ABBASI Tahreem
VELLETRIE Aalliayah

RAPPORT

Le projet de conception de la base de données "BDD_musique" a été initié dans le cadre du développement d'un nouveau réseau social open source dédiée à la musique. L'objectif principal de cette base de données est de fournir une structure robuste et complète pour gérer les différentes entités et fonctionnalités liées à l'industrie musicale. Ce réseau social vise à offrir une plateforme participative sans publicité, permettant à ses utilisateurs de découvrir de nouveaux morceaux, de suivre des artistes, de participer à des concerts, et d'interagir avec d'autres passionnés de musique. Ce rapport détaille les choix de conception faits pour chaque table, en mettant l'accent sur la normalisation des données, les contraintes et la facilité d'utilisation.

1. Modele conceptuel de données (schema entites-associations)

Le modèle conceptuel a été élaboré en identifiant les principales entités du système, leurs attributs et les relations entre elles. Les entités majeures incluent **Utilisateur, Artistes, Association, Lieu, Concert, et Playlist**. Les associations, telles que Followers, Interet, Annonce_concert ont été établies pour refléter les interactions entre ces entités. Les types de données ont été soigneusement sélectionnés pour assurer une représentation précise des informations stockées. Par exemple, l'utilisation de *VARCHAR* pour les champs textuels limités à un nombre de caractères comme les noms, permet une flexibilité suffisante sans gaspiller d'espace. *INTEGER* a été utilisé pour les identifiants et *BOOLEAN* pour les valeurs binaires. Des contraintes d'intégrité, comme les clés primaire et étrangère, ont été mise en place pour garantir la cohérence des données et établir des relations significatives entre les entités. Les clés primaires ont été définies de manière à garantir l'unicité des enregistrements dans chaque table. Les clés étrangères établissent des liens entre les tables, facilitant ainsi la navigation et la récupération d'informations interconnectées sans redondance des informations. Afin de garantir la qualité et la cohérence des données, nous avons insérer des contraintes d'intégrité. Par exemple, la table **Concert_archive** utilise une contrainte *CHECK* et une fonction pour garantir que la date d'archivage est postérieure à la date du concert correspondant. D'autres contraintes d'intégrité spécifiques ont été introduites pour renforcer la qualité des données telle que la contrainte dans la table **Avis**, qui limite la note à une plage spécifique, assurant la validité des évaluations. L'ajout de déclencheurs (triggers) pour imposer des contraintes spécifiques, comme dans la table **Concert_Archive** avec *check_archive_date_trigger*, renforce la sécurité des données et assure la cohérence des données stockées dans notre base de données. Cela assure que les données entrantes respectent des conditions spécifiques avant d'être acceptées.

2. Normalisation des tables

La normalisation des tables a été effectuée pour minimiser les redondances et prévenir les anomalies de mise à jour, suppression et insertion. La principale table de la base de données est la table **Utilisateur**. Elle correspond au « compte » que l'on crée sur un réseau social classique. Les premières tables sont celles directement liées aux fonctions internes du « compte » comme les tables **Publications**, **Message_Privé** et **Suggestions**. Par exemple, la table **Personne** est une extension de la table **Utilisateur**, évitant ainsi la duplication des informations de cette table. La séparation des entités **Utilisateur** et **Personne** permet d'étendre la plateforme pour inclure d'autres types d'utilisateurs tels que les groupes, associations, etc., tout en préservant la capacité à stocker des informations spécifiques à une personne. Les

relations symétriques, comme celles dans **Amis** (conçue pour enregistrer les relations d'amitiés entre les utilisateurs) et **Follower** (conçue pour enregistrer les relations de suivi entre utilisateurs), ont été gérées pour permettre des interactions suivies ou d'amitié entre utilisateurs. Ces deux tables sont essentielles pour la mise en œuvre des fonctionnalités sociales. La flexibilité de cette approche permet à un utilisateur d'en suivre un autre sans obligation réciproque.

3. Gestion des Diverses Fonctionnalités

La base de données prend en charge diverses fonctionnalités telles que la gestion d'avis, d'annonces de concerts, d'archives de concerts, de playlists, etc. Les tables **PlaylistMorceau** et **Tag** ont été créées pour gérer les associations entre playlists et morceaux, ainsi que pour permettre le tagging de différentes entités. L'utilisation d'identifiants uniques a été favorisée pour garantir l'unicité des enregistrements. Par exemple, la combinaison de l'ID et du nom dans la table **Lieu** assure que chaque lieu est unique. Cela simplifie les opérations de recherche et garantit des résultats précis. L'utilisation de *TIMESTAMP* et *DATE* pour enregistrer des informations temporelles, telles que la date d'inscription des utilisateurs, la date des concerts, et la date d'archivage des concerts, permet une gestion précise des événements dans le temps. Le modèle intègre la flexibilité des tags pour permettre une catégorisation polyvalente. Les tags peuvent être utilisés pour identifier des entités telles que groupes, avis, concerts, playlists et lieux. Cette fonctionnalité favorise la recherche, la découverte et la recommandation d'éléments similaires. Nous avons décidé de ne pas lier la table **Avis** avec la table **Utilisateur** car à chaque avis émis par un utilisateur donné, on répète toutes ses informations. Ceci générait beaucoup de redondance.

4. Générer des données

Dans un premier temps, nous voulions récupérer un jeu de données déjà existant, mais lorsque que nous nous sommes aperçus du volume de notre base de données, nous avons décidé qu'il était plus judicieux de créer les nôtres. Pour cela nous avons utilisé la bibliothèque Faker de python. L'avantage est que nous travaillons sur nos propres données, fictives et ces dernières nous permettaient de simuler des scénarios d'utilisation de notre application. De plus cela nous empêchait de traiter des données sensibles ou personnelles réelles. Les données sont donc cohérentes et diversifiées et elles permettent d'illustrer nos requêtes.

5. Limites et améliorations possibles

Bien que le modèle réponde aux exigences initiales, il présente quelques limitations. De plus, des améliorations peuvent être apportées pour gérer plus efficacement les tags en ajoutant des catégories introduisant des genres musicaux, des émotions, instruments etc… On pourrait également implémenter un système de recommandation plus avancé basé sur l'historique d'écoute, les préférences des utilisateurs et les interactions sociales. Il faudrait également envisagez la mise en place d'un mécanisme de réplication des données, assurant la sauvegarde et la redondance en cas de défaillance du système. De plus, un système de notifications pour informer les utilisateurs des nouveaux concerts, des mises à jour de playlists, etc… pourrait être envisagé.

ABBASI Tahreem
VELLETRIE Aalliayah

La conception de la base de données BDD_musique a été pensée de manière à garantir la précision, la cohérence et la facilité d'utilisation. Les choix de types de données, de normalisation, de contraintes d'intégrité et d'utilisation de fonctions ont été guidés par ces principes pour créer un système fiable et performant répondant aux besoins variés de l'industrie musicale. Ce modèle offre une base robuste et flexible pour le développement du réseau social musical, avec la capacité de s'adapter à des évolutions futures.

SCHEMA RELATIONNEL

Des contraintes d'unicité ont été ajoutées pour certaines colonnes, notamment dans les tables **Association, Artistes, Lieu** et **Utilisateur**. Les relations entre les tables sont indiquées par les clés étrangères (#). La conversion du modèle E/R en modèle relationnel génère des nouvelles. Ces tables **supplémentaires** sont introduites pour gérer les relations entre les entités (par exemple, les amis d'un utilisateur, les tags associés à un lieu, etc…)

1. Historique (**Historique_ID,** Type, Date)
2. Publication (**ID**, Date)
3. Suggestion (**Suggestions_ID**, Type, Date)
4. Message-privé (**Message_ID**, Destinataire_pseudo, Contenu, Date, Etat)
5. Association (**Association_ID**, Nom, Site-web
6. Artistes (**Artistes_ID**, Nom, Genre, Nb_Membres)
7. Personne (**Personne_ID**, Age, Genre, Nationalite)
8. Playlist (**Playlist_ID**, Nom, IsGroupe, Groupe_ID, Nb_morceau, Artistes_ID)
9. Morceau (**Morceau_ID**, Titre, Nom_album, Genre)
10. PlaylistMorceau (#Playlist_Id, #Morceau_Id)
11. Avis (**Avis_ID**, Note, Commentaire, Pseudo, Date, Tags)
12. Lieu (**Lieu_ID**, Nom, adresse, #Avis_ID)
13. Concert (**Concert_ID**, Nom, Date, Organisateurs, Line-up, Nombre_places, Cause_soutien, Espace_exterieur, Enfants_permis, Prix, Nb_volontaires, #ID_Lieu)
14. Archive (**Archive_Id**, #Concert_Id, Date_archivage, Nb_participants, Avis_participants, Photo_path, Video_path, #Avis_ID)
15. Participe (#Concert_ID, #Personne_ID)
16. Interet ((#Concert_ID, #Personne_ID)
17. Utilisateur (**Utilisateur_ID**, Pseudo, Email, Password, Biographie, Date_inscription, #Historique_Id, #Publication_Id, #Association_Id, #Personne_Id, #Lieu_Id, #Artistes_Id, #Playlist_Id)
18. Annonce_Concert (**Annonce_ID**, #Utilisateur_Id, #Concert_Id, Date_annonce)
19. Relation (**Relation_ID**, User1)
20. Follower (#Utilisateur_Id, #Relation_ID)
21. Tag (**Tag_Id**, Mot)

Amis (#Utilisateur_Id, #Relation_ID
Tag_Lieu (#Tag_Id, #Lieu_ID)
Tag_Concert (#Tag_ID, #Concert_ID)
Tag_Artistes (#Tag_Id, #Artistes_ID
Tag_Playlist (#Tag_Id, #Playlist_ID)
Tag_Avis (#Tag_Id, #Avis_ID)
Uti_MP (#Utilisateur_ID, #Message_ID)

ABBASI Tahreem
VELLETRIE Aalliayah

Uti_Sugg (#Utilisateur_ID, #Suggestion_ID)
Avis_Artistes (#Avis_Id, #Artistes_ID)
Avis_Morceau (#Avis_Id, #Morceau_ID)

REQUETES

Les requêtes ont été le point le plus laborieux à traiter lors de ce projet. On devait interroger et manipuler des données stockées une base de données volumineuse. Il fallait toujours s'assurer d'avoir des index appropriés sur les colonnes impliquées dans les jointures et il fallait couvrir un large éventail de scénario. De plus, il y avait des exigences à respecter comme la récursivité, le nombre d'agrégats à utiliser ou encore les sous-requêtes. Prenons par exemple la requête numéros 7. Pour afficher les morceaux et les playlists auxquelles ils appartiennent, il fallait tenir compte des contraintes d'intégrité référentielles en utilisant des LEFT JOIN pour inclure les morceaux qui ne sont pas encore liés à une playlist. Les requêtes 2 et 3 participent à la normalisation des données évoquée précédemment en utilisant des jointures appropriées pour récupérer des informations à partir de plusieurs tables, favorisant ainsi une conception normalisée et évitant la redondance des données. Enfin, la requête 14 est un exemple d'optimisation car l'utilisation du LEFT JOIN et de la clause HAVING minimise le nombre de lignes traitées. Nous avons essayé d'utiliser une approche méthodique associée à notre apprentissage approfondi des principes de conception de base de données afin de créer des requêtes optimisées et robustes.

1. **Lister tous les concerts avec leurs détails**

```
SELECT Concert.*, Lieu.Nom AS Lieu_Nom, Lieu.Adresse AS Lieu_Adresse
FROM Concert
JOIN Lieu ON Concert.ID_Lieu = Lieu.Lieu_ID;
```

2. **Afficher les utilisateurs et leurs playlists associées**

```
SELECT Utilisateur.Pseudo, Playlist.Nom AS Playlist_Nom
FROM Utilisateur
LEFT JOIN Playlist ON Utilisateur.Playlist_Id = Playlist.Playlist_ID;
```

3. **Afficher les utilisateurs et leurs annonces de concerts associées**

```
ELECT Concert.Nom AS Concert_Nom, Lieu.Nom AS Lieu_Nom
FROM Concert
INNER JOIN Lieu ON Concert.ID_Lieu = Lieu.Lieu_ID;
```

4. **Récupérer les morceaux d'une playlist spécifique**

```
SELECT Morceau.*
FROM Morceau
JOIN PlaylistMorceau ON Morceau.Morceau_Id = PlaylistMorceau.Morceau_Id
WHERE PlaylistMorceau.Playlist_Id = 4 ;
```

5. **Obtenir les lieux associés à un tag spécifique**

ABBASI Tahreem
VELLETRIE Aalliayah

```
SELECT Lieu.*
FROM Lieu
JOIN Tag_Lieu ON Lieu.Lieu_ID = Tag_Lieu.Lieu_ID
WHERE Tag_Lieu.Tag_Id = 8;
```

6. **Renvoie les 5 concerts ayant le plus de participants**

```
SELECT Concert.Concert_Id, Concert.Nom AS Nom_Concert,
COUNT(Participe.Personne_ID) AS Nombre_Participants
FROM Concert
JOIN Participe ON Concert.Concert_Id = Participe.Concert_ID
GROUP BY Concert.Concert_Id, Concert.Nom
ORDER BY Nombre_Participants DESC
LIMIT 5;
```

7. **Afficher les morceaux et les playlists auxquelles ils appartiennent**

```
SELECT Morceau.Titre, Playlist.Nom AS Playlist_Nom
FROM Morceau
LEFT JOIN PlaylistMorceau ON Morceau.Morceau_Id = PlaylistMorceau.Morceau_Id
LEFT JOIN Playlist ON PlaylistMorceau.Playlist_Id = Playlist.Playlist_Id;
```

8. **Utilisateur qui a créé des playlists de groupe**

```
SELECT Utilisateur.Utilisateur_ID, Utilisateur.Pseudo, Playlist.Playlist_ID, Playlist.Nom AS
Nom_Playlist
FROM Utilisateur
JOIN Playlist ON Utilisateur.Playlist_Id = Playlist.Playlist_ID
WHERE Playlist.IsGroupe;
```

9. **Requêtes sur au moins 3 tables : Renvoie une liste d'utilisateurs avec des infos sur leur historique/publications**

```
SELECT Utilisateur.Pseudo, Historique.Type, Publication.Date AS Publication_Date
FROM Utilisateur
JOIN Historique ON Utilisateur.Historique_Id = Historique.Historique_ID
LEFT JOIN Publication ON Utilisateur.Publication_Id = Publication.Publication_ID;
```

10. **Auto-jointure (jointure réflexive) : renvoie les playlists avec des noms similaires mais des identifiants différents.**

```
SELECT
  p1.Playlist_ID AS Playlist1_ID,
  p1.Nom AS Playlist1_Nom,
  p2.Playlist_ID AS Playlist2_ID,
  p2.Nom AS Playlist2_Nom
FROM
  Playlist p1
JOIN
  Playlist p2 ON p1.Nom = p2.Nom AND p1.Playlist_ID < p2.Playlist_ID;
```

ABBASI Tahreem
VELLETRIE Aalliayah

11. **Sous-requête corrélée : Sélectionne les pseudos des utilisateurs qui ont au moins un follower**

```
SELECT Utilisateur.Pseudo
FROM Utilisateur
WHERE EXISTS (
   SELECT 1
   FROM Follower
   WHERE Follower.Utilisateur_Id = Utilisateur.Utilisateur_ID
```

12. **Sous-requête dans le FROM : compte le nombre de followers pour chaque utilisateur**

```
SELECT Utilisateur.Pseudo,
   (SELECT COUNT(*) FROM Follower WHERE Follower.Utilisateur_Id =
Utilisateur.Utilisateur_ID) AS NombreFollowers
FROM Utilisateur;
```

13. **Sous-requête dans le Where : renvoie les pseudos des utilisateurs qui ont au moins 1 follower**

```
SELECT Pseudo
FROM Utilisateur
WHERE Utilisateur_ID IN (SELECT Utilisateur_Id FROM Follower);
```

14. **Agrégats nécessitant GROUP BY et HAVING : renvoie le nombre de lieu ayant plus de 5 concerts**

```
SELECT
  L.Lieu_ID,
  L.Nom AS Nom_Lieu,
  COUNT(C.Concert_Id) AS Nombre_Concerts
FROM
  Lieu L
LEFT JOIN
  Concert C ON L.Lieu_ID = C.ID_Lieu
GROUP BY
  L.Lieu_ID, L.Nom
HAVING
  COUNT(C.Concert_Id) > 5;
```

15. **Requête impliquant le calcul de deux agrégats : renvoie les utilisateurs qui ont envoyé au moins un message privé**

```
SELECT
  Utilisateur.Utilisateur_ID,
  Utilisateur.Pseudo,
  COUNT(Message_Prive.Message_ID) AS Nombre_Messages,
  ROUND(AVG(LENGTH(Message_Prive.Contenu)                                    -
LENGTH(REPLACE(Message_Prive.Contenu, ' ', '')) + 1), 1) AS Moyenne_Mots
FROM
  Utilisateur
```

```
LEFT JOIN
  Uti_MP ON Utilisateur.Utilisateur_ID = Uti_MP.Utilisateur_Id
LEFT JOIN
  Message_Prive ON Uti_MP.Message_ID = Message_Prive.Message_ID
GROUP BY
  Utilisateur.Utilisateur_ID, Utilisateur.Pseudo
HAVING
  COUNT(Message_Prive.Message_ID) > 0;
```

16. **Jointure externe (left join) : renvoie le nom de chaque utilisateur ainsi que le nombre de concert auxquels il a participé**

```
SELECT Utilisateur.Pseudo, COUNT(Concert.Concert_Id) AS NombreParticipations
FROM Utilisateur
LEFT JOIN Participe ON Utilisateur.Utilisateur_ID = Participe.Personne_ID
LEFT JOIN Concert ON Participe.Concert_ID = Concert.Concert_Id
GROUP BY Utilisateur.Pseudo;
```

17. **Deux requêtes équivalentes exprimant une condition de totalité : identifie les utilisateurs qui ont à la fois des followers et des amis (avec et sans agrégation)**

```
-- #sans aggregation
SELECT Pseudo
FROM Utilisateur
WHERE EXISTS (
    SELECT 1
    FROM Follower F
    WHERE F.Utilisateur_Id = Utilisateur.Utilisateur_ID
) AND EXISTS (
    SELECT 1
    FROM Amis A
    WHERE A.Utilisateur_Id = Utilisateur.Utilisateur_ID
);

-- #avec agregation
SELECT Utilisateur.Utilisateur_ID, Utilisateur.Pseudo
FROM Utilisateur
LEFT JOIN (
  SELECT Utilisateur_Id, COUNT(*) AS FollowerCount
  FROM Follower
  GROUP BY Utilisateur_Id
) AS FollowerCounts ON Utilisateur.Utilisateur_ID = FollowerCounts.Utilisateur_Id
LEFT JOIN (
  SELECT Utilisateur_Id, COUNT(*) AS AmisCount
  FROM Amis
  GROUP BY Utilisateur_Id
) AS AmisCounts ON Utilisateur.Utilisateur_ID = AmisCounts.Utilisateur_Id
WHERE COALESCE(FollowerCounts.FollowerCount, 0) = 0 OR
COALESCE(AmisCounts.AmisCount, 0) = 0;
```

ABBASI Tahreem
VELLETRIE Aalliayah

18. **Requête avec différences si présence de null : affiche les informations sur les utilisateurs et leur historique**

```
-- Requête initiale
SELECT Utilisateur.Pseudo, Historique.Type
FROM Utilisateur
LEFT JOIN Historique ON Utilisateur.Historique_Id = Historique.Historique_ID;

-- #requete sans différence en présence de null
-- Modification pour rendre les résultats équivalents en cas de nulls
SELECT Utilisateur.Pseudo, COALESCE(Historique.Type, '0') AS Type
FROM Utilisateur
LEFT JOIN Historique ON Utilisateur.Historique_Id = Historique.Historique_ID;
```

19. **Requête récursive : calculer le prochain jour off d'un groupe**

```
WITH ClassementConcerts AS (
  SELECT
    C.Concert_Id,
    C.Nom AS Nom_Concert,
    A.Nom AS Nom_Artiste,
    COUNT(P.Personne_ID) AS Nombre_Participants,
    RANK() OVER (PARTITION BY EXTRACT(MONTH FROM C.Date) ORDER BY
COUNT(P.Personne_ID) DESC) AS Classement_Mensuel
  FROM Concert C
  JOIN Participe P ON C.Concert_Id = P.Concert_ID
  JOIN Artistes A ON C.Nom = A.Nom

  WHERE EXTRACT(YEAR FROM C.Date) = 2023
  GROUP BY C.Concert_Id, C.Nom, A.Nom, C.Date

)
SELECT
  Concert_Id,
  Nom_Concert,
  Nom_Artiste,
  Nombre_Participants
FROM ClassementConcerts
WHERE Classement_Mensuel <= 10;
```

20. **Requête utilisant du fenêtrage : détails des concerts ayant eu lieu en 2023, les classe mensuellement par le nombre de participants et retourne ceux se classant parmi les 10 premiers dans leur mois respectif.**

```
WITH ClassementConcerts AS (
  SELECT
    C.Concert_Id,
    C.Nom AS Nom_Concert,
    A.Nom AS Nom_Artiste,
    COUNT(P.Personne_ID) AS Nombre_Participants,
```

ABBASI Tahreem
VELLETRIE Aalliayah

```
   RANK() OVER (PARTITION BY EXTRACT(MONTH FROM C.Date) ORDER BY
COUNT(P.Personne_ID) DESC) AS Classement_Mensuel
  FROM Concert C
  JOIN Participe P ON C.Concert_Id = P.Concert_ID
  JOIN Artistes A ON C.ID_Lieu = A.Artistes_ID
  WHERE EXTRACT(YEAR FROM C.Date) = 2023 -- peut etre changé
  GROUP BY C.Concert_Id, C.Nom, A.Nom
)
SELECT
  Concert_Id,
  Nom_Concert,
  Nom_Artiste,
  Nombre_Participants
FROM ClassementConcerts
WHERE Classement_Mensuel <= 10;
```

CONTRAINTES

CONTRAINTES :

- Une archive existe uniquement pour les concerts ayant deja eu lieu : Date_Archive < Date_Concert
- Cardinalité minimale de 0 pour l'historique d'un utilisateur (si supprime son historique)

CONTRAINTES CHECK :

- Message_Prive(Etat) peut soit etre 'distribue' soit 'lu'
- Historique(Type) a des possibilités prédéfini dans une liste avec la contriante check lors de la création de la table
- Suggestion(Type) déja prédéfinies car c'est un réseau social axé sur la musique
- Playlist(Nb_morceau) est compris entre 0 et 20 (une plyalist a maximum 20 morceaux)
- Avis (Note) est une note comprise entre 0 et 10

CONTRAINTES D'UNICITE :

- Lieu(Nom, Adresse)
- Association(Nom, Site_web)
- Utilisateur(Pseudo, email, password)
- Artistes(Nom, genre, membres)

CONTRAINTES EXTERNES:

- (Personne, Concert) ∈ Intéret U Participe

➔ Une personne peut soit etre intéréssé soit participe a un concert donné

Voici les clés étrangères associées aux différentes tables :

- Suggestions(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id)
- Playlist(Artistes_ID) ∈ Artistes(Artistes_ID)
- PlaylistMorceau(Playlist_Id) ∈ Playlist(Playlist_Id) U (Morceau_Id fait) ∈ Morceau(Morceau_Id)
- Lieu(Avis_ID) ∈ Avis(Avis_ID)
- Concert(ID_Lieu) ∈ Lieu(Lieu_ID)

ABBASI Tahreem
VELLETRIE Aalliayah

- Archive(Concert_Id)∈ Concert(Concert_Id) U Archive(Avis_ID) ∈ Avis(Avis_Id)
- Participe(Concert_ID) ∈ Concert(Concert_ID) U Participe(Personne_ID) ∈ Personne(Personne_ID)
- Interet(Concert_ID) ∈ Concert(Concert_ID) U Interet(Personne_ID) ∈ Personne(Personne_ID)

- Utilisateur(Historique_Id) ∈ Historique(Historique_Id) et
  - Uti(Publication_Id) ∈ Publication(Publication_Id),
  - Uti(Association_Id) ∈ Association(Association_Id),
  - Uti(Personne_Id) ∈ Personne(Personne_Id),
  - Uti(Lieu_Id) ∈ Lieu(Lieu_Id),
  - Uti(Artistes_Id) ∈ (Artistes_Id),
  - Uti(Playlist_Id) ∈ Playlist(Playlist_Id)

- Annonce_Concert(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id) U Ann_Con(Concert_Id)∈ Concert(Concert_Id)

- Follower(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id) U Follower(Relation_ID) ∈ Relation(Relation_ID)

- Amis(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id) U Amis(Relation_ID ∈ Relation(Relation_ID)

- Uti_MP(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id) U Uti_MP(Message_ID) ∈ Message_Prive(Message_ID)

- Uti_Sugg(Utilisateur_Id) ∈ Utilisateur(Utilisateur_Id) U Uti_Sugg(Suggestion_ID) ∈ Suggestion(Suggestion_ID)

- Avis_Artiste(Avis_Id) ∈ Avis(Avis_Id) U Avis_Artistes(Artistes_ID) ∈ Artistes(Artistes_ID)

- Avis_Morceau(Avis_Id) ∈ Avis(Avis_Id) U Avis_Morceau(Morceau_ID) ∈ Morceau(Morceau_ID)

- Tag_Lieu(Tag_Id) ∈ Tag(Tag_Id) U Tag_Lieu(Lieu_ID) ∈ Lieu(Lieu_ID)

- Tag_Concert(Tag_Id) ∈ Tag(Tag_Id) U Tag_Concert(Concert_ID) ∈ Concert(Concert_ID)

- Tag_Artistes(Tag_Id) ∈ Tag(Tag_Id) U (Artistes_ID) ∈ Artistes(Artistes_ID)

- Tag_Playlist(Tag_Id) ∈ Tag(Tag_Id) U (Playlist_ID) ∈ Playlist(Playlist_ID)

- Tag_Avis(Tag_Id) ∈ Tag(Tag_Id) U Tag_Avis(Avis_ID) ∈ Avis(Avis_ID)

Diagrammme E/R de la Base de Données Musique