Projet Bases de Données (BD6)
Groupe 30: Rapport

Tous les documents mentionnés sont présents et visualisables dans l'archive.

I) Modèle et Diagramme entités-associations :

a. Specialisation

Schema_Pre_Heredite.png

Nous avons tout d'abord réalisé un premier diagramme E/R avec représentation des héritages. Ce premier modèle présentait beaucoup de spécialisations à supprimer, principalement au niveau de la table Utilisateur et de ses tables filles. De même récursivement, pour les tables filles de la table Organisation, fille d'Utilisateur.
Après réflexion, nous avons trouvé un moyen de suppression des hérédités économe en nombre de tables et raisonnable en terme de perte d'information et de contraintes.

a. Restructuration du diagramme E/R «

Diagramme_UML_Final.drawio.png

Chaque Utilisateur peut s'identifier comme un Artiste, chaque Arstite peut avoir joué dans des Films ce qui fait de lui un acteur et peut également en avoir réalisé ce qui fait de lui un réalisateur. Est cinéphile l'Utilisateur lambda qui n'a ni joué ni réalisé, et on procède à l'élimination des filles de la classe Organisation qu'on renomme Groupe avec un simple attribut type pour indiquer à quel type de groupe appartient l'utilisateur. Les autres cas d'hérédité sont plus triviaux à résoudre. La table Emoji Fille de la classe réaction qui devient une liaison 1-1 à 1-n de Reaction vers Emoji et la table Jaime disparaît, le jaime devient un type d'emoji parmi d'autres. La table Evenement_Passe fille de la table Evenement devient une liaison 0-1 à 1-1.
Une fois la restructuration terminée, on améliore encore notre diagramme sur quelques points.
La table Amis disparaît car il suffit de modéliser le concept d'amis comme deux personnes qui se follow mutuellement, la table Participe disparaît car nous avons trouvé une solution à une seule table pour régler la contrainte externe voulant qu'un utilisateur doit choisir entre être intéressé et participer à coup sûr. Enfin, certaifns attributs de certaines tables sont ajoutés ou enlevés ou renommés selon la pertinence.

Toutes les explications de choix de modélisation finaux sont détaillés dans le document :

ExplicationDiagrammeUML.odt

II) Traduction en Schéma Relationnel:

Utilisateur

Utilisateur(login, pseudo, mdp, anniversaire, #ID_Artiste)
Utilisateur[#ID_Artiste] ⊆ Artiste[ID_Artiste]

Publication

Publication(ID_Publication, texte , sujet1, sujet2 , sujet3 , date_p, #auteur, #categorie)
Publication[#auteur] ⊆ Utilisateur[login]
Publication[#categorie] ⊆ Categorie[nom]

Categorie

Categorie(nom)

Commentaire

Commentaire (ID_Commentaire , texte , #auteur, #origine_p , #origine_c)
Commentaire[#auteur] ⊆ Utilisateur[login]
Commentaire[#origine_p] ⊆ Publication[ID_Publication]
Commentaire[#origine_c] ⊆ Commentaire[ID_Commentaire]

Réaction

Réaction (#ID_Utilisateur , #ID_Publication , #ID_Emoji)
Réaction[#ID_Utilisateur] ⊆ Utilisateur[login]
Réaction[#ID_Publication] ⊆ Publication[ID_Publication]
Réaction[#ID_Emoji] ⊆ Emoji[ID_Emoji]

Emoji

Emoji(ID_Emoji , nom)

Sujet

Sujet(nom)

Hashtag

Hashtag(#ID_Publication , #nom_Sujet, #date_h)
Hashtag[#ID_Publication] ⊆ Publication[ID_Publication]
Hashtag[#nom_Sujet] ⊆ Sujet[nom]
Hashtag[#date_h] ⊆ Publication[date_p]

Evenement

Evenement(ID_Evenement , titre , lieu , date_e , prix , nbPlaces ,#annonce, #organisateur)
Evenement[#annonce] ⊆ Publication[ID_Publication]
Evenement[#organisateur] ⊆ Utilisateur[login]

Artiste

Artiste(ID_Artiste, nom, sexe, nb_Oscars)

Evenement_Passe

Evevenement_Passe(ID_Archive, programme, nb_Participants, lien, #ID_Evenement)Evenement_Passe[#ID_Evenement] ⊆ Evenement[ID_Evenement]

Interesse

Interesse(#ID_Evenement , #ID_Utilisateur , participe)
Interesse[#ID_Evenement] ⊆ Evenement[ID_Evenement]
Interesse[#ID_Utilisateur] ⊆ Utilisateur[login]

Film

Film(titre, #ID_Realisateur , nb_Oscars, nb_Entrees, annee, #genre)
Film[#ID_Realisateur] ⊆ Artiste[ID_Artiste]
Film[#genre] ⊆ Genre[nom]

Acteur

Acteur(#titre , #realisateur, #acteur)
Acteur[#titre] ⊆ Film[titre]
Acteur[#realisateur] ⊆ Film[ID_Realisateur]
Acteur[#acteur] ⊆ Artiste[ID_Artiste]

Genre

Sous_Genre(nom, #parent1, #parent2)
Sous_Genre[#parent1] ⊆ Genre[nom]
Sous_Genre[#parent2] ⊆ Genre[nom]
Genre(nom)

Groupe

Groupe(ID_Groupe, nom, type_g, #createur)
Groupe[#createur] ⊆ Utilisateur [login]

Membres

Membres(#groupe, #login)
Membres[#groupe] ⊆ Groupe [ID_Groupe]
Membres[#login] ⊆ Utilisateur [login]

Follow

Follow(#suivi , #suiveur)
Follow[#suivi] ⊆ Utilisateur [login]
Follow[#suiveur] ⊆ Utilisateur [login]

Historique

Historique(#utilisateur , #sujet, date)
Historique[#ID_Utilisateur] ⊆ Utilisateur [login]
Historique[#ID_Sujet] ⊆ Sujet [nom]

III) Explication des choix et traduction en sql:

ExplicationDiagrammeUML.odt

Plusieurs types de contraintes externes devaient être respectées au niveau de la base de donnée.

1. **Les contraintes de clé primaire**

Nous les avons indiquées précédemment en soulignant les attributs concernés.
Pour choisir ce qui constituera la clé primaire d'une table, on se demande ce qui permet d'identifier un tuple par rapport à un autre.

Exemple:

Table Hashtag :

Il impossible de mettre 2 fois le même hashtag dans une publication, on met donc comme clé primaire le couple #ID_Publication, #nom_Sujet .
Si nous avions choisi d'inclure l'attribut #date à la clé primaire, alors il aurait été possible qu'une même publication mentionne le même sujet à 2 dates différentes ce qui serait absurde.

On implémentera cette contrainte simplement en indiquant :

```
PRIMARY KEY (ID_Publication, nom_Sujet),
```

2. **Les contraintes de clés étrangères**

Nous les avons indiquées précédemment en mettant un # devant les attributs concernés.
On détaille le choix de chaque clé étrangère dans la partie en rouge du schéma.

Exemple :

Table Membres :

Membres(#groupe, #login)
Membres[#groupe] ⊆ Groupe [ID_Groupe]
Membres[#login] ⊆ Utilisateur [login]

On implémentera cette contrainte simplement en indiquant :

```
FOREIGN KEY (groupe) REFERENCES projet.Groupe(ID_Groupe) ON DELETE CASCADE ON UPDATE CASCADE,
FOREIGN KEY (logins) REFERENCES projet.Utilisateur(logins) ON DELETE CASCADE ON UPDATE CASCADE
```

On indiquera pour chaque clé étrangère ON UPDATE CASCADE qui assure que si une ligne dans la table référencée (la table parent) est mise à jour, toutes les lignes correspondantes dans la table qui contient la clé étrangère (la table enfant) sont automatiquement mises à jour pour refléter cette modification.

De même avec ON DELETE CASCADE qui assure que si une ligne dans la table référencée est supprimée, toutes les lignes correspondantes dans la table contenant la clé étrangère sont également supprimées.

3. Les contraintes de non Nullité

Certains attributs ne font pas parti de la clé primaire mais on ne souhaite pas qu'ils puissent être NULL pour autant.

Exemple:

Table Reaction :

L'attribut #ID_Emoji n'a pas été mis dans la clé primaire car on ne souhaite pas qu'un utilisateur puisse réagir 2 fois à une publication avec un emoji différent. Cependant,chaque réaction doit obligatoirement avoir un émoji pour exister, on décide donc de mettre cet attribut en NOT NULL.

On implémentera cette contrainte simplement en indiquant :

```
ID_Emoji INTEGER NOT NULL,
```

4. Les contraintes d'Unicité

Certains attributs ne font pas parti de la clé primaire mais on ne souhaite pas qu'ils puissent être partagés par d'autres tuples de la table pour autant.

Exemple:

Table Evenement_Passe :

L'attribut #ID_Evenement n'a pas été mis dans la clé primaire car cela voudrait dire qu'un même évènement peut être archivé par 2 évènements passés. Pour s'assurer que 2 tuples de Evenement_passe n'archivent pas le même évènement, on met cet attribut en Unique.

On implémentera cette contrainte simplement en indiquant :

```
UNIQUE (ID_Evenement)
```

5. Les contraintes de vérification

Certains attributs doivent pouvoir respecter certaines conditions.

Exemples:

Table Film :

L'attribut nb_Entrees ne peut pas être un nombre négatif.

On implémentera cette contrainte simplement en indiquant :

```
nb_Entrees INT CHECK (nb_Entrees >=0),
```

Table Groupe :

L'attribut type doit se référer à des données précises, ici on limite les types possibles à «club », « presse », « studio », « festival » et « cinema ».

On implémentera cette contrainte simplement en indiquant :

```
type_g VARCHAR(30) CHECK (type_g IN ('club','presse', 'studio', 'festival', 'cinema')),
```

IV) Peuplement des tables:

a. Création de la base de données

creation/traduction.sql

Une fois le schéma relationnel terminé, on crée nos tables dans le fichier traduction.sql en prenant soin d'implémenter toutes les contraintes externes nécessaires comme indiqué précédemment.
On utilise en priorité des attribut de type Integer pour les chiffres qui sont des ID et INT pour les autres.
On décide d'utiliser majoritairement des attributs de type Varchar pour les différentes chaînes de caractères et de limiter leur taille à 30.
On privilégie le type TIMESTAMP pour les dates puisqu'on veut pouvoir les ranger de façon précise.

b. remplissage des tables

creation/script/remplissage.sql

Pour alimenter nos tables avec des données, nous avons utilisé un script Python pour générer des ensembles de données variés. Ces ensembles ont été ajustés selon nos besoins spécifiques tout en respectant les contraintes définies dans notre schéma de base de données. Chaque ensemble de données a ensuite été sauvegardé dans un fichier CSV distinct, organisé dans le répertoire CSV/ressources.

Le script remplissage.sql, localisé dans le dossier script, nous permet d'importer facilement ces fichiers CSV dans nos tables correspondantes. En utilisant COPY, nous transférons les données du fichier CSV vers la base de données, assurant ainsi un processus de chargement efficace et cohérent.

Ce processus nous offre la flexibilité nécessaire pour ajuster et enrichir nos ensembles de données en fonction de nos besoins évolutifs. En automatisant cette tâche, nous gagnons du temps et minimisons les risques d'erreurs lors du chargement des données dans notre système.

V) Requetes

creation/requete/req_echo.sql

On remplit les le fichier req_echo.sql avec des requêtes correspondant aux types de requêtes demandées.

Exemple :

```
6. Agrégats nécessitant GROUP BY et HAVING :

Cette requête sélectionne les artistes ayant remporté plus de 9 Oscars, en affichant leur identifiant et le nombre maximal d'Oscars remporté.
\! echo Requête 6 : artistes ayant remporté plus de 9 Oscars;
SELECT ID_Artiste, MAX(nb_Oscars) AS mx_oscars
FROM projet.Artiste
GROUP BY ID_Artiste
HAVING MAX(nb_Oscars) > 9
ORDER BY ID_Artiste DESC;
```

VI) Recommandation:

creation/requete/req_echo.sql

On cherche à inclure au fichier certaines requêtes qui pourront participer au développement d'un algorithme de recommandation :

```
14. Récupérer les événements auxquels un utilisateur est intéressé :

15. Recommandation de publications en fonction des sujets consultés par l'utilisateur :

16. Calculer un score de recommandation :

17. Recommander les n meilleurs événements :

18. Recommandation de films basée sur les films aimés par les utilisateurs suivis :

19. Recommandation d'événements en fonction de l'historique des événements suivis par l'utilisateur

20. Recommandation de publications basée sur les publications aimées par l'utilisateur et son réseau :
```

VII) Regard critique sur les choix effectués:

ExplicationDiagrammeUML.odt

Dans certains cas, il n'existe pas de solution parfaite dans les choix de modélisation et d'implémentation d'une base de données, aussi certains des choix que nous avons effectués sont des

parti pris qui méritent d'être critiqués.

Exemples :

Les tables Amis/Follow

Ici, nous avons fait le choix de symboliser la relation d'amitié comme le fait de se suivre mutuellement et en avons profité pour supprimer la table Amis.
Ce choix nous a permis d'alléger la base de données et peut sembler justifiable dans un contexte de réseau social puisque beaucoup d'entre eux fonctionnent également de la sorte.
Cependant, on aurait pu vouloir garder ces 2 types de relations distinctes et implémenter la table amitié de la façon suivante :

```
ALTER TABLE Amis ADD CONSTRAINT asym Check (ami1<ami2)
```

Il aurait alors fallu bien faire attention à bien utiliser des disjonctions dans les requêtes pour tenir compte de cette asymétrie.
On pourra alors également recréer artificiellement la symétrie en utilisant la vue suivante Ami_full à la place de Amis :

```
WITH Ami_full AS
( SELECT * FROM amis UNION SELECT ami1 AS ami2 , ami2 AS ami1 FROM amis)
```

Les tables Interesse/Participe

Ici, nous avons fait le choix de régler la contrainte externe voulant qu'un utilisateur ne pouvait pas se déclarer à la fois intéressé et certain de venir au même évènement, en fusionnant les 2 tables et en ajoutant un attribut booléen. Cette solution nous a semblé optimale dans cette situation puisque les 2 tables fusionnées partageaient exactement les mêmes attributs.

Les tables Réaction/Participe

L'une des plus grosses limites de notre modélisation, réside dans le fait que l'on ne peut réagir qu'aux publication de premiers niveau, mais pas aux commentaires.
Parmi les solutions que nous aurions pu envisager pour régler ce problème, on aurait tout d'abord simplement pu créer une autre table de réaction Reaction_C avec exactement les mêmes propriétés que la table Reaction mais cette fois-ci reliée à la table Commentaire et non Publication. Nous apprécions peu ce genre de solution qui nous semblent manquer d'élégance en distinguant 2 tables qui ne devraient en former qu'une.
On aurait également pu envisager de fusionner les tables commentaires et publication, avec simplement chaque publication se référant à une publication d'origine qu'elle commente, et pouvant être nulle pour les publications de premier niveau. Nous avons également choisi d'écarter cette solution car celle-ci faisait perdre la distinction imposée par l'énoncé selon laquelle seules les publications de premier niveau doivent indiquer leur sujet, ainsi que celle que nous nous étions nous même imposée selon laquelle seule les publications de premier niveau peuvent annoncer un évènement.

Les tables Sujet/Film/Publication

Une autre grande limite de notre modélisation réside dans le fait que toutes les réactions des utilisateurs ne peuvent se référer qu'à une publication. La table sujet elle même n'est d'ailleurs reliée qu'à la table publication. On aurait pu imaginer d'autres tables de transition qui permettent de relier les films aux utilisateurs sans passer par les tables Sujet et Publication