des ON DELETE CASCADE ou ON DELETE SET NULL sur plusieurs attributs ayant une contrainte de clé étrangère. Par exemple, on veut qu'un utilisateur puisse supprimer une publication sans que les réponses à celles-ci soient supprimées, comme c'est le cas sur Discord. (ON DELETE SET NULL).

V. Requêtes

pas demandées mais intéressantes :

1. Les couples d'utilisateurs qui sont amis. Sans doublons ni couples symétriques et avec création d'une vue. Cette vue est indispensable pour les autres requêtes !
2. Les couples d'id d'utilisateurs qui ne sont pas amis mais qui ont au moins un amis en commun.
3. L'utilisateur le plus suivi.

requêtes sur au moins 3 tables :

4. Les films ayant reçu la notation de 5/5 par un utilisateur certifié.

requêtes sur au moins 3 tables et avec autojointure :

5. Les artistes ayant joué dans un film qu'ils ont eux même réalisé. Résultat sous la forme (nom, prénom, titre).

requêtes avec sous requête dans le FROM :

6. Les films diffusés dans des événements à plus de 200€.

requêtes avec sous requête dans le WHERE :

7. Idem mais avec une sous requête dans le WHERE.

requêtes avec sous requête corrélée :

8. Les utilisateurs ayant fait deux publications à moins d'une minute d'intervalle.

requêtes avec GROUP BY et HAVING :

9. Les publications qui comptent au moins 5 likes.
10. Les films dont la moyenne des notes est supérieure à la moyenne générale de tous les films confondus.

requêtes nécessitant le calcul de deux agrégats :

11. La moyenne des notes maximales attribuées aux films.

requêtes avec une jointure externe :

12. La liste de toutes les entreprises, et s'ils existent, les comptes qui leur sont associés.

requêtes équivalentes pour une condition de totalité :

13. Les réalisateurs dont tous les films commencent par 'the', avec une sous requête corrélée.
14. Idem mais avec de l'agrégation.

requêtes ambiguës avec de la valeur NULL

15. 16. 17. L'utilisateur le plus âgé, avec la valeur NULL qui affecte les résultats.
18. 19. 20. Idem mais avec correction des effets de la valeur NULL .

requêtes récursives :

21. Les couples de publications ayant un lien de parenté vertical plus ou moins éloigné
22. Les couples de genres qui ont un lien de parenté vertical plus ou moins éloigné.

requêtes avec du fenêtrage :

23. Pour chaque événement passé, son nombre de participants, comparé à la moyenne des événements ayant eu lieu au même endroit.

VI. Algorithme de recommandations

Nous avons implémenté un algorithme de recommandations pour les films, qui repose sur la méthode suivante :

À partir de critères choisis, nous construisons les tables des films qui pourraient potentiellement plaire à l'utilisateur. C'est-à-dire, les films des artistes auxquels il est abonné, les films diffusés lors d'événements qui ont plu à ses amis… Évidemment cette liste est non-exhaustive et nous pouvons ajouter autant de critères que voulus.

Ensuite, pour accorder l'indice de recommandation correspondant, nous regardons le nombre de critères que valide chaque film.

Notons que les critères peuvent avoir des poids différents, selon leur proximité avec l'utilisateur en question et la probabilité qu'il lui plaise réellement.

Cet algorithme est facilement adaptable à la recommandation d'événements ou de publications.

Voici l'algorithme en question :

```
PREPARE recommandations(INT) AS

WITH
followed_artists AS
(
    SELECT C.a_id
    FROM ComptesArtistes C, Relations R
    WHERE R.followed = C.u_id
    AND R.follower = $1
),
films_liked_by_friends AS
(
    SELECT F.f_id
    FROM Films F NATURAL JOIN AvisFilms A
    WHERE A.u_id IN (SELECT id1 FROM Amis WHERE id2 = $1)
    OR A.u_id IN (SELECT id2 FROM Amis WHERE id1 = $1)
    AND A.notation >= 3
),
films_from_events_liked_by_friends AS
(
    SELECT F.f_id
    FROM Films F NATURAL JOIN Programmes P NATURAL JOIN Evenements E NATURAL JOIN AvisEvenements A
    WHERE A.u_id IN (SELECT id1 FROM Amis WHERE id2 = $1)
    OR A.u_id IN (SELECT id2 FROM Amis WHERE id1 = $1)
    AND A.notation >= 4
),
films_with_followed_artists AS
(
    SELECT F.f_id
    FROM Films F NATURAL JOIN Staff S
    WHERE S.a_id IN (SELECT a_id FROM followed_artists)
),
films_recommandes AS
(
SELECT F.f_id,
       F.titre,
        (CASE
            WHEN (F.f_id IN (SELECT f_id FROM films_liked_by_friends)) THEN 1.0
            ELSE 0.0
        END +
        CASE
            WHEN (F.f_id IN (SELECT f_id FROM films_from_events_liked_by_friends)) THEN 0.5
            ELSE 0.0
        END +
        CASE
            WHEN (F.f_id IN (SELECT f_id FROM films_with_followed_artists)) THEN 2.0
            ELSE 0.0
        END) AS indice
FROM Films F
)

SELECT f_id,titre, indice
FROM films_recommandes
WHERE indice <> 0;
```