Rapport projet BDR :
Alpha HYF

Kobbi Fares

Ben Mzoughia Youssef SI3

Younes Hammoud Groupe 3

Sommaire :

1. Description du projet:

L'objectif de ce projet est de concevoir et de mettre en place une base de données relationnelle qui modélise les interactions sociales entre utilisateurs, leurs publications, et les groupes thématiques auxquels ils peuvent appartenir. La base de données doit permettre de gérer les relations entre utilisateurs (amis, abonnés), les publications (contenu, visibilité, interactions), et les interactions sociales (likes, partages, commentaires). Elle doit également inclure la gestion des groupes thématiques et de leurs membres.

La base de données fournira une gamme de services essentiels pour gérer les utilisateurs, les publications, les interactions et les groupes thématiques. Tout d'abord, elle assurera la gestion des utilisateurs en stockant les informations de profil telles que le nom, le prénom, la date de naissance, et d'autres détails pertinents. Elle gérera également les relations entre les utilisateurs, notamment les amis et les abonnés, permettant ainsi de modéliser les connexions sociales de manière claire et structurée.

En ce qui concerne la gestion des publications, la base de données stockera les publications des utilisateurs, y compris le contenu, la date de publication, l'auteur et la visibilité (privée ou publique). Un utilisateur pourra publier du contenu visible uniquement par ses amis (privé) ou par tout le monde (public). De plus, une publication pourra être associée à un groupe thématique, auquel cas elle ne sera visible que par les membres de ce groupe, offrant ainsi une gestion fine de la visibilité des contenus.

La base de données prendra également en charge la gestion des interactions en enregistrant les réactions des utilisateurs aux publications, telles que les likes, les partages et les commentaires. Les utilisateurs pourront interagir avec les publications en les likant, en les commentant ou en les partageant. La base de données suivra également le nombre de likes, de partages et de commentaires pour chaque publication, fournissant ainsi une vue détaillée des interactions sociales.

Enfin, la base de données gérera les groupes thématiques en permettant la création et la gestion de groupes avec des thèmes spécifiques. Les membres pourront être ajoutés ou retirés d'un groupe, et les publications pourront être associées à un groupe, les rendant visibles uniquement par les membres de ce groupe. Cela permettra une organisation ciblée des contenus en fonction des intérêts des utilisateurs, tout en garantissant une expérience utilisateur cohérente et personnalisée.

En résumé, la base de données offrira une solution complète pour gérer les utilisateurs, les publications, les interactions et les groupes thématiques, en respectant les règles de visibilité et d'appartenance définies, tout en facilitant les connexions sociales et les échanges entre utilisateurs.

2. Schéma de la base de données :

fig 1: Modèle conceptuel EA

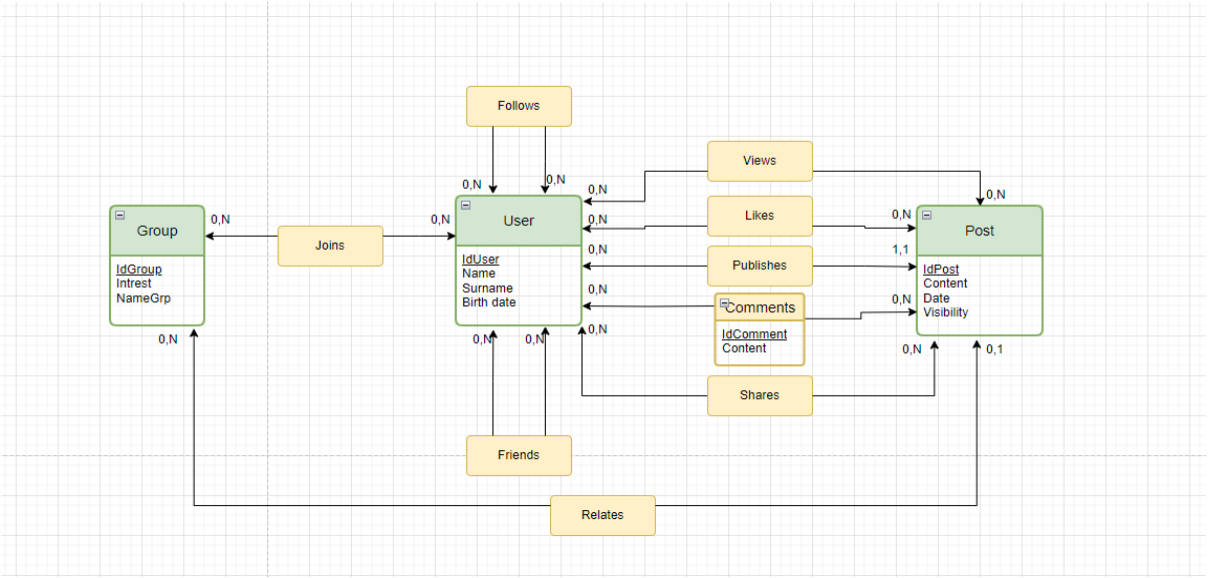
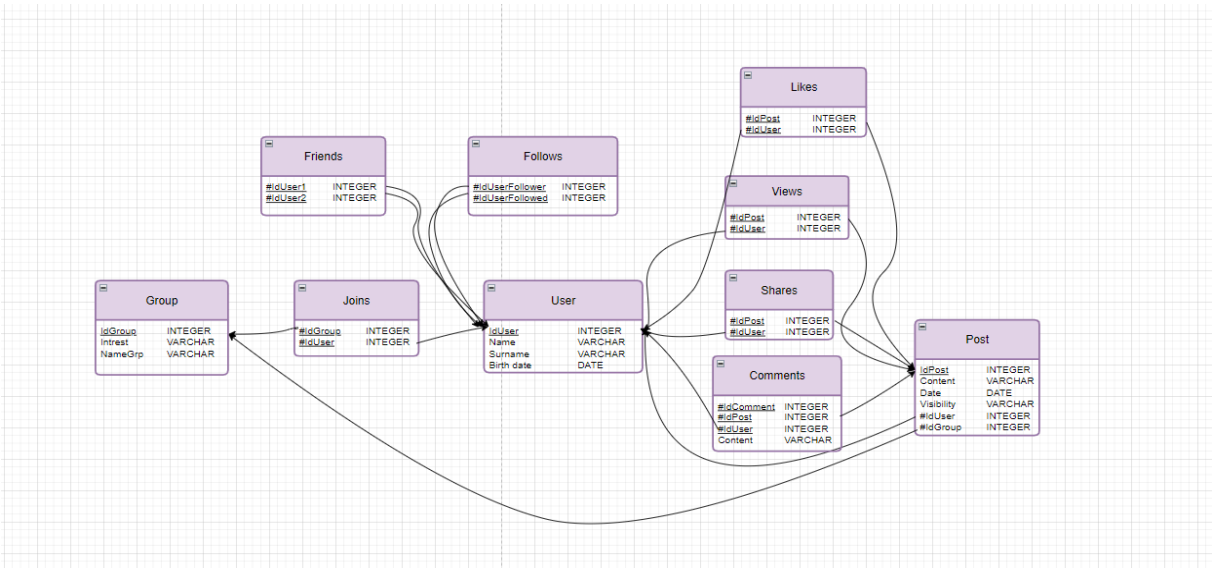


fig 2: Modèle logique



Les figures 1 et 2 représentent notre base de données, respectivement le modèle conceptuel de donnée et le modèle logique de donnée. Ceux-ci ont été rigoureusement normalisés en BCNF. En voici ci-dessous une justification pour chaque table :

1. Table User

- **1NF:** Chaque attribut contient des valeurs atomiques (par exemple, Name, Surname, Birth date sont des valeurs simples).
- **2NF:** La clé primaire est IdUser, toutes les colonnes dépendent entièrement de IdUser. Il n'y a pas de dépendance partielle.
- **3NF:** Aucune colonne non clé ne dépend d'une autre colonne non clé, donc il n'y a pas de dépendance transitive.
- **BCNF:** Il n'existe aucune dépendance fonctionnelle où une colonne non super-clé détermine une autre colonne. Ainsi, cette table est en BCNF.

2. Table Post

- **1NF**: Chaque attribut contient une valeur unique et atomique (exemple : Content, Date, Visibility).
- **2NF**: La clé primaire est IdPost, et toutes les colonnes dépendent entièrement de IdPost. Il n'y a pas de dépendance partielle.
- **3NF**: IdUser et IdGroup sont des clés étrangères, mais elles ne causent pas de dépendances transitives.

- **BCNF**: Toutes les dépendances fonctionnelles impliquent des super-clés, donc cette table est en BCNF.

3. Table Group

- **1NF**: Les attributs IdGroup, Intrest, et NameGrp sont atomiques.
- **2NF**: La clé primaire est IdGroup, et toutes les colonnes dépendent entièrement de IdGroup.
- **3NF**: Pas de dépendance transitive.
- **BCNF**: Aucune dépendance fonctionnelle d'une colonne non super-clé sur une autre colonne.

4. Table Friends

- **1NF**: Les paires (IdUser1, IdUser2) sont atomiques.
- **2NF**: La clé primaire est (IdUser1, IdUser2), et il n'y a pas de dépendance partielle.
- **3NF**: Aucune dépendance transitive entre les colonnes.
- **BCNF**: La seule dépendance fonctionnelle est (IdUser1, IdUser2) → {IdUser1, IdUser2}, donc la table est en BCNF.

5. Table Follows

- **1NF**: Les valeurs des attributs sont atomiques.
- **2NF**: La clé primaire est (IdUserFollower, IdUserFollowed), et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est (IdUserFollower, IdUserFollowed) → {IdUserFollower, IdUserFollowed}, donc la table est en BCNF.

6. Table Joins

- **1NF**: Les attributs IdGroup et IdUser sont atomiques.
- **2NF**: La clé primaire est (IdGroup, IdUser), et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est (IdGroup, IdUser) → {IdGroup, IdUser}, donc la table est en BCNF.

7. Table Likes

- **1NF**: Les attributs IdPost et IdUser sont atomiques.

- **2NF**: La clé primaire est (IdPost, IdUser), et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est (IdPost, IdUser) → {IdPost, IdUser}, donc la table est en BCNF.

8. Table Views

- **1NF**: Les attributs sont atomiques.
- **2NF**: La clé primaire est (IdPost, IdUser), et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est (IdPost, IdUser) → {IdPost, IdUser}, donc la table est en BCNF.

9. Table Shares

- **1NF**: Les attributs sont atomiques.
- **2NF**: La clé primaire est (IdPost, IdUser), et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est (IdPost, IdUser) → {IdPost, IdUser}, donc la table est en BCNF.

10. Table Comments

- **1NF**: Les attributs IdComment, IdPost, IdUser et Content sont atomiques.
- **2NF**: La clé primaire est IdComment, et toutes les colonnes dépendent entièrement de cette clé.
- **3NF**: Aucune dépendance transitive.
- **BCNF**: La seule dépendance fonctionnelle est IdComment → {IdPost, IdUser, Content}, donc la table est en BCNF.

Toutes les tables respectent les règles de BCNF car elles sont en 1NF, aucune colonne ne contenant de valeurs multiples. Elles sont également en 2NF, puisque toutes les colonnes dépendent entièrement de la clé primaire, et en 3NF, car aucune colonne ne dépend transitivement d'une autre. Enfin, elles

sont en BCNF, car toutes les dépendances fonctionnelles sont basées sur une super-clé. Ainsi, la base de données est bien normalisée en BCNF.

Choix de conception du modèle de données :

La table User constitue le cœur du système et stocke les informations essentielles de chaque utilisateur. Pour modéliser les relations entre les utilisateurs, nous avons créé deux tables distinctes : Follows et Friends. La table Follows permet de représenter les relations de suivi entre utilisateurs, tandis que la table Friends gère les relations d'amitié. Lors de l'implémentation, un mécanisme de gestion des doublons sera prévu pour éviter des redondances dans les relations d'amitié.

Les interactions entre les utilisateurs et les publications sont représentées par les tables Likes, Views, Shares et Comments. Nous avons fait le choix de créer une table Views dédiée, pour pouvoir récupérer le nombre de vues d'un post et par quel utilisateur. Cette décision vise à assurer une meilleure gestion des vues en ne comptabilisant qu'une seule fois la première visualisation d'un post par un utilisateur. De plus, cette table permet d'intégrer des contraintes garantissant la cohérence des interactions, comme l'empêchement d'un like ou d'un commentaire sur un post qu'un utilisateur n'aurait jamais consulté.

Enfin, la table Joins est utilisée pour représenter l'appartenance des utilisateurs aux groupes, qui sont eux-mêmes définis dans la table Group.

3. Requêtes SQL des différentes tables:

```sql
CREATE TABLE IF NOT EXISTS MyUser (
    IdUser INTEGER PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Surname VARCHAR(50) NOT NULL,
    BirthDate DATE NOT NULL
);

CREATE TABLE IF NOT EXISTS MyGroup (
    IdGroup INTEGER PRIMARY KEY,
    Interest VARCHAR(50) NOT NULL,
```

```
    NameGrp VARCHAR(50) NOT NULL
);

CREATE TABLE IF NOT EXISTS Friends (
    IdUser1 INTEGER,
    IdUser2 INTEGER,
    PRIMARY KEY (IdUser1, IdUser2),
    FOREIGN KEY (IdUser1) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
    FOREIGN KEY (IdUser2) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
    CHECK (IdUser1 < IdUser2)
);

CREATE TABLE IF NOT EXISTS Follows (
    IdUserFollower INTEGER,
    IdUserFollowed INTEGER,
    PRIMARY KEY (IdUserFollower, IdUserFollowed),
    FOREIGN KEY (IdUserFollower) REFERENCES MyUser(IdUser) ON
DELETE CASCADE,
    FOREIGN KEY (IdUserFollowed) REFERENCES MyUser(IdUser) ON
DELETE CASCADE,
    CHECK (IdUserFollower <> IdUserFollowed)
);

CREATE TABLE IF NOT EXISTS Joins (
    IdGroup INTEGER,
    IdUser INTEGER,
    PRIMARY KEY (IdGroup, IdUser),
    FOREIGN KEY (IdGroup) REFERENCES MyGroup(IdGroup) ON
DELETE CASCADE,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE
);

DO $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
```

```
        FROM pg_type
        WHERE typname = 'visibility_enum'
    ) THEN
        CREATE TYPE visibility_enum AS ENUM ('public',
'private');
    END IF;
END $$;

CREATE TABLE IF NOT EXISTS Post (
    IdPost INTEGER PRIMARY KEY,
    Content VARCHAR(255) NOT NULL,
    Date DATE NOT NULL,
    Visibility visibility_enum NOT NULL,
    IdUser INTEGER NOT NULL,
    IdGroup INTEGER,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
    FOREIGN KEY (IdGroup) REFERENCES MyGroup(IdGroup) ON
DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS MyViews (
    IdPost INTEGER,
    IdUser INTEGER,
    PRIMARY KEY (IdPost, IdUser),
    FOREIGN KEY (IdPost) REFERENCES Post(IdPost) ON DELETE
CASCADE,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE
);

CREATE TABLE IF NOT EXISTS Likes (
    IdPost INTEGER,
    IdUser INTEGER,
    PRIMARY KEY (IdPost, IdUser),
    FOREIGN KEY (IdPost) REFERENCES Post(IdPost) ON DELETE
CASCADE,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
```

```
    FOREIGN KEY (IdPost, IdUser) REFERENCES MyViews(IdPost,
IdUser) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Shares (
    IdPost INTEGER,
    IdUser INTEGER,
    PRIMARY KEY (IdPost, IdUser),
    FOREIGN KEY (IdPost) REFERENCES Post(IdPost) ON DELETE
CASCADE,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
    FOREIGN KEY (IdPost, IdUser) REFERENCES MyViews(IdPost,
IdUser) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS MyComments (
    IdComment INTEGER PRIMARY KEY,
    IdPost INTEGER NOT NULL,
    IdUser INTEGER NOT NULL,
    Content VARCHAR(255) NOT NULL,
    FOREIGN KEY (IdPost) REFERENCES Post(IdPost) ON DELETE
CASCADE,
    FOREIGN KEY (IdUser) REFERENCES MyUser(IdUser) ON DELETE
CASCADE,
    FOREIGN KEY (IdPost, IdUser) REFERENCES MyViews(IdPost,
IdUser) ON DELETE CASCADE
);
```

Les **contraintes avancées** dans ce schéma introduisent des règles métier spécifiques et des dépendances complexes entre les tables :

1. `CHECK (IdUser1 < IdUser2) dans Friends` :
   Cette contrainte évite les doublons symétriques dans les relations d'amitié. Par exemple, si (1, 2) existe, (2, 1) ne peut pas être ajouté, garantissant une représentation unique de chaque amitié.
2. `CHECK (IdUserFollower <> IdUserFollowed) dans Follows` :
   Cette contrainte empêche un utilisateur de se suivre lui-même, ce qui serait illogique dans un contexte de suivi.

3. **Dépendance via `MyViews` dans `Likes`, `Shares`, et `MyComments`** :
   Ces tables imposent qu'un utilisateur ne peut **aimer**, **partager** ou **commenter** un post que s'il l'a **vu** (via la référence à `MyViews`). Cela crée une hiérarchie stricte des interactions :
   - Un utilisateur doit d'abord voir un post (`MyViews`) avant de pouvoir interagir avec (`Likes`, `Shares`, `MyComments`).
4. `ON DELETE CASCADE` **sur les clés étrangères** :
   Cette règle garantit que la suppression d'un enregistrement parent (exemple : un utilisateur dans `MyUser`) entraîne la suppression automatique de tous les enregistrements dépendants dans les tables liées (exemple : ses posts, commentaires, likes, etc.). Cela maintient la cohérence de la base de données sans laisser d'orphelins.

Ces contraintes avancées structurent les interactions entre les tables et imposent des règles métier strictes, garantissant un système robuste et cohérent.

4. Requêtes SQL des services mentionné:

- Quels utilisateurs influencent le plus les interactions dans un groupe donné ? (Youssef Ben Mzoughia)

```
CREATE OR REPLACE FUNCTION GetTopInfluencersInGroup(groupId
INT)
RETURNS TABLE(
    UserName VARCHAR(50),
    UserSurname VARCHAR(50),
    InfluenceCount BIGINT
) AS $$
BEGIN
    RETURN QUERY
    SELECT u.Name, u.Surname, COUNT(*) AS InfluenceCount
    FROM MyUser u
    JOIN (
        SELECT IdUser FROM Likes WHERE IdPost IN (SELECT
IdPost FROM Post WHERE IdGroup = groupId)
        UNION ALL
        SELECT IdUser FROM Shares WHERE IdPost IN (SELECT
```

```
IdPost FROM Post WHERE IdGroup = groupId)
        UNION ALL
        SELECT IdUser FROM MyComments WHERE IdPost IN (SELECT
IdPost FROM Post WHERE IdGroup = groupId)
    ) AS Interactions ON u.IdUser = Interactions.IdUser
    GROUP BY u.IdUser
    ORDER BY InfluenceCount DESC;
END;
$$ LANGUAGE plpgsql;
```

La fonction `GetTopInfluencersInGroup` permet d'identifier les utilisateurs les plus influents au sein d'un groupe spécifique sur une plateforme sociale, en mesurant leur engagement à travers le cumul de leurs interactions (likes, partages et commentaires) sur les publications associées à ce groupe. Concrètement, elle retourne un classement des membres, affichant leur prénom, nom et nombre total d'interactions, triés par ordre décroissant d'influence. Pour y parvenir, la fonction combine les données des tables `Likes`, `Shares` et `MyComments` à l'aide d'une sous-requête utilisant `UNION ALL`, une opération qui agrège toutes les interactions sans supprimer les doublons, essentielle pour comptabiliser les actions multiples d'un même utilisateur sur un post. Les posts analysés sont filtrés dès la sous-requête pour ne retenir que ceux appartenant au groupe cible (`groupId`), optimisant ainsi le traitement des données. Les résultats sont ensuite regroupés par utilisateur et ordonnés pour mettre en évidence les contributeurs les plus actifs.

- Quels utilisateurs ont des connexions indirectes (amis d'amis) avec un utilisateur donné, et quels sont leurs intérêts communs ? (Fares Kobbi)

```
WITH RECURSIVE IndirectConnections AS (
     SELECT IdUser2 AS FriendId
     FROM Friends
     WHERE IdUser1 = 1
     UNION
     SELECT f.IdUser2
     FROM Friends f
```

```
    JOIN IndirectConnections ic ON f.IdUser1 = ic.FriendId
)

SELECT DISTINCT ic.FriendId, g.Interest
FROM IndirectConnections ic
JOIN Joins j ON ic.FriendId = j.IdUser
JOIN MyGroup g ON j.IdGroup = g.IdGroup
WHERE g.Interest IN (
    SELECT g2.Interest
    FROM Joins j2
    JOIN MyGroup g2 ON j2.IdGroup = g2.IdGroup
    WHERE j2.IdUser = 1
);
```

Cette requête SQL utilise une Common Table Expression (CTE) récursive pour identifier les connexions indirectes (amis d'amis) d'un utilisateur donné, `IdUser1`. La CTE **`IndirectConnections`** commence par récupérer les amis directs de l'utilisateur, puis, de manière récursive, trouve les amis de ces amis, élargissant ainsi le réseau de connexions. Ensuite, la requête joint ces connexions indirectes avec les tables **`Joins`** et **`MyGroup`** pour déterminer les groupes auxquels ces amis indirects appartiennent et les intérêts associés. Enfin, elle filtre ces intérêts pour ne conserver que ceux qui sont partagés avec l'utilisateur donné, en comparant avec les intérêts de ce dernier via une sous-requête. Le résultat final est une liste distincte de paires (**`FriendId`**, **`Interest`**), où **`FriendId`** représente un ami indirect et **`Interest`** un intérêt commun avec l'utilisateur donné. Ainsi, la requête répond à la question en identifiant à la fois les connexions indirectes et les intérêts partagés.*

- Comment intégrer un nouvel utilisateur et ses connexions dans le réseau existant ? (Hammoud Younes)

```
BEGIN;

INSERT INTO MyUser (IdUser, Name, Surname, BirthDate) VALUES (17, 'New',
'User', '2000-01-01');

INSERT INTO Friends (IdUser1, IdUser2) VALUES
```

```
(17, 1),
(17, 2);

INSERT INTO Follows (IdUserFollower, IdUserFollowed) VALUES
(3, 17),
(4, 17),
(17, 5),
(17, 6);

COMMIT;
```

Cette requête ajoute un nouvel utilisateur (`IdUser = 17`) à la base de données avec son nom, prénom et date de naissance. Elle établit ensuite des relations d'amitié avec les utilisateurs 1 et 2, en insérant ces connexions dans la table `Friends`, qui représente des relations réciproques. En parallèle, la requête ajoute des relations de suivi dans la table `Follows`, où les utilisateurs 3 et 4 suivent 17, tandis que 17 suit les utilisateurs 5 et 6. Contrairement aux amis, le suivi est une relation unidirectionnelle. Enfin, la transaction est validée avec `COMMIT`, garantissant que toutes les modifications sont définitivement enregistrées.

5.Description du jeu de données(tables/populate.sql):

Le jeu de données utilisé pour valider les requêtes SQL a été conçu pour couvrir un éventail de scénarios typiques tout en respectant les contraintes d'intégrité de la base de données. Bien que sa taille soit volontairement réduite (16 utilisateurs, 12 groupes, 14 posts), il garantit une complétude fonctionnelle grâce aux éléments suivants :

❖ Diversité des utilisateurs et des groupes

- 16 utilisateurs avec des profils variés (dates de naissance, noms, prénoms) pour simuler une communauté active.
- 12 groupes organisés autour de trois intérêts principaux (Technologie, Sports, Musique), incluant des groupes similaires (ex. : "Tech Enthusiasts" vs "Tech Innovators") pour tester les distinctions entre entités partageant un même thème.

❖ Relations sociales complexes

- Amis mutuels et indirects (ex. John est ami avec Jane, Alice et Bob ; Jane est amie avec Alice et Bob) pour tester les requêtes impliquant des réseaux interconnectés.
- Follows unidirectionnels (ex. John suit Jane, mais Jane ne suit pas John) pour valider les flux d'actualités ou les recommandations.

❖ Interactions réalistes avec les posts

- 14 posts avec des visibilités mixtes (public vs privé) et des associations à des groupes ou à des utilisateurs individuels.

❖ Contraintes d'interaction respectées :

- Un utilisateur ne peut liker/partager/commenter que les posts qu'il a préalablement vus (ex. : Jane like le post de John après l'avoir vu).
- Les clés étrangères (ex. : IdGroup dans Post) garantissent que les interactions sont liées à des entités valides.

6. Analyse critique du déroulement du projet:

Répartition du travail:

Tous les membres du groupe: Modèle EA et relationnel, jeu de donnés

Fares Kobbi : Questions 1, 5, 6

Youssef Ben Mzoughia : Questions 2, 4

Hammoud Younes : Questions 3, 7

Déroulement du projet:

Le projet a été mené en équipe avec un découpage des tâches basé sur les compétences de chacun. La conception de la base de données et l'écriture des requêtes SQL ont été réparties entre plusieurs membres afin d'optimiser le temps de développement. Fares avait commencé à préparer des questions pour le client pour clarifier certaines notions ambiguës. Entre-temps, Youssef et Hammoud ont commencé à produire une première itération du modèle EA qu'on avait modifié après la visite du client. Une fois le

modèle relationnel était bien prêt et normalisé, nous avions travaillé sur les requêtes SQL de création de table et de leurs populations .Cependant, certaines difficultés ont émergé, notamment une coordination parfois insuffisante, entraînant des chevauchements de travail ou des incohérences dans les relations entre les tables.

Pour améliorer l'organisation, une meilleure planification initiale avec des réunions plus fréquentes auraient permis d'assurer une meilleure cohésion. De plus, la mise en place de revues de code régulières aurait aidé à identifier plus tôt les erreurs et à harmoniser les pratiques.

Malgré ces défis, plusieurs aspects du projet ont été une réussite. La collaboration entre les membres a permis une montée en compétences mutuelle, et l'implication de chacun a contribué à la finalisation du projet dans les délais.

Si c'était à refaire, nous mettrons en place une gestion de projet plus structurée, avec un découpage des tâches plus précis et un suivi rigoureux de l'avancement. Une explication du métier plus détaillée et une meilleure communication entre les membres garantiraient une intégration plus fluide et une réduction des erreurs.