PENELOUX
Nicolas
groupe 29

FLICI
Yacine

COMPTE RENDU

Projet Base de données

NoiseBook

I – Introduction

Voici notre compte rendu concernant le projet de la base de données NoiseBook, le nouveau réseau social centré sur la musique.

II – Schéma de modélisation

Pour des raisons de visibilités, nous ne pouvons pas mettre la modélisation ici, je vous invite donc à consulter l'annexe à la dernière page du document.

Choix de modélisation :

Nous voulions à la base faire un système d'héritage pour Utilisateurs et pour Concert : Pour Utilisateurs, on imaginait que les entités « Personnes » et « Organismes » héritaient de Utilisateurs, et que les entités Concert_fini et Concert_prévu héritaient de Concert. Cependant, pour des raisons que nous ignorons, nous n'avons pas pu implémenté ce système avec PSQL : avec l'héritage, on ne pouvait pas créer les relations de Suivi et d'Amitié, quand on essayait de remplir la table Suivi ou Amis, nous avions le message d'erreur comme quoi la clé étrangère n'était pas présente dans Utilisateurs, alors que pourtant elle y était bien. Nous avons essayé d'autres moyen, en vain. Dans ce cas, nous préférions avoir un modèle imparfait mais plutôt complet, plutôt qu'un modèle incomplet mais plus propre. Dans notre cas, les entités « Personnes », « Organismes, » « Concert_prevu » et « Concert_fini » devront donc avoir une relation d'entité faible avec respectivement Utilisateurs ou Concert, pour permettre leur existence sans qu'il y est des problèmes d'intégrité.
Nous avons fait deux relations à part pour modéliser l'amitié et le système de suivis.

Elles ont la même « structure » cependant l'une est forcément symétrique.
L'entité « Archive » est une entité faible. En effet, dans notre modélisation, on estime qu'une archive existe uniquement s'il existe un concert fini, sinon il n'y a pas d'archive.
De la même façon, Concert est une entité faible, car il est nécessaire d'avoir un lieu pour avoir un concert. Par ailleurs, il faut au moins un artiste pour qu'il y est un concert, mais il peut y avoir n artistes disponibles au concert. L'attribut « index » de la relation LineUp permet de donner un ordre d'apparition des artistes à un concert. Par ailleurs, les LineUp indique quel artiste sera à l'affiche de quel événement, et non quel groupe sera à l'affiche.

Pour les relations Tag et Avis, nous avons décidé de limité le nombre de tag à certaines relations, de même pour avis. De ce fait, on ne peut avoir que des tags pour les groupes de musiques, les concerts les lieux et les playlists, et des avis sur les concert, les morceaux, les lieux, les artistes et les archives. Cependant, nous avons autorisé le fait d'avoirs des tag sur les avis, même s'il n'y a pas forcément de commentaires.

Morceau ne peut pas ne pas avoir de genre, c'est donc également une entité faible.

Playlist est une relation qui est aussi une entité faible, il doit y avoir un propriétaire pour chaque playlist.

Enfin, le concept de sous catégorie est représenté par une relation « Relation_genre » qui aura un attribut parent et un attribut fils. Logiquement, les id de genres qui n'apparaitront jamais dans la colonne « fils » seront donc les parents d'une plus ou moins grande arborescence.

III – Schéma Relationnel et choix

Voici donc le schéma relationnel associé à notre base de donnée :

Utilisateurs(id_user,pseudo,email,pwd,pays,ville,date_creation);
[id_user=INTEGER,pseudo=VARCHAR(30),email=VARCHAR(30),pays=VARCHAR(30),ville=VARCHAR(30),date_creation=DATE]
Personnes(id_personne*,date_naissance)
Organisme(id_orga*,nom)
Suivis(suiveur*,suivi*)
Amis(id_user*,id_ami*)
Concert(id_concert,nom,date_concert,prix,nb_places,volontaires,cause_soutien, enfants_admissibles)
Concert_prevu(nb_places_restantes)
Concert_fini(id_archive*)

Participation(id_personne*,id_concert*,est_interesse)
Organisation(id_user*,id_concert*)
Archive(id_archive,id_concert*,nb_participant,photo,video)
Archive_avis(id_avis*,id_archive*)
Lieu(id_lieu,adresse,ville,code_postal,pays)
Lieu_concert(id_concert*,id_lieu*)
Artiste(id_artiste,id_groupe*,nom)
Groupe(id_groupe,nom,date_creation)
Avis(id_avis,id_type*,type_avis,note,commentaire,date_avis)
Auteur_avis(id_avis*_id_user*)
Playlist(id_playlist,id_user*,nom)
Morceau(id_morceau,nom,id_artiste*,duree)
Contient(id_playlist*,id_morceau*)
Genre(id_genre,nom)
Genre_Morceau(id_genre*,id_morceau*)
Relation_genre(id_parent*,id_fils*)
Tag(id_tag,id_type*,type_tag,valeur)
Lineup(id_concert*,id_artiste*,performance_index)

Les relations en italique sont des relations qui hérite d'une autre relation. Pour Groupe et Personnes, elles héritent de Utilisateurs, et pour Concert_prevu et Concert_fini, elles héritent de Concert.

Les attributs souligné représentent les clés primaires.

Les attributs avec une '*' représentent les clés étrangères :

- Toutes les clés étrangères de id_user, correspondent à id_user dans Utilisateurs.
- Toutes les clés étrangères id_concert correspondent à id_concert dans Concert (sauf celle de la relation Archive, qui correspond à id_concert dans Concert_fini).
- Les clés étrangères « suiveurs » et « suivis » de « Suivis » correspondent à « id_user » dans Utilisateurs, de même pour « id_user » et « id_ami » de Amis.
- « id_personne » de Participation correspond à id_user dans la table Personnes.
- « id_lieu » de Lieu_concert correspond à « id_lieu » dans Lieu.
- « id_parent » et « id_fils » dans Relation_genre correspond à « id_genre » dans Genre.
- « id_orga » dans Organisme référence « id_user » dans Utilisateurs.

- Pour tout les « id_type », cela correspond à des clés étrangères qui peuvent venir de plusieurs tables différentes. En effet, en fonction de la valeur de type_tag, id_type peut ne pas avoir de référence. Pour id_type dans Avis, cela peut référencé les clés primaires des relations suivantes : Morceau, Artiste, Concert, Lieu et Archive. Pour id_type dans Tag, cela peut référencé les clés primaires des relations suivantes : Groupe, Concert, Lieu et Playlist. Cependant, il n'y a donc pas de contraintes de clé étrangère.

Pour la plupart des attributs, nous mettons une contrainte de non nullité. Cependant, les attributs suivant n'en ont pas :

- pays,ville (Utilisateurs)
- date_naissance (Personnes)
- volontaires, cause_soutien,enfants_admissible (Concert)
- photo,video (Archive)
- commentaire (Avis)

On a également des contraintes de vérification, pour les attributs :

- pseudo (Utilisateurs) : on vérifie une taille inférieur ou égale à 20.
- pwd (Utilisateurs) : on vérifie une taille d'au moins 8.
- note (Avis) : on vérifie que la note est comprise entre 0 et 10.
- participe et est_interesse (Participation) : On vérifie que les deux ont des valeurs différentes.

Nous avons une contrainte d'unicité pour « valeur » dans Tag, pour nous assurer de ne pas avoir un doublon de tag.

Il n'y a pas de contrainte permettant de modélisé la symétrie pour la table Amis. Nous estimons que cela se fera au moment de l'insertion, où nous devrions insérer dans les deux sens. Egalement, si nous n'ajoutons pas dans les deux sens, alors il faudra le comprendre dans la récupération des données.

De plus, nous voulions également rajouter deux contraintes de vérifications pour playlist (il ne peut y avoir que 10 playlists par utilisateurs, et il ne peut y avoir que 20 morceaux). Cependant, nous n'avons pas réussi à implémenté ça, car en PSQL, il n'était pas possible d'utiliser un « IN » dans un Check. En nous renseignant sur internet, nous avons appris qu'il fallait utiliser des « triggers » pour implémenté cela, chose que nous n'avons pas fait, car hors programme. Notre modélisation ne nous permet donc pas de faire ces deux contraintes.

Nous aurions pu également rajouter des contraintes d'unicité à certains endroit, notamment pour les pseudos, pour s'assurer que chaque compte à un pseudo différent.

De même, nous voulions ajouter du hachage pour le mot de passe présent dans Utilisateurs, mais ce n'était pas le but du projet, cela aurait pu cependant rajouter un côté plus naturel à la base de donnée.

IV – Requêtes :

Nous avons voulu faire un ordre de complexité des requêtes, avec d'abord des requêtes simples, qui seront plus difficile ensuite.

Pour des raisons de limite de place, vous retrouverez les requêtes correctement écrite dans le fichier « Requete_sql.sql » dans l'archive du projet.

Requête 1 : La requête est simple, elle récupère les pseudos des utilisateurs qui participeront à des concerts dans des pays autre que la France, ainsi que les noms des concerts. A partir de la table Utilisateurs, nous allons Joindre les tables Participation, Concert, Lieu_Concert et Concert avec les attributs en commun entre chacune des tables, puis récupérer les pseudos des utilisateurs qui sont intéressés quand le lieu n'est pas en France.

Requête 2 : Requête simple qui renvoie les pseudonymes de deux utilisateurs différents et leur ville respective s'ils résident dans la même ville, en utilisant une auto-jointure sur la table Utilisateurs.

Requête 3 : La requête récupère les pseudos des utilisateurs qui participeront à tout les concerts disponibles, en utilisant deux sous requêtes corrélées.

Requête 4 : La requête renvoies les villes et la moyenne des prix des concerts dans les dites villes, dans l'ordre décroissant. On effectue une sous requête dans le FROM, qui recupère la moyenne des prix par ville.

Requête 5 : La requête renvoie les noms des artistes qui ont des concerts prévus à partir de la date actuelle, on effectue une sous requête dans le WHERE pour simplement récupérer uniquement les identifiants d'artistes qui correspondent à nos critères.

Requête 6 : Requête qui va récupérer les artistes avec une moyenne des avis d'au moins 8. On utilisera une fonction d'agrégation pour cela.

Requête 7 : La requête renvoie les noms des artistes et le nombre de morceau qu'ils ont fait, disponible sur la plateforme.

Requête 8 : Requête qui récupère les HITS de la plateforme, c'est à dire les morceaux les plus appréciés d'après les avis. On effectue plusieurs Jointures pour récupérer tout ce qu'il nous faut, (Avis, Morceau et l'Artiste) puis on fait la moyenne des notes, et on ordonne.

Requête 9 : La requête renvoie le nombre total de concerts et le prix moyen des concerts où au moins une personne à participé.

Requête 10 : La requête renvoie les pseudonymes des utilisateurs et le nombre d'avis qu'ils ont publiés (s'ils en ont publiés). On utilise des LEFT JOIN pour lié les tables Utilisateurs, Auteur_avis et Avis entre elles.

Requête 11 : La requête renvoie les personnes qui ont assisté à au moins un concert en utilisant l'agrégation. **La requête 12** fait la même chose, mais cette fois en utilisant une corrélation.

Requête 13 : On récupère la liste des 5 paires d'artistes qui se produisent généralement le plus ensemble, avec le nombre de performances qu'ils ont fait ensemble.

Requête 14 : La requête fait le classement des utilisateurs en fonction de leur participation à des concerts. On utilise la fonction RANK() après avoir récupérer le nombre de participation par utilisateurs qui ont participé au moins une fois.

Requête 15 : On récupère les utilisateurs qui ont participé à tout les concerts d'une ville donnée. Grace à prompt, on peut choisir la ville (la seul ville dans la BDD qui semble donné une réponse non null semble être Berne)

Requête 16 : La requête recupère la liste de tout les organismes, ainsi que le nombres de concert qu'ils ont organisés par ordre décroissant. On va seulement faire des jointures et compter le nombre d'id de concert.

Requête 16bis : La liste des artistes qui ont joué plus de 2 fois dans des concerts dont le prix était supérieur à 120 euros. On fera une sous-requête dans le WHERE qui va récupérer les concerts puis on les comptera.

Requête 17 : On récupère la liste des villes qui ont une moyenne de prix de concert supérieur à la moyenne générale. On utilisera une table temporaire avec WITH, pour récupérer la moyenne des prix de concert par ville, puis une autre pour récupérer le prix moyen au global, on compare ensuite les deux et on garde les villes souhaité.

Requête 18 : La liste des utilisateurs ayant des morceaux en commun dans leur playlist respective, ainsi que le nombre de morceaux. On utilisera cette fois-ci de l'auto jointure ainsi que de l'aggrégation pour compter.

Requête 19 : Requête récursive qui va récupérer la liste de tous les utilisateurs qui sont suivis par un utilisateurs donné. On donne en argument un identifiant d'utilisateur avec prompt, puis en premier lieu on récupère les données dans la table Suivis où notre utilisateur de départ est le suiveur. La deuxième partie de la requête fait une jointure entre la première partie et la table Suivis pour que l'on puisse récupérer les utilisateurs suivis par le précédent suiveur. Quand il n'y a plus d'enregistrement à lire, on récupère uniquement les utilisateurs suivis sans doublon.

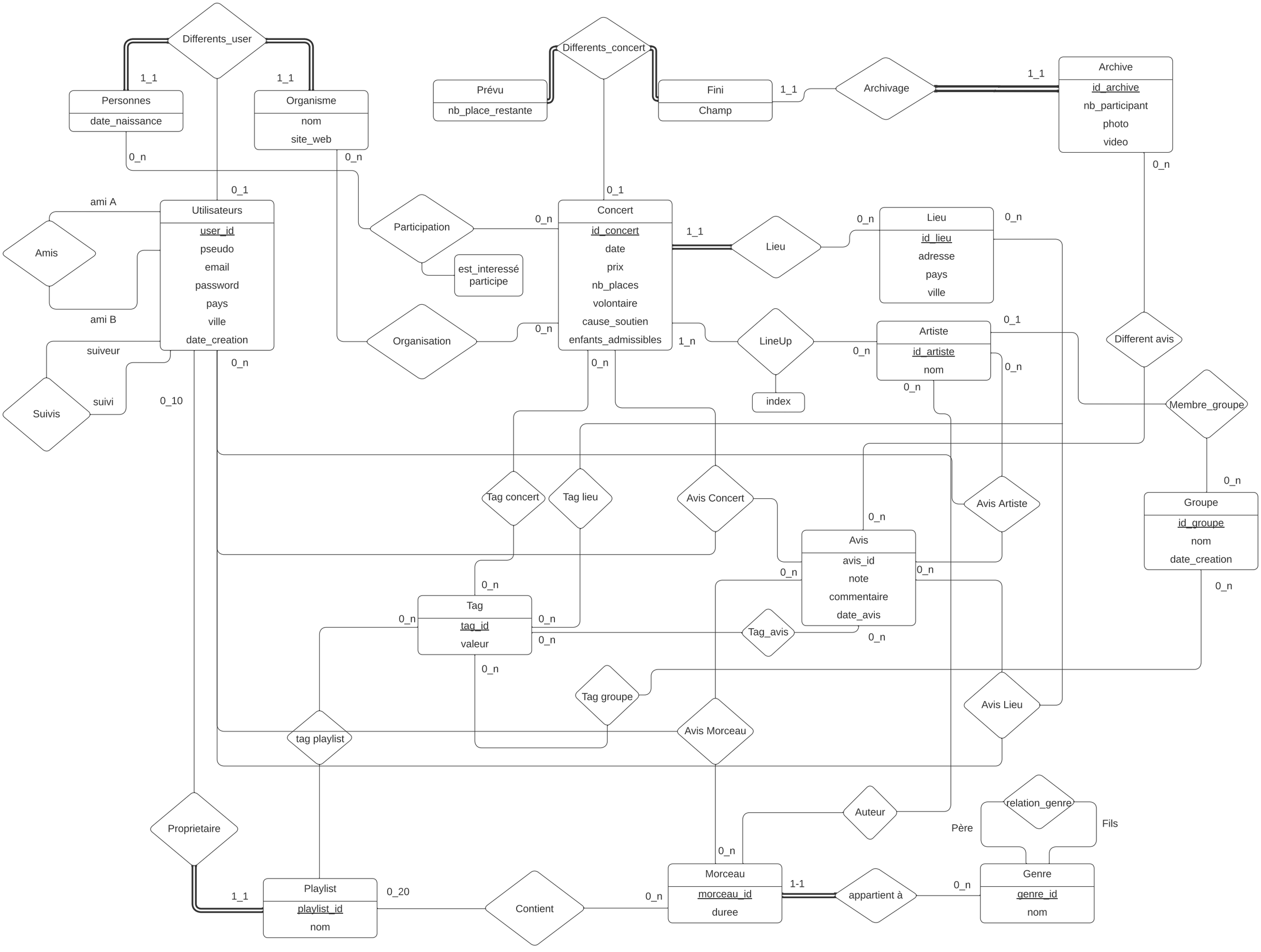
Requête 20 : Requête récursive qui va récupérer tout les parents du genre donné. On donne d'abord un identifiant de genre avec prompt, puis dans la première partie on sélectionne les enregistrements de la table Relation_Genre où le genre de départ est le fils. Ensuite dans la seconde partie, On effectue une jointure entre le résultat obtenus juste avant et la table pour que l'on récupère les genres parents des genres précédemment récupérés. Une fois qu'il n'y a plus de récursion, on sélectionne les identifiants des genres parents.

V – Algorithme de recommandation

On a implémenté un algorithme de recommandation qui prend en compte les critères suivants :
- La localisation de l'utilisateur : on regarde si le pays de l'utilisateur est le même que celui du concert.
- L'historique de participation de l'utilisateur : on regarde si l'utilisateur a déjà participé à un concert de l'artiste.
- L'historique de participation des amis de l'utilisateur : on regarde si les amis de l'utilisateur ont déjà participé à un concert de l'artiste.

On a donc créé trois fonctions qui calculent les scores de ces trois critères. On a ensuite créé une fonction qui calcule le score total d'un concert
en additionnant les trois scores précédents. On a enfin créé une fonction qui calcule le score total de tous les concerts et qui renvoie le concert avec le score le plus élevé.

Nous estimons que ce petit algorithme est envisageable pour permettre l’implémentation d’un plus gros algorithme permettant de recommander au mieux les utilisateurs de la plateforme.


Differents_user
1_1
1_1
Personnes
date_naissance
Organisme
nom
site_web
0_n
0_n
0_1
ami A
Utilisateurs
user_id
pseudo
email
password
pays
ville
date_creation
Amis
ami B
suiveur
Suivis
suivi
0_10
0_n
Participation
est_interessé
participe
Organisation
Differents_concert
Prévu
nb_place_restante
Fini
Champ
1_1
Archivage
1_1
Archive
id_archive
nb_participant
photo
video
0_n
0_1
Concert
id_concert
date
prix
nb_places
volontaire
cause_soutien
enfants_admissibles
0_n
0_n
1_1
Lieu
0_n
0_n
Lieu
id_lieu
adresse
pays
ville
0_n
1_n
LineUp
index
0_n
0_1
Artiste
id_artiste
nom
0_n
0_n
Different avis
Membre_groupe
0_n
Groupe
id_groupe
nom
date_creation
0_n
Tag concert
Tag lieu
Avis Concert
Avis Artiste
0_n
Avis
avis_id
note
commentaire
date_avis
0_n
0_n
0_n
Tag
tag_id
valeur
0_n
0_n
0_n
0_n
Tag_avis
0_n
Tag groupe
Avis Lieu
tag playlist
Avis Morceau
Proprietaire
Auteur
relation_genre
Père
Fils
0_n
1_1
Playlist
playlist_id
nom
0_20
Contient
0_n
Morceau
morceau_id
duree
1-1
appartient à
0_n
Genre
genre_id
nom