RAPPORT DE PROJET

SI40

Piana Victor

Jury Cyprien

Chairi Hamza

JUIN 2024

SOMMAIRE

LISTE DES FIGURES

REMERCIEMENTS

Nous tenons à exprimer notre profonde gratitude à Mme Christine Lahoud, notre professeur de l'unité de valeur SI40 à l'UTBM, pour son soutien, ses conseils avisés et son encadrement tout au long de ce projet. Son expertise et sa passion pour les bases de données ont été une source d'inspiration constante et ont grandement contribué à notre apprentissage et à la réussite de notre travail. Nous la remercions sincèrement pour sa patience, sa disponibilité et pour avoir créé un environnement d'apprentissage stimulant et enrichissant.

PRESENTATION DU CADRE GENERAL

INTRODUCTION

Dans ce rapport pour l'unité de valeur SI40 de l'Université de Technologie de Belfort-Montbéliard, nous explorerons diverses notions essentielles relatives aux bases de données en informatique. Ce travail s'inscrit dans un contexte pédagogique double, car il est réalisé parallèlement à l'UV WE4, qui implique la création d'un site web sous forme de réseau social. La conception d'une base de données pour un réseau social représente un défi particulièrement stimulant et enrichissant, s'insérant parfaitement dans notre parcours de formation d'ingénieurs.

Ce rapport détaillera notre approche face à la problématique posée par ce projet, en décrivant notre environnement de travail et notre processus de conception. Nous illustrerons notre démarche à travers les diagrammes de modèles conceptuels de données (MCD) et de modèles logiques de données (MLD), qui ont guidé la structuration de notre base de données. En outre, nous présenterons des tests effectués directement sur la base à l'aide de requêtes SQL, ainsi qu'une analyse OLAP permettant d'exploiter les données dans un contexte décisionnel.

L'objectif de ce rapport est de montrer comment la théorie et les techniques de modélisation des données ont été appliquées concrètement pour répondre aux besoins spécifiques d'un réseau social, soulignant ainsi l'importance de la maîtrise des bases de données dans le domaine de l'ingénierie.

CADRE DU PROJET

Problématique

Le développement d'une base de données pour un réseau social dans le cadre de notre formation d'ingénieurs en informatique pose des défis uniques et complexes. Nous sommes confrontés à plusieurs questions essentielles :

1. Comment concevoir une base de données répondant aux besoins d'un réseau social ?
2. Quels dispositifs peut-on mettre en place pour s'assurer que les données des utilisateurs restent intactes et privées ?
3. En quoi les décisions prises lors de la création de la base de données affectent-elles sa facilité de gestion et sa capacité à évoluer avec le temps ?

Méthodologie

Pour aborder cette problématique, nous avons adopté une approche méthodique structurée en plusieurs étapes clés :

1. Analyse des Besoins : Identification des exigences fonctionnelles et non fonctionnelles du système.
2. Modélisation : Conception des modèles conceptuel et logique de données (MCD et MLD) pour structurer efficacement les informations.
3. Implémentation : Mise en place de la base de données en utilisant PostgreSQL.
4. Tests : Réalisation de tests pour vérifier l'adéquation du système avec les exigences identifiées.
5. Analyse OLAP : Implémentation de solutions d'analyse en ligne (OLAP) pour extraire des insights à partir des données, facilitant la prise de décision stratégique.

Les acteurs du système

Les principaux acteurs impliqués dans ce projet sont :

1. Étudiants en Informatique : Responsables de la conception, du développement, et des tests du système.
2. Utilisateurs Imaginaires du Réseau Social : Acteurs clés qui interagissent avec le système, fournissant les inputs principaux pour la base de données.
3. Professeure Superviseure : Fournie l'encadrement académique, les conseils techniques, et supervise l'avancement du projet.

ENVIRONNEMENT DE TRAVAIL

Langage de développement

Dans ce projet, nous avons principalement utilisé SQL et PL/pgSQL, qui est bien adapté pour la maipulation de base de donnée en raison de sa robustesse et de sa capacité à interagir avec PostgreSQL.

Logiciels et Frameworks

Nous avons utilisé Win Design pour créer les modèles. Ensuite nous avons utilisé PgAdmin4. Et pour finir nous avons utilisé Power BI pour la partie Olap.

Répartition du travail (entre les membres du groupe)

Pour la répartition du travail, la décision s'est prise assez spontanément. Victor a commencé par prendre en charge les modèles MCD et MLD sur Win Design, qu'il a ensuite partagés avec les membres du groupe. Cyprien s'est occupé de la partie PgAdmin et des requêtes SQL(aidé ensuite par Victor), tandis qu'Hamza a géré la partie OLAP. Concernant le rapport, Hamza a rédigé la section OLAP et Cyprien ainsi que Victor se sont partagé le reste.

ANALYSE DES BESOINS ET SPECIFICATIONS

ANALYSE DES BESOINS (CAHIER DE CHARGE/UML)

Les besoins sont vite clairs puisque nous connaissons bien les réseaux sociaux. Malgré tout, on ne peut pas bien se rendre compte de tout ce qu'il y a derrière. Mais finalement nous en sommes arrivé à la conclusion qu'il y a plusieurs choses indispensable. L'utilisateur peut faire des postes, aimer ou commenter des postes, lui faisant donc recevoir une notification. Il peut également suivre un autre utilisateur. En partant de ceci avec quelques précisions et prise de conscience pendant la modélisation. Nous en sommes arrivés au modèles suivants.

MODELISATION (MCD ET MLD)

En annexe (figure 20 et figure 21), vous pouvez voir notre première version des deux diagrammes ci-dessous, fait au début de semestre .

Voici notre modèle MCD final :

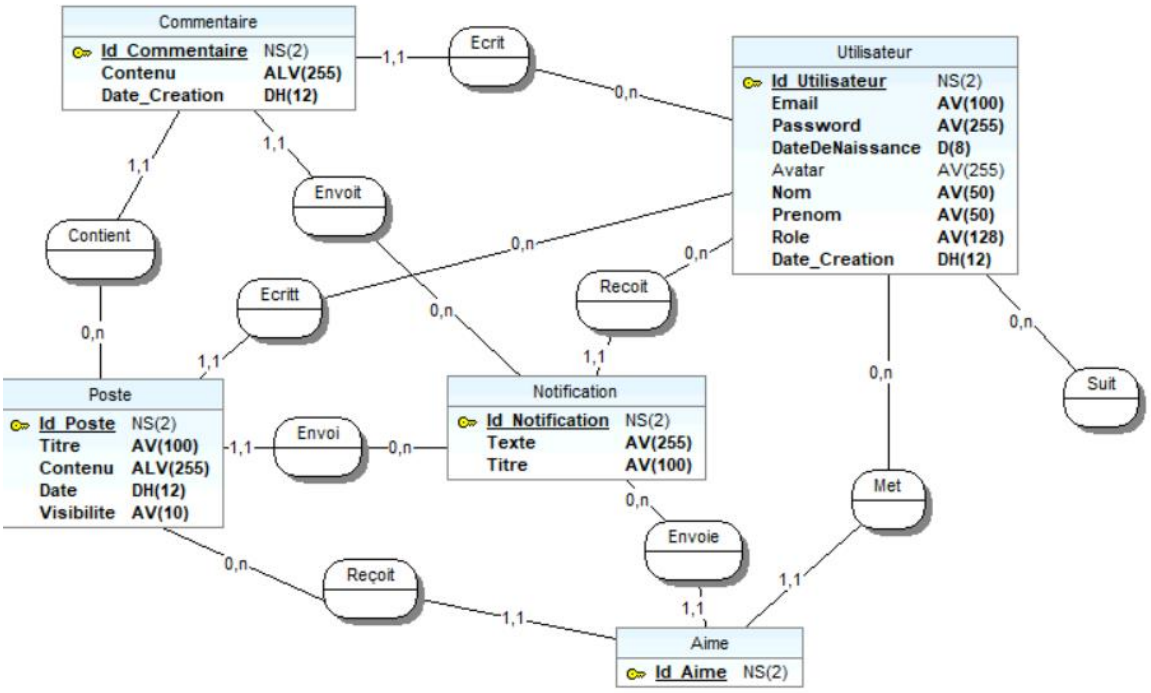


Figure 1: Modele MCD

Dans notre projet, un Utilisateur peut écrire 0 à plusieurs Commentaires, recevoir 0 à plusieurs Notifications, et écrire 0 à plusieurs Postes. L'Utilisateur peut également mettre 0 à plusieurs Likes et suivre 0 à plusieurs autres Utilisateurs.

Un Commentaire est forcément écrit par un seul Utilisateur et est contenu dans un unique Poste.

Un Poste est écrit par un seul Utilisateur, peut recevoir 0 à plusieurs Likes, et peut contenir 0 à plusieurs Commentaires.

Un Like est mis par un seul et même Utilisateur, et est reçu sur un seul et même Poste.

Une Notification est reçue par un seul et même Utilisateur.

Un Utilisateur peut suivre 0 à plusieurs autres Utilisateurs.

Tout cela est lié à une table de Notifications de sorte que l'Utilisateur reçoive une notification si besoin.

L'ensemble de ces contraintes nous a donc amenés à ce modèle MCD :

Ce modèle relationnel MCD capture toutes les interactions possibles entre les entités de notre système.

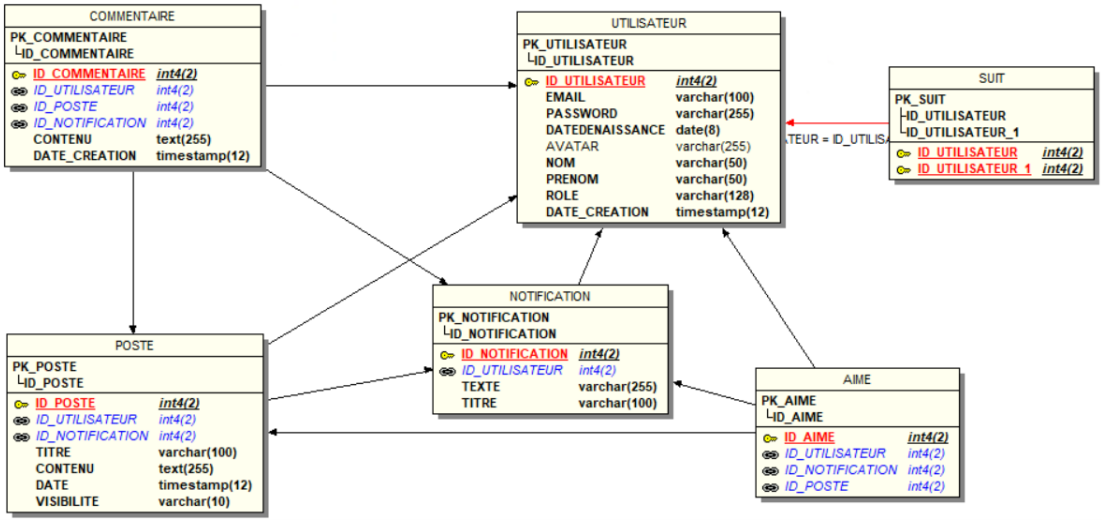


Figure 2: Modele MLD

PARTIE POSTGRESQL

DESCRIPTION ET CODE DES REQUETES SQL

1.Insert

Query | Query History

```
1  -- Un exemple d'INSERT avec notre database:
2  INSERT INTO UTILISATEUR (ID_UTILISATEUR, EMAIL, PASSWORD, DATEDENAISSANCE, AVATAR, NOM, PRENOM, ROLE, DATE_CREATION)
3  VALUES (5, 'lucas.martin@example.com', 'password987', '1985-05-05', 'http://example.com/avatar5.jpg',
4          'Martin', 'Lucas', TRUE, '2023-01-05 14:00:00');
5
6
```

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 33 msec.
```

Figure 3: Exemple INSERT

INSERT est une requête SQL qui va nous permettre d'insérer des données dans une table. Dans cet exemple, on insère toutes les données d'un nouvel utilisateur dans la table Utilisateur.

2.Select

Query | Query History

```
1  -- Sélection de tous les utilisateurs
2  SELECT * FROM UTILISATEUR;
3
4  -- Sélection des utilisateurs avec un rôle spécifique (ceux qui sont admin)
5  SELECT * FROM UTILISATEUR WHERE ROLE = TRUE;
6
```

Data Output | Messages | Notifications

	id_utilisateur [PK] integer	email character varying (100)	password character varying (255)	datedenaissance date	avatar character varying (255)	nom character varying (50)	prenom character varying (50)	role boolean
1	1	john.doe@example.com	password123	1980-01-01	http://example.com/avatar1.jpg	Doe	John	true
2	3	alice.jones@example.com	password789	2000-03-03	http://example.com/avatar3.jpg	Jones	Alice	true
3	5	lucas.martin@example.com	password987	1985-05-05	http://example.com/avatar5.jpg	Martin	Lucas	true

Figure 3: Exemple SELECT

SELECT est une requête SQL qui va nous permettre d'extraire des données d'une table de notre base de donnée. Ici par exemple, la deuxième requête nous permet d'extraire toutes les données des utilisateurs qui possèdent un rôle.

3.Update

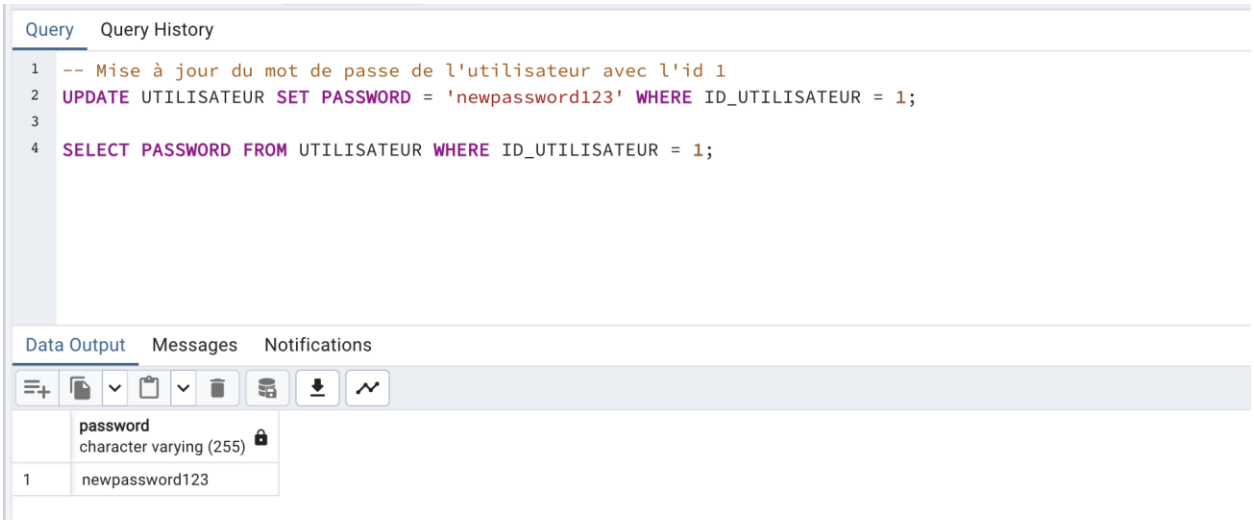


Figure 4: Exemple UPDATE

UPDATE est une requête SQL qui va nous permettre de mettre à jour des données au sein d'une table. Dans l'exemple ci-dessus, on utilise UPDATE pour modifier le mot de passe de l'utilisateur avec l'id 1 en « newpassword123 ».

4.Delete

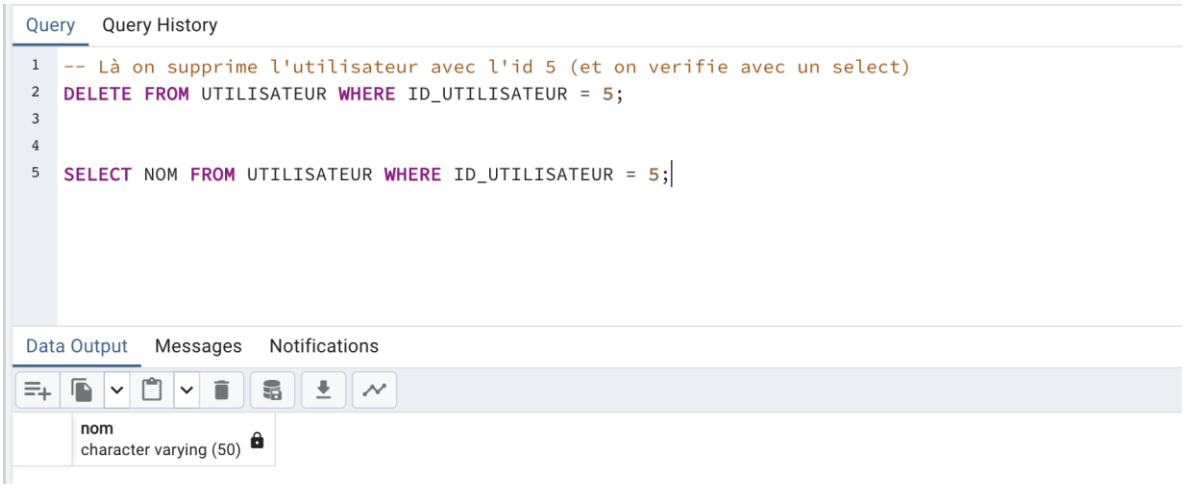


Figure 5: Exemple DELETE

DELETE est une requête SQL qui va nous permettre de supprimer un ou plusieurs enregistremenets d'une table. Ci-dessus, on utilise DELETE pour supprimer l'utilisateur possédant l'id 5, puis on vérifie qu'il a bien été supprimé avec un SELECT.

5.Fonction d'agrégation

Somme :

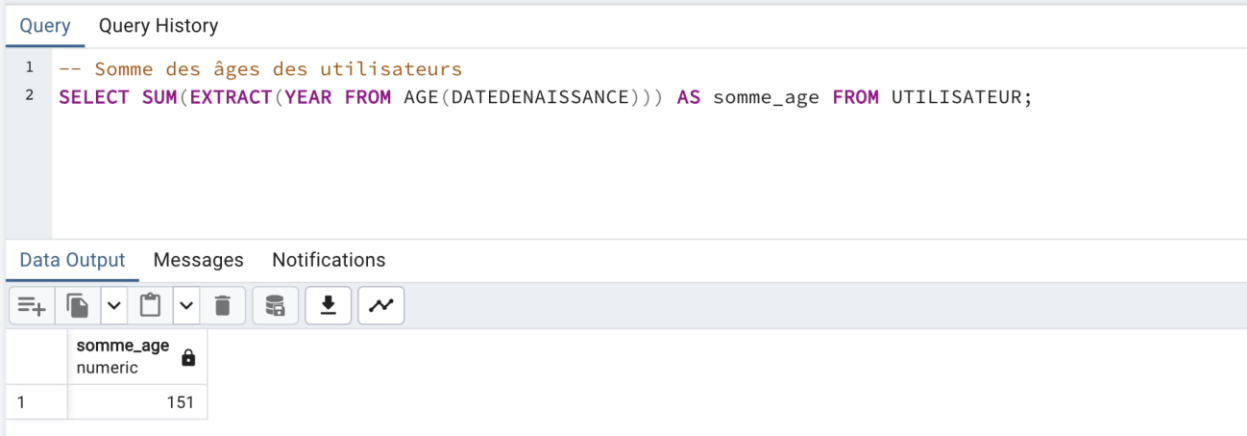


Figure 6: Exemple SUM

SUM en SQL nous permet de récupérer la somme de toutes les valeurs de l'expression. Dans la capture d'écran ci-dessus, nous récupérons les âges des utilisateurs grâce à leur date de naissance puis nous récupérons la somme des âges des utilisateurs grâce à SELECT SUM.

Moyenne :

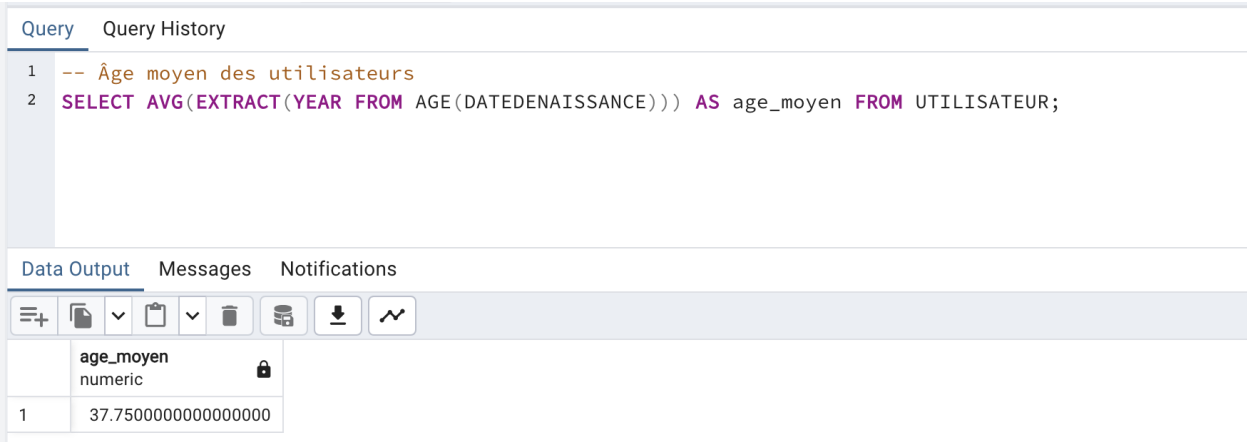


Figure 7: Exemple AVG

AVG en SQL nous permet d'extraire la moyenne des valeurs d'un groupe de données. Ci-dessus, nous avons utilisés SELECT AVG pour récupérer la moyenne des âges de tous les utilisateurs.

Count :

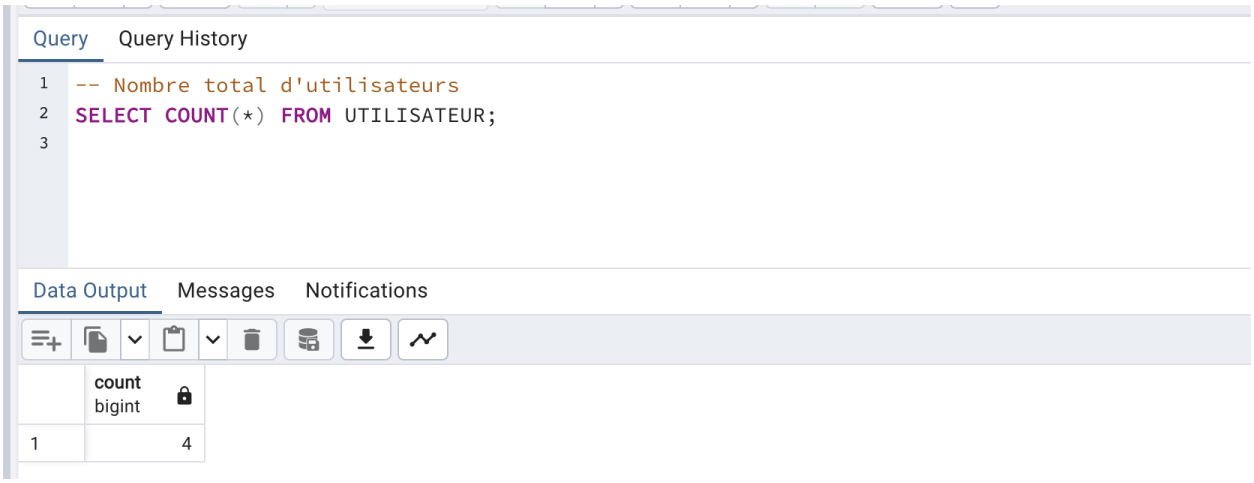


Figure 8: Exemple COUNT

COUNT en SQL nous permet de calculer le nombre total d'enregistrement, y compris ceux qui contiennent des champs NULL. Nous avons utilisés SELECT COUNT ci-dessus pour récupérer le nombre total d'utilisateurs.

6.Index

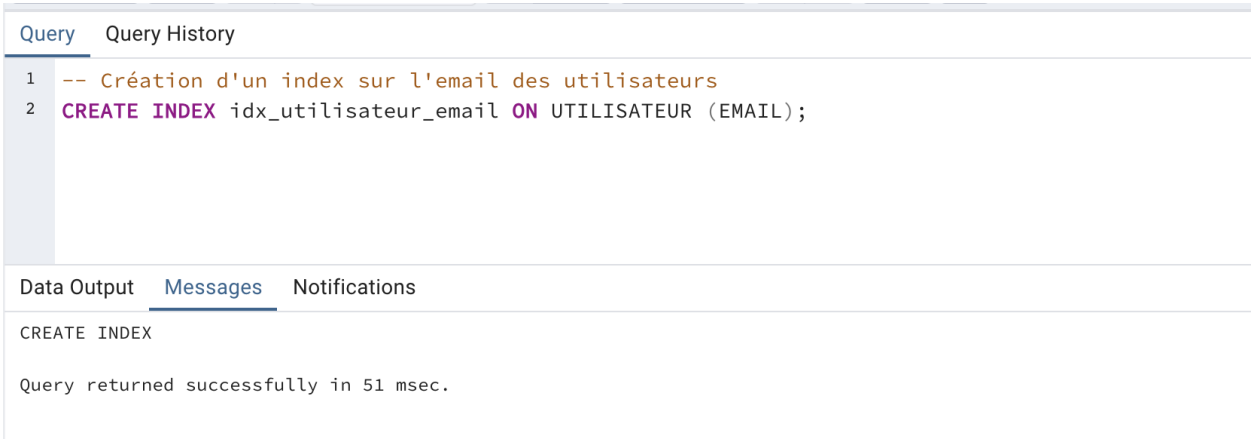


Figure 9: Exemple INDEX

INDEX est une requête SQL qui nous permet de créer un index, permettant ainsi de localiser rapidement les données, réduisant le temps nécessaire pour obtenir les résultats. Dans l'exemple ci-dessus, nous avons créés un index sur l'email des utilisateurs : idx_utilisateur_email.

7.Procedure

```
Query  Query History
1  -- Création d'une procédure pour ajouter un nouvel utilisateur
2  CREATE OR REPLACE PROCEDURE add_utilisateur(
3      id_utilisateur INTEGER,
4      email VARCHAR,
5      password VARCHAR,
6      datedenaissance DATE,
7      avatar VARCHAR,
8      nom VARCHAR,
9      prenom VARCHAR,
10     role BOOLEAN,
11     date_creation TIMESTAMP
12 ) LANGUAGE plpgsql AS $$
13 BEGIN
14     INSERT INTO UTILISATEUR (ID_UTILISATEUR, EMAIL, PASSWORD, DATEDENAISSANCE, AVATAR, NOM, PRENOM, ROLE, DATE_CREATION)
15     VALUES (id_utilisateur, email, password, datedenaissance, avatar, nom, prenom, role, date_creation);
16 END;
17 $$;
18
19

Data Output  Messages  Notifications
CREATE PROCEDURE

Query returned successfully in 52 msec.
```

Figure 10: Exemple PROCEDURE

PROCEDURE en SQL va nous permettre de créer une nouvelle procédure, qui pourra ainsi être exécutée rapidement au moment de l'appel. Ci-dessus, nous avons créés une procédure pour ajouter un nouvel utilisateur dans la table utilisateur.

Maintenant faisons un appel à cette procédure et vérifions si elle marche :

```
Query  Query History
1  -- Appel de la procédure add_utilisateur pour ajouter un nouvel utilisateur
2  CALL add_utilisateur(
3      5,                          -- id_utilisateur
4      'marie.curie@example.com',  -- email
5      'curie123',                 -- password
6      '1867-11-07',               -- datedenaissance
7      'http://example.com/avatar7.jpg', -- avatar
8      'Curie',                    -- nom
9      'Marie',                    -- prenom
10     TRUE,                       -- role
11     '2024-06-08 10:00:00'       -- date_creation
12 );
13
14 -- On vérifie que l'appel de notre procédure a bien marché avec marie curie
15 SELECT * FROM UTILISATEUR WHERE EMAIL = 'marie.curie@example.com';
16
17
18

Data Output  Messages  Notifications
```

	id_utilisateur [PK] integer	email character varying (100)	password character varying (255)	datedenaissance date	avatar character varying (255)	nom character varying (50)	prenom character varying (50)	role boolean
1	5	marie.curie@example.com	curie123	1867-11-07	http://example.com/avatar7.jpg	Curie	Marie	true

Figure 11: Vérif PROCEDURE

On appelle la procédure grâce à CALL et on vérifie avec un SELECT.

8.Trigger

```
Query  Query History
1  -- Création de la fonction et du trigger
2  CREATE OR REPLACE FUNCTION check_email() RETURNS TRIGGER AS $$
3  BEGIN
4      IF NEW.EMAIL !~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$' THEN
5          RAISE EXCEPTION 'Invalid email format: %', NEW.EMAIL;
6      END IF;
7      RETURN NEW;
8  END;
9  $$ LANGUAGE plpgsql;
10
11 CREATE TRIGGER trg_check_email
12 BEFORE INSERT OR UPDATE ON UTILISATEUR
13 FOR EACH ROW EXECUTE FUNCTION check_email();
14
15
16

Data Output  Messages  Notifications
CREATE TRIGGER

Query returned successfully in 75 msec.
```

Figure 12: Exemple TRIGGER

TRIGGER en SQL va nous permettre d'exécuter un ensemble d'instruction SQL juste après un évènement. Ci-dessus, nous utilisons trigger pour vérifier si l'email saisit est valide lorsque l'on rentre un utilisateur dans la table. Si l'email n'est pas valide, alors l'erreur « Invalid email format » apparaîtra.

Maintenant on vérifie en essayant d'insérer un User avec un email invalide :

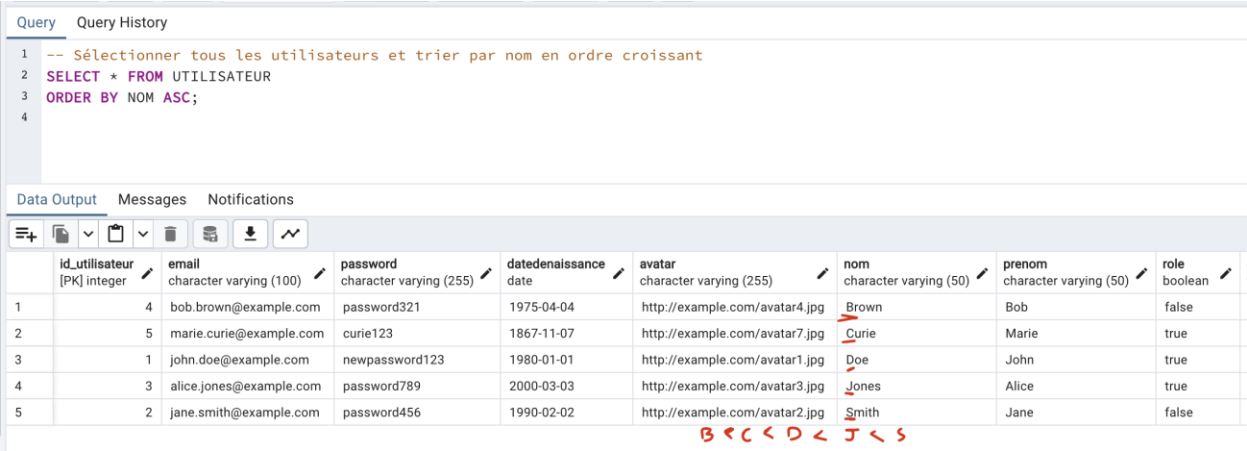
```
Query  Query History
1  -- Tentative d'insertion d'un utilisateur avec un email invalide
2  BEGIN;
3  INSERT INTO UTILISATEUR (EMAIL, PASSWORD, DATEDENAISSANCE, AVATAR, NOM, PRENOM, ROLE, DATE_CREATION)
4  VALUES ('invalid-email', 'password123', '1980-01-01', 'http://example.com/avatar1.jpg', 'Doe', 'John', TRUE, '2024-06-08 10:00:00');
5  COMMIT;
6
7  -- Requête pour vérifier que l'utilisateur avec email invalide n'a pas été inséré
8  SELECT * FROM UTILISATEUR WHERE EMAIL = 'invalid-email';
9
10

Data Output  Messages  Notifications
ERROR:  Invalid email format: invalid-email
CONTEXT:  PL/pgSQL function check_email() line 4 at RAISE

SQL state: P0001
```

Figure 13: Vérif TRIGGER

On peut voir que ça ne marche effectivement pas et que l'on obtient une erreur, donc le trigger est bien fonctionnel.

9.Tri

Grâce à ORDER BY en SQL, on peut trier les valeurs, soit en utilisant ASC (par ordre croissant) :



Query | Query History

```
1  -- Sélectionner tous les utilisateurs et trier par nom en ordre croissant
2  SELECT * FROM UTILISATEUR
3  ORDER BY NOM ASC;
4
```

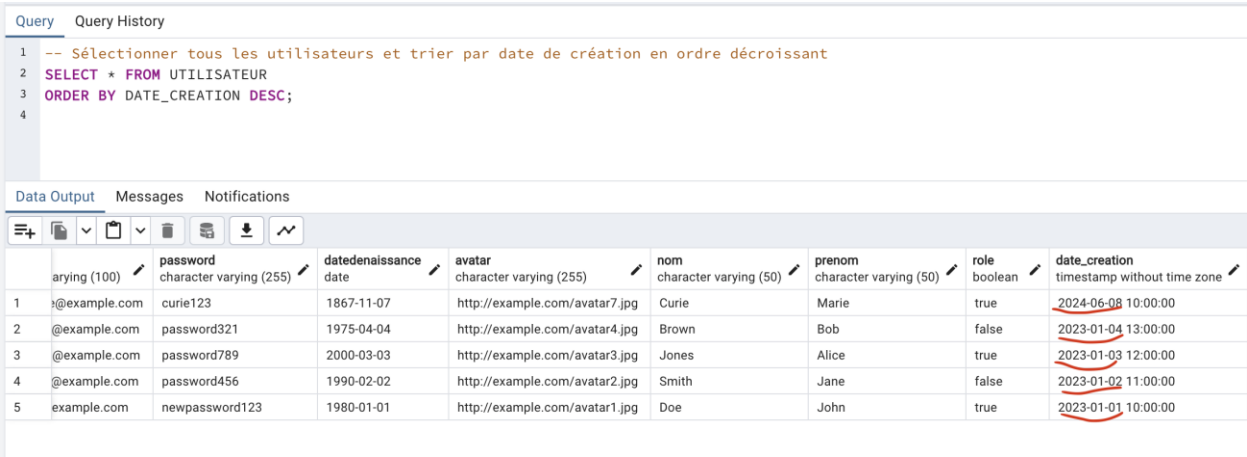
Data Output | Messages | Notifications

	id_utilisateur [PK] integer	email character varying (100)	password character varying (255)	datedenaissance date	avatar character varying (255)	nom character varying (50)	prenom character varying (50)	role boolean
1	4	bob.brown@example.com	password321	1975-04-04	http://example.com/avatar4.jpg	Brown	Bob	false
2	5	marie.curie@example.com	curie123	1867-11-07	http://example.com/avatar7.jpg	Curie	Marie	true
3	1	john.doe@example.com	newpassword123	1980-01-01	http://example.com/avatar1.jpg	Doe	John	true
4	3	alice.jones@example.com	password789	2000-03-03	http://example.com/avatar3.jpg	Jones	Alice	true
5	2	jane.smith@example.com	password456	1990-02-02	http://example.com/avatar2.jpg	Smith	Jane	false

B < C < D < J < S

Figure 14 Exemple ASC

ou DESC (par ordre décroissant) :



Query | Query History

```
1  -- Sélectionner tous les utilisateurs et trier par date de création en ordre décroissant
2  SELECT * FROM UTILISATEUR
3  ORDER BY DATE_CREATION DESC;
4
```

Data Output | Messages | Notifications

	arying (100)	password character varying (255)	datedenaissance date	avatar character varying (255)	nom character varying (50)	prenom character varying (50)	role boolean	date_creation timestamp without time zone
1	e@example.com	curie123	1867-11-07	http://example.com/avatar7.jpg	Curie	Marie	true	2024-06-08 10:00:00
2	@example.com	password321	1975-04-04	http://example.com/avatar4.jpg	Brown	Bob	false	2023-01-04 13:00:00
3	@example.com	password789	2000-03-03	http://example.com/avatar3.jpg	Jones	Alice	true	2023-01-03 12:00:00
4	@example.com	password456	1990-02-02	http://example.com/avatar2.jpg	Smith	Jane	false	2023-01-02 11:00:00
5	example.com	newpassword123	1980-01-01	http://example.com/avatar1.jpg	Doe	John	true	2023-01-01 10:00:00

Figure 15: Exemple DESC

10.User Management

```
Query  Query History
1  -- Là on créé les trois rôles
2  CREATE ROLE user_normal;
3  CREATE ROLE administrateur;
4  CREATE ROLE spectateur;
5
6  -- Privilèges de l'user
7  GRANT CONNECT ON DATABASE si40 TO user_normal;
8  GRANT USAGE ON SCHEMA public TO user_normal;
9  GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO user_normal;
10
11 -- Privilèges de l'admin
12 GRANT ALL PRIVILEGES ON DATABASE si40 TO administrateur;
13 GRANT ALL PRIVILEGES ON SCHEMA public TO administrateur;
14 GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO administrateur;
15
16 -- Et privilèges d'un spectateur
17 GRANT CONNECT ON DATABASE si40 TO spectateur;
18 GRANT USAGE ON SCHEMA public TO spectateur;
19 GRANT SELECT ON ALL TABLES IN SCHEMA public TO spectateur;

Data Output  Messages  Notifications
GRANT

Query returned successfully in 34 msec.
```

Figure 16 : Exemple User Management

Pour les parties précédentes, nous avions des rôles par défaut en true et false. Nous avons donc décidés de modifier la colonne rôle dans la table utilisateur pour la mettre de type TEXT, pour qu'elle soit valide pour la suite.

ROLE en SQL va permettre va permettre de créer un rôle qui peut correspondre à un « droit » comme par exemple le droit de lecture ou d'écriture. Ci-dessus, nous créons les rôles user_normal, administrateur, et spectateur

GRANT en SQL va nous permettre d'attribuer les privilèges au rôle que nous venons de créer.

Maintenant on vérifie que ces requêtes d'user management sont bien fonctionnelles :

Verif 1 :

```
Query  Query History
1  -- On vérifie si nos rôles sont bien créés
2  SELECT rolname
3  FROM pg_roles
4  WHERE rolname IN ('user_normal', 'administrateur', 'spectateur');
5

Data Output  Messages  Notifications
```

	rolname name
1	user_normal
2	administrateur
3	spectateur

Figure 17: Verif création rôle

Verif 2 :

```
-- On vérifie les privilèges sur les tables de chaque rôle
SELECT grantee, table_name, privilege_type
FROM information_schema.role_table_grants
WHERE table_schema = 'public'
AND grantee IN ('user_normal', 'administrateur', 'spectateur');
```

	grantee name	table_name name	privilege_type character varying
1	user_normal	utilisateur	INSERT
2	user_normal	utilisateur	SELECT
3	user_normal	utilisateur	UPDATE
4	user_normal	utilisateur	DELETE
5	administrateur	utilisateur	INSERT
6	administrateur	utilisateur	SELECT
7	administrateur	utilisateur	UPDATE
8	administrateur	utilisateur	DELETE
9	administrateur	utilisateur	TRUNCATE
10	administrateur	utilisateur	REFERENCES
11	administrateur	utilisateur	TRIGGER
12	spectateur	utilisateur	SELECT

Figure 18: Verif permissions rôle

On voit donc bien que ces requêtes d'user management sont bien fonctionnelles.

11. Amélioration des procédure et trigger (8bis)

Après l'ajout des rôles, nous avons donc dû modifier la procédure pour ajouter un utilisateur et le trigger de vérification d'email. Cette fois, ils seront fonctionnels pour un rôle qui prend une valeur TEXT et non BOOLEAN comme dans la partie 8.

Nous commençons par recréer un trigger pour vérifier le format de l'email :

```
CREATE OR REPLACE FUNCTION validate_email_format()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.email !~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' THEN
        RAISE EXCEPTION 'Invalid email format: %', NEW.email;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER validate_email_before_insert_or_update
BEFORE INSERT OR UPDATE ON UTILISATEUR
FOR EACH ROW
EXECUTE FUNCTION validate_email_format();
```

```
CREATE TRIGGER

Query returned successfully in 117 msec.
```

Figure 19: Fonction et trigger pour vérifier email format

Puis nous créons la procédure pour ajouter un utilisateur avec cette fois un rôle valide (la procédure vérifiera si le rôle existe également).

Query Query History

```
CREATE OR REPLACE PROCEDURE add_user(
    p_email TEXT,
    p_password TEXT,
    p_datedenaissance DATE,
    p_avatar TEXT,
    p_nom TEXT,
    p_prenom TEXT,
    p_role TEXT,
    p_date_creation DATE
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Vérification du rôle valide
    IF p_role NOT IN ('administrateur', 'user_normal', 'spectateur') THEN
        RAISE EXCEPTION 'Invalid role: %', p_role;
    END IF;

    INSERT INTO UTILISATEUR (
        EMAIL,
        PASSWORD,
        DATEDENAISSANCE,
        AVATAR,
        NOM,
        PRENOM,
        ROLE,
        DATE_CREATION
    ) VALUES (
        p_email,
        p_password,
```

Data Output Messages Notifications

```
CREATE PROCEDURE

Query returned successfully in 75 msec.
```

Figure 20 : Fonction permetant l'ajout d'un utilisateur

Maintenant, vérifions l'implantation du trigger et de la procedure.

Vérification de la validité d'email :

Query Query History

```
CALL add_user(
    'invalid-email-format',  -- Email invalide
    'securepassword',
    '1990-01-01',
    'avatar.png',
    'Doe',
    'John',
    'user_normal',
    CURRENT_DATE
);
```

Data Output Messages Notifications

```
ERROR:  Invalid email format: invalid-email-format
CONTEXT:  PL/pgSQL function validate_email_format() line 4 at RAISE
SQL statement "INSERT INTO UTILISATEUR (
        EMAIL,
        PASSWORD,
        DATEDENAISSANCE,
        AVATAR,
        NOM,
        PRENOM,
        ROLE,
        DATE_CREATION
    ) VALUES (
        p_email,
        p_password,
        p_datedenaissance,
        p_avatar,
        p_nom,
        p_prenom,
```

Figure 21: Appel à add_user avec mauvais mot de passe

On peut voir que l'on a bien une erreur lorsque l'on appelle la procédure avec un email invalide.

Vérification de la procédure pour un rôle invalide :

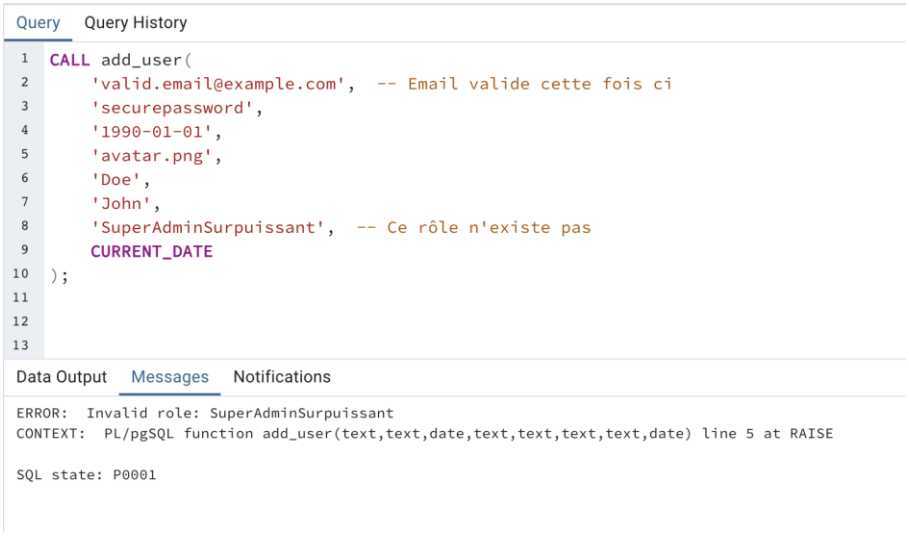


Figure 22: Appel add_user avec mauvais rôle

Vérification de la procédure pour des valeurs valides :

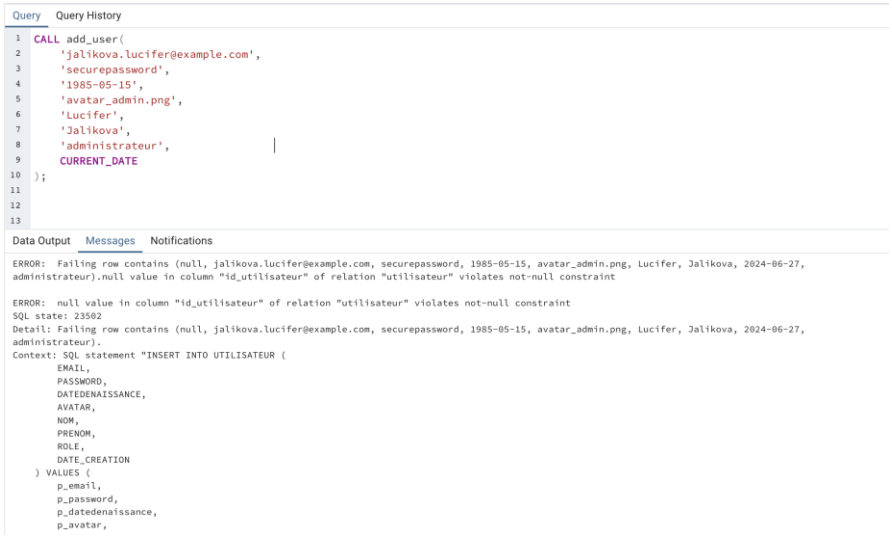


Figure 23: Appel add_user mal géré

La procédure ne marche pas. On va donc utiliser les séquences pour générer automatiquement des valeurs pour ID_UTILISATEUR.

```
Query  Query History
1 CREATE SEQUENCE utilisateur_id_seq;
2 ALTER TABLE UTILISATEUR
3 ALTER COLUMN ID_UTILISATEUR SET DEFAULT nextval('utilisateur_id_seq');
4
5
Data Output  Messages  Notifications
ALTER TABLE

Query returned successfully in 83 msec.
```

Figure 24: Ajout de séquence à utilisateur

On retente maintenant d'appeler notre procédure pour ajouter un utilisateur :

```
Query  Query History
1  CALL add_user(
2      'jalikova.lucifer@example.com',
3      'securepassword',
4      '1985-05-15',
5      'avatar_admin.png',
6      'Lucifer',
7      'Jalikova',
8      'administrateur',
9      CURRENT_DATE
10 );
11
Data Output  Messages  Notifications
CALL

Query returned successfully in 76 msec.
```

Figure 25: Ajout utilisateur Lucifer avec add_user

On regarde si la table à bien été mise à jour :

```
Query  Query History
1 SELECT * FROM utilisateur
Data Output  Messages  Notifications
```

	id_utilisateur [PK] integer	email text	password text	datedenaissance date	avatar text	nom text	prenom text	date_creation date	role text
1	1	jalikova.lucifer@example.com	securepassword	1985-05-15	avatar_admin.png	Lucifer	Jalikova	2024-06-27	administrateur
2	2	winston.winstony@example.com	securepassword	1992-02-28	avatar_winston.png	Winstony	Winston	2024-06-27	user_normal

Figure 26: Affichage table utilisateur

12.Cursor

```
Query  Query History
1  CREATE OR REPLACE PROCEDURE process_utilisateurs()
2  LANGUAGE plpgsql
3  AS $$
4  DECLARE
5      utilisateur_record RECORD;
6      utilisateur_cursor CURSOR FOR
7          SELECT ID_UTILISATEUR, EMAIL, NOM, PRENOM, ROLE FROM UTILISATEUR;
8  BEGIN
9      -- Là on ouvre le cursor
10     OPEN utilisateur_cursor;
11     LOOP
12         -- On recup chaque ligne dans utilisateur record
13         FETCH utilisateur_cursor INTO utilisateur_record;
14         EXIT WHEN NOT FOUND;
15
16         -- Là on affiche pour chaque enregistrement
17         RAISE NOTICE 'ID: %, Email: %, Nom: %, Prénom: %, Rôle: %',
18             utilisateur_record.ID_UTILISATEUR,
19             utilisateur_record.EMAIL,
20             utilisateur_record.NOM,
21             utilisateur_record.PRENOM,
22             utilisateur_record.ROLE;
23     END LOOP;
24     CLOSE utilisateur_cursor;
25 END;
26 $$;

Data Output  Messages  Notifications

CREATE PROCEDURE

Query returned successfully in 104 msec.
```

Figure 27: Utilisation d'un curseur

Cette procédure utilisant un cursor va parcourir tous les utilisateurs de notre table ‘utilisateur’ et imprimer leur détails.

Essayons maintenant de l’appeler:

```
Query  Query History
1  CALL process_utilisateurs();
2
3

Data Output  Messages  Notifications

NOTICE:  ID: 1, Email: jalikova.lucifer@example.com, Nom: Lucifer, Prénom: Jalikova, Rôle: administrateur
NOTICE:  ID: 2, Email: winston.winstony@example.com, Nom: Winstony, Prénom: Winston, Rôle: user_normal
CALL

Query returned successfully in 97 msec.
```

Figure 28: Appel à la fonction utilisant le curseur

Elle est bien fonctionnelle.

13. Gérer les notifications

Dans cette partie, on va implémenter plusieurs fonctions avec trigger pour configurer les notifications (like, commentaire, follow).

Comment par la fonction de notification de like :

Query | Query History

```
CREATE OR REPLACE FUNCTION notify_post_liked()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO NOTIFICATION (ID_UTILISATEUR, TEXTE, TITRE)
    VALUES (
        (SELECT ID_UTILISATEUR FROM POSTE WHERE ID_POSTE = NEW.ID_POSTE),
        'Votre poste vient d être liké par user:' || NEW.ID_UTILISATEUR,
        'Post Liké'
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_post_liked
AFTER INSERT ON AIME
FOR EACH ROW
WHEN (NEW.ID_POSTE IS NOT NULL)
EXECUTE FUNCTION notify_post_liked();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 85 msec.
```

Figure 29: Fonction gestion like notification

Maintenant la fonction de notificaiton de commentaire :

Query | Query History

```
CREATE OR REPLACE FUNCTION notify_comment_posted()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO NOTIFICATION (ID_UTILISATEUR, TEXTE, TITRE)
    VALUES (
        (SELECT ID_UTILISATEUR FROM POSTE WHERE ID_POSTE = NEW.ID_POSTE),
        'Un nouveau commentaire vient dêtre posté sous votre poste par lutilisateur:' || NEW.ID_UTILISATEUR,
        'Nouveau commentaire'
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_comment_posted
AFTER INSERT ON COMMENTAIRE
FOR EACH ROW
EXECUTE FUNCTION notify_comment_posted();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 86 msec.
```

Figure 30: Fonction gestion Comment Notifications

Et enfin la fonction de notification de follow :

Query | Query History

```
1  CREATE OR REPLACE FUNCTION notify_user_followed()
2  RETURNS TRIGGER AS $$
3  BEGIN
4      INSERT INTO NOTIFICATION (ID_UTILISATEUR, TEXTE, TITRE)
5      VALUES (
6          NEW.ID_UTILISATEUR_1,
7          'Vous avez un nouvel abonné:' || NEW.ID_UTILISATEUR,
8          'Nouvel abonné'
9      );
10     RETURN NEW;
11 END;
12 $$ LANGUAGE plpgsql;
13
14 CREATE TRIGGER trigger_user_followed
15 AFTER INSERT ON SUIT
16 FOR EACH ROW
17 EXECUTE FUNCTION notify_user_followed();
18
19
20
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 87 msec.
```

Figure 31: gestion Follow Notifications

Après avoir testé ces nouvelles fonctions pour l'ajout d'un like, commentaire et follow, on voit et que la table notification a bien été mise à jour :

Query | Query History

```
1  SELECT * FROM notification
2
3
4
```

Data Output | Messages | Notifications

	id_notification [PK] integer	id_utilisateur integer	texte text	titre text
1	1	1	Notif initiale	Notif initiale
2	3	1	Votre poste vient d être liké par user:2	Post Liké
3	4	1	Un nouveau commentaire a été posté sur votre poste par user...	Nouveau Commentaire
4	5	1	Vous avez un nouveau follower: user:2	Nouveau Follower

Figure 32: Affichage table Notification

14.Tg_op

On va se servir de TG_OP afin de sauvegarder un historique des actions effectuées sur nos tables.

On commence par créer la table d'historique :

```
Query  Query History
1  CREATE TABLE HISTORIQUE_ACTIONS (
2      ID SERIAL PRIMARY KEY,
3      OPERATION TEXT NOT NULL,
4      TABLE_NAME TEXT NOT NULL,
5      TIMESTAMP TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
6      OLD_DATA JSONB,
7      NEW_DATA JSONB
8  );
9
10
11
12
Data Output  Messages  Notifications
CREATE TABLE

Query returned successfully in 158 msec.
```

Figure 33: Utilisation TG_OP pour historique_action table

On créé la fonction de trigger historique_trigger :

```
Query  Query History
1  CREATE OR REPLACE FUNCTION historique_trigger()
2  RETURNS TRIGGER AS $$
3  BEGIN
4      IF TG_OP = 'INSERT' THEN
5          INSERT INTO HISTORIQUE_ACTIONS (OPERATION, TABLE_NAME, NEW_DATA)
6          VALUES ('INSERT', TG_TABLE_NAME, row_to_json(NEW));
7      ELSIF TG_OP = 'UPDATE' THEN
8          INSERT INTO HISTORIQUE_ACTIONS (OPERATION, TABLE_NAME, OLD_DATA, NEW_DATA)
9          VALUES ('UPDATE', TG_TABLE_NAME, row_to_json(OLD), row_to_json(NEW));
10     ELSIF TG_OP = 'DELETE' THEN
11         INSERT INTO HISTORIQUE_ACTIONS (OPERATION, TABLE_NAME, OLD_DATA)
12         VALUES ('DELETE', TG_TABLE_NAME, row_to_json(OLD));
13     END IF;
14     RETURN NULL;
15 END;
16 $$ LANGUAGE plpgsql;
17
18
19
Data Output  Messages  Notifications
CREATE FUNCTION

Query returned successfully in 83 msec.
```

Figure 34: Fonction historique_trigger

On a également créé des trigger pour toutes les tables à surveiller (utilisateur, poste et commentaire).

Testons maintenant si notre table sauvegarde bien les actions réalisées sur nos autres tables :

Query | Query History

```
INSERT INTO UTILISATEUR (ID_UTILISATEUR, EMAIL, PASSWORD, DATEDENAISSANCE, AVATAR, NOM, PRENOM, ROLE, DATE_CREATION)
VALUES (3, 'keebler@gmail.com', 'securepassword', '1999-01-01', 'avatar.png', 'Keebler', 'Cat', 'user_normal', CURRENT_DATE);


UPDATE POSTE
SET TITRE = 'Titre poste modifié'
WHERE ID_POSTE = 1;

DELETE FROM COMMENTAIRE
WHERE ID_COMMENTAIRE = 1;
```

Data Output | Messages | Notifications

```
DELETE 1

Query returned successfully in 100 msec.
```

Figure 35: Actions pour tester fonction historique_trigger

Query | Query History

```
SELECT * FROM HISTORIQUE_ACTIONS;
```

Data Output | Messages | Notifications

	id [PK] integer	operation text	table_name text	timestamp timestamp without time zone	old_data jsonb
1	1	INSERT	utilisateur	2024-06-27 18:41:49.947401	[null]
2	2	UPDATE	poste	2024-06-27 18:41:49.947401	{"date": "2024-06-27", "titre": "Poste de Jalikova", "contenu": "Ceci est un poste de Jalikova Lucifer.", "id_poste": 1, "visibilite"
3	3	DELETE	commentaire	2024-06-27 18:41:49.947401	{"contenu": "Ceci est un commentaire sous le poste de Jalikova par Winston", "id_poste": 1, "date_creation": "2024-06-27", "

Figure 36: Affichage table Historique_Actions

La table est donc bien fonctionnelle avec tous les trigger.

RESUME DES FONCTIONS ET TRIGGER UTILISABLES

Nom	Type	Description
add_user	Procédure	Ajoute un nouvel utilisateur dans la table `UTILISATEUR` avec un mot de passe haché pour des raisons de sécurité.
verify_user_password	Fonction	Vérifie si le mot de passe fourni correspond au mot de passe haché stocké pour l'utilisateur spécifié.
notify_post_liked	Fonction	Crée une notification lorsqu'un poste est liké par un utilisateur.
notify_comment_posted	Fonction	Crée une notification lorsqu'un commentaire est posté sur un poste.
notify_user_followed	Fonction	Crée une notification lorsqu'un utilisateur en suit un autre.
historique_trigger	Fonction	Journalise les opérations `INSERT`, `UPDATE`, et `DELETE` effectuées sur les tables spécifiées dans la table `HISTORIQUE_ACTIONS`.
trigger_post_liked	Trigger	Déclenche la fonction `notify_post_liked` après une insertion dans la table `AIME`.
trigger_comment_posted	Trigger	Déclenche la fonction `notify_comment_posted` après une insertion dans la table `COMMENTAIRE`.
trigger_user_followed	Trigger	Déclenche la fonction `notify_user_followed` après une insertion dans la table `SUIT`.
historique_utilisateur	Trigger	Déclenche la fonction `historique_trigger` après une insertion, mise à jour ou suppression dans la table `UTILISATEUR`.
historique_poste	Trigger	Déclenche la fonction `historique_trigger` après une insertion, mise à jour ou suppression dans la table `POSTE`.
historique_commentaire	Trigger	Déclenche la fonction `historique_trigger` après une insertion, mise à jour ou suppression dans la table `COMMENTAIRE`.

Figure 37: Resume des fontions et trigger utilisable

ERD DE NOTRE DATABASE

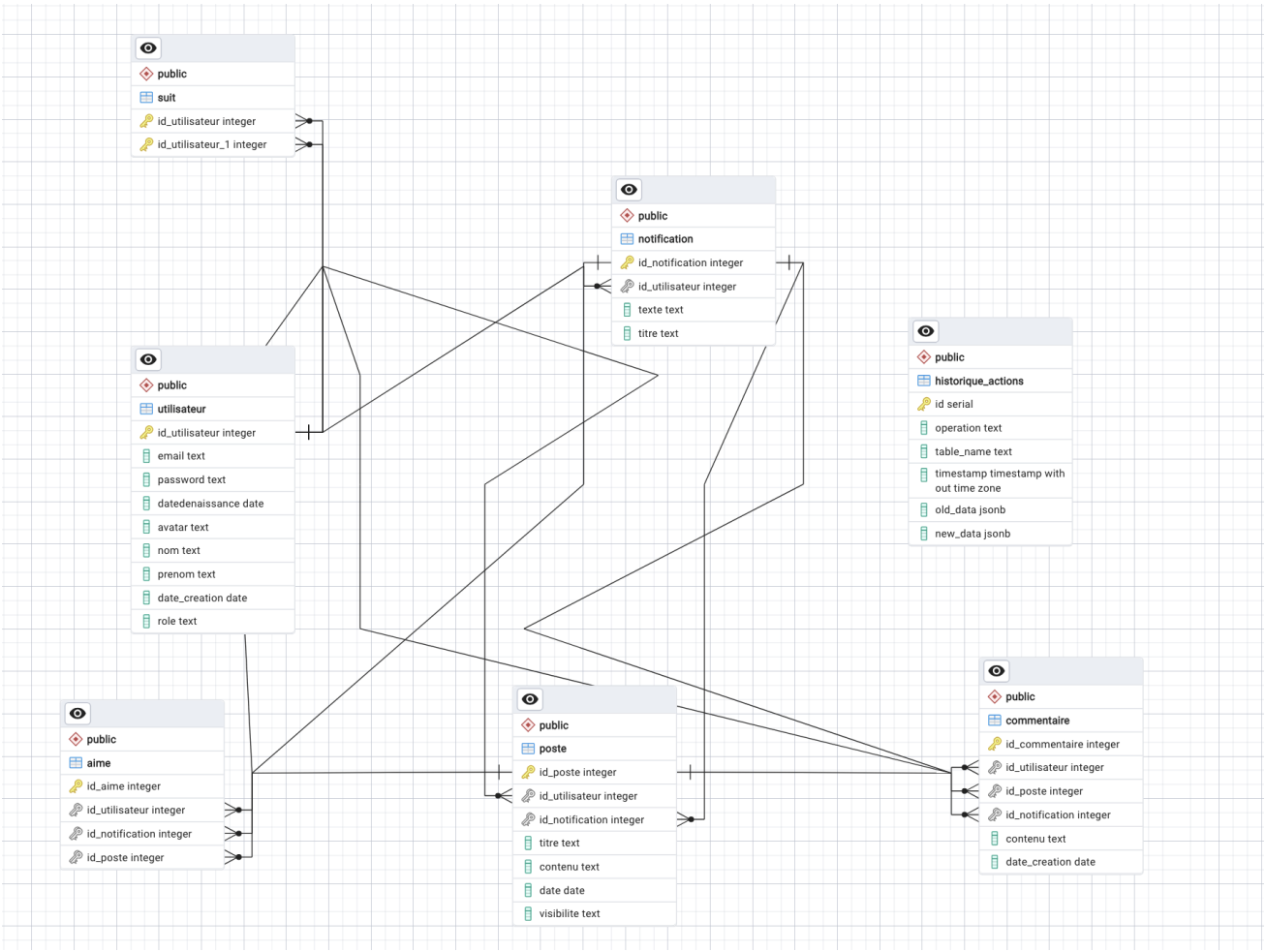


Figure 38: ERD Database

PARTIE OLAP

ANALYSE OLAP AVEC POWER BI

Dans cette section, nous explorons l'analyse OLAP (Online Analytical Processing) réalisée à l'aide de Power BI pour notre projet de réseau social. Cette approche implique la construction d'un modèle de données multidimensionnel robuste, appelé cube OLAP, en basant sur un schéma en étoile, avec une table de faits centrale et des dimensions autour, essentielles pour une analyse approfondie : intégrant diverses dimensions et tables de faits essentielles pour comprendre le comportement des utilisateurs, l'engagement avec le contenu et la performance de la plateforme.

MISE EN PLACE DU CUBE OLAP

Nous avons utilisé les capacités de Power BI pour concevoir et construire méticuleusement un modèle de données multidimensionnel adapté aux besoins spécifiques de notre plateforme de réseau social. Le cube OLAP est centré autour de dimensions clés et de tables de faits essentielles pour une analyse approfondie :

- **Dimension Utilisateur** : Capture des informations détaillées sur le profil des utilisateurs, permettant la segmentation et l'analyse basées sur la démographie, l'activité des utilisateurs et les niveaux d'engagement.
- **Dimension Commentaire** : Fournit des insights sur les interactions des utilisateurs à travers les commentaires, incluant l'analyse de sentiment lorsque cela est applicable.
- **Dimension Notification** : Suit les notifications générées par les actions des utilisateurs, facilitant l'analyse des modèles d'engagement des utilisateurs.
- **Dimension Like** : Enregistre les instances de likes des utilisateurs sur les posts, contribuant à comprendre la popularité du contenu et les préférences des utilisateurs.
- **Dimension Suivi** : Détaille les relations de suivi des utilisateurs, aidant à l'analyse de l'influence des utilisateurs et de la dynamique du réseau.

Tables de Faits

En plus des dimensions, notre cube OLAP inclut les tables de faits suivantes :

- **Table de Faits Post** : Se concentre sur les attributs liés à chaque post, incluant le type de contenu, la date de création et les métriques d'engagement comme les likes et les commentaires. Cette table de faits sert de point central pour analyser la performance des postes et la popularité du contenu.Le schéma en étoile utilisé

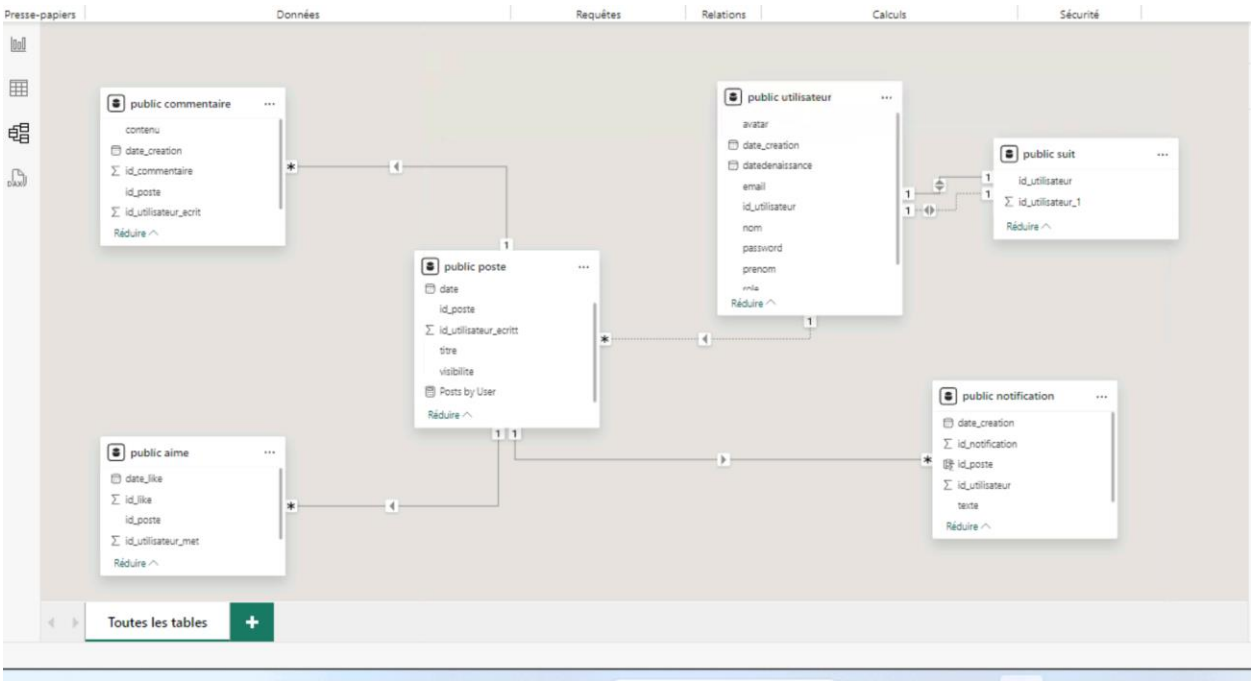


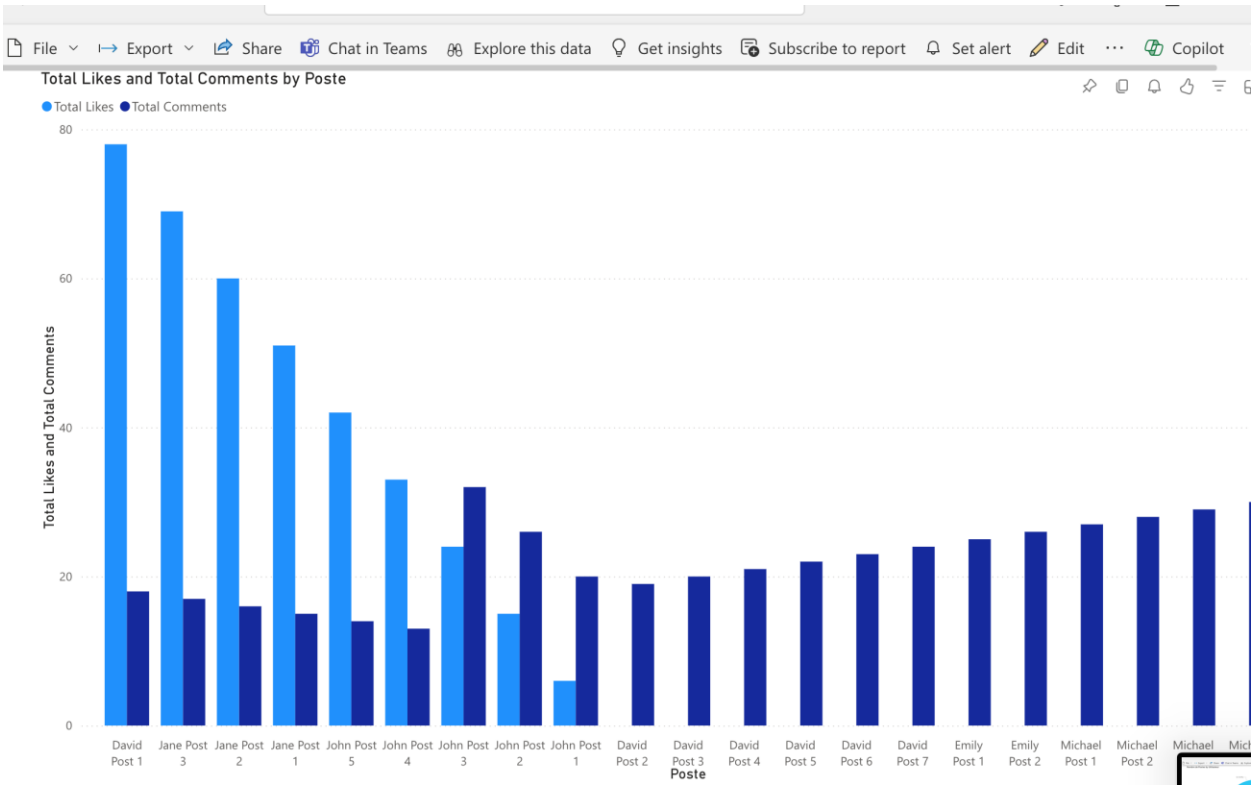
Figure 39: Le schéma en étoile utilisé

VISUALISATIONS ET ANALYSES

Notre cube OLAP permet des visualisations dynamiques et informatives à travers Power BI, soutenant la prise de décision stratégique et les insights opérationnels :

- **Visualisation 1 : Analyse de la Popularité des Postes**

Cette visualisation met en évidence la popularité des différents types de postes basée sur les likes et les commentaires reçus, offrant des insights sur la performance du contenu et les préférences des utilisateurs.

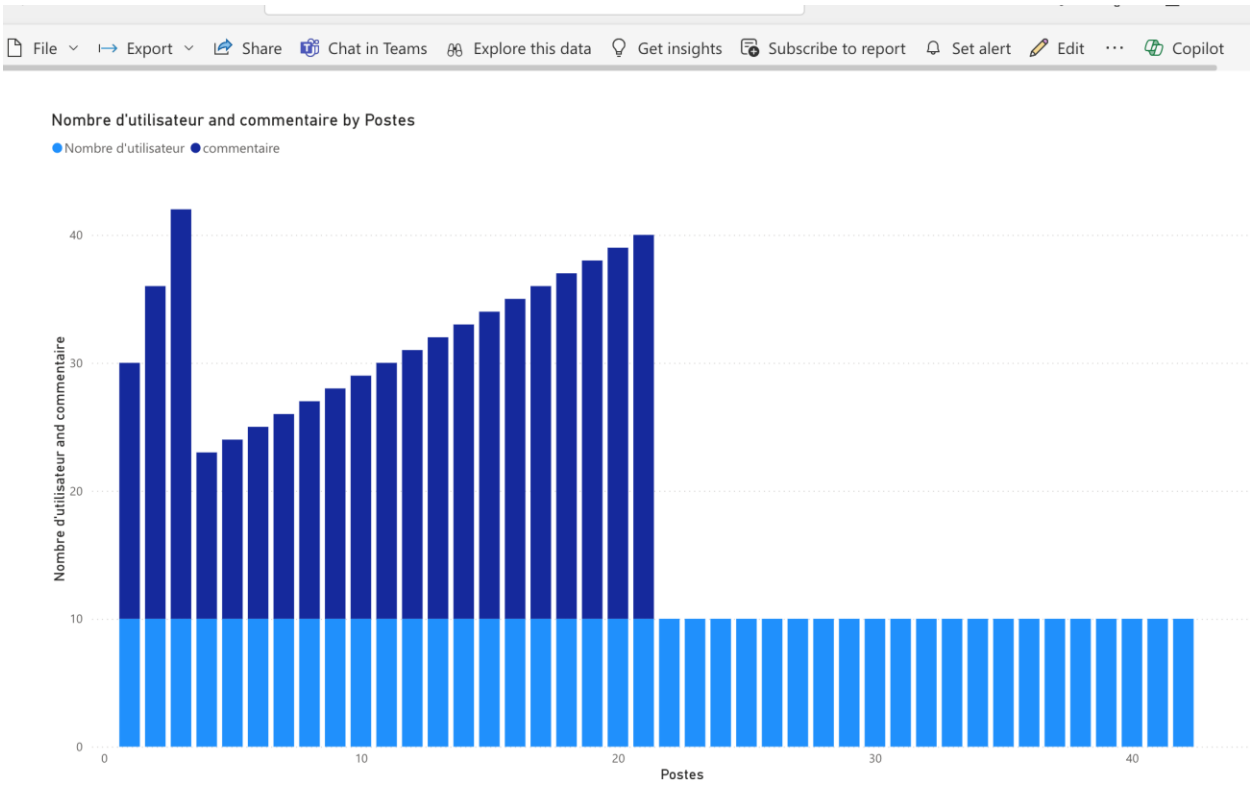


- **Visualisation 2 : Analyse de la Popularité des Postes par Utilisateur**

Cette visualisation analyse la popularité des postes de chaque utilisateur en se basant uniquement sur le nombre de postes publiés. Elle permet de comparer la fréquence et la quantité de contenu partagé par chaque utilisateur, offrant ainsi des insights sur leur activité et leur contribution globale à la plateforme.

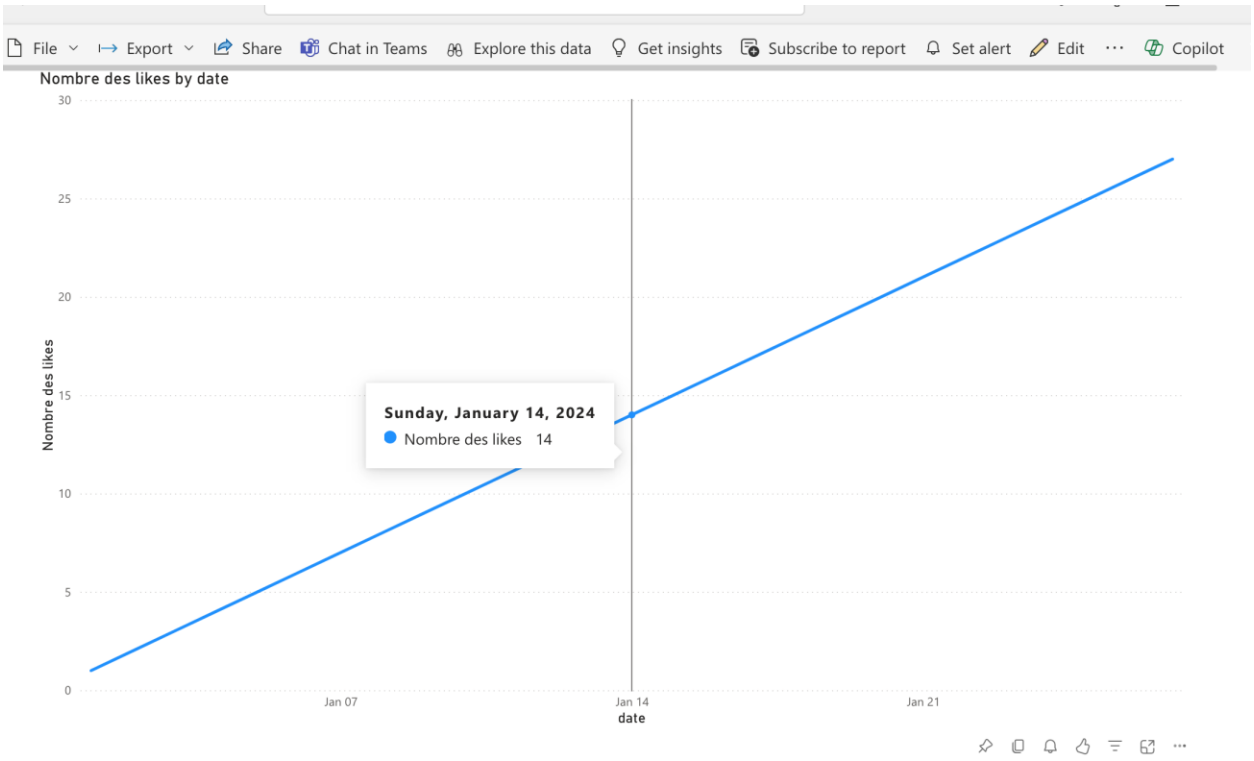
- **Visualisation 3 : Analyse de l'Activité des Utilisateurs par Nombre de Postes et Commentaires**

Cette visualisation examine l'activité des utilisateurs en se concentrant sur deux aspects clés : le nombre de postes publiés par utilisateur et le nombre moyen de commentaires par poste. Elle permet d'évaluer la contribution individuelle des utilisateurs à travers la fréquence de leurs publications et l'engagement généré par ces postes sous forme de commentaires.



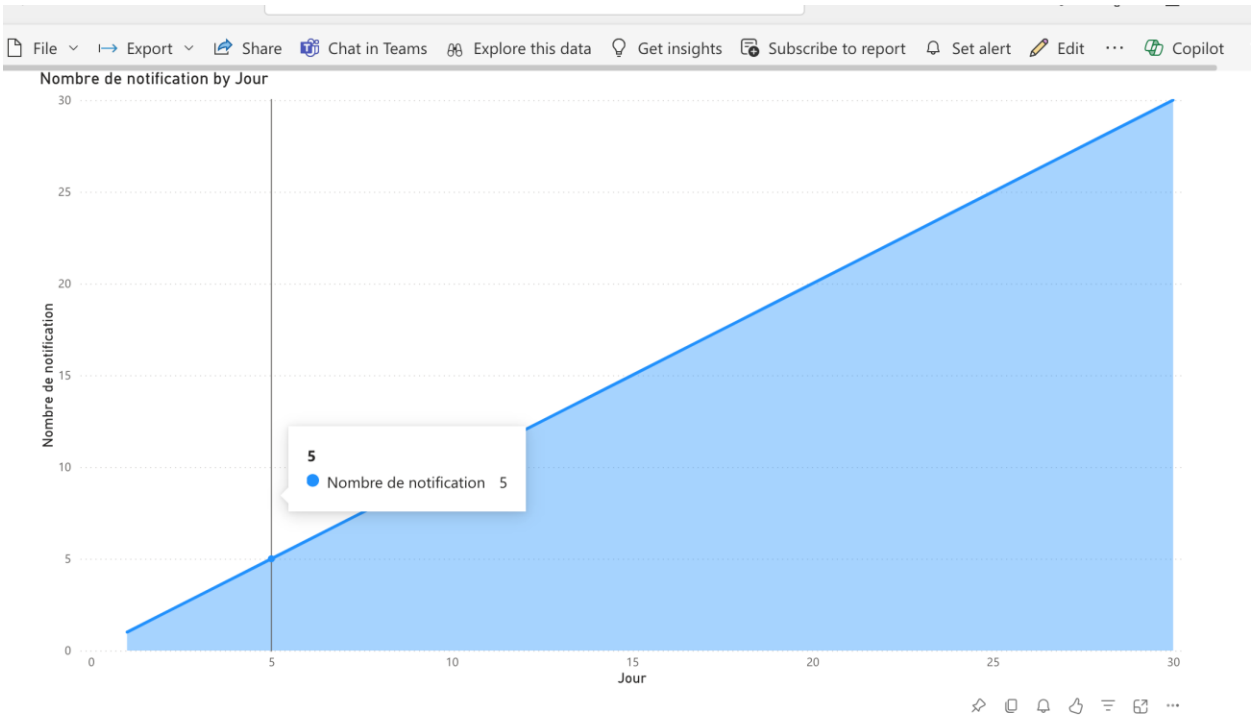
- **Visualisation 4 : Analyse du Nombre de Likes par Date**

Cette visualisation se focalise sur l'évolution du nombre de likes au fil du temps. Elle permet d'observer comment la popularité des postes varie selon les dates de publication, offrant ainsi des insights sur les tendances d'engagement des utilisateurs au fil du temps.



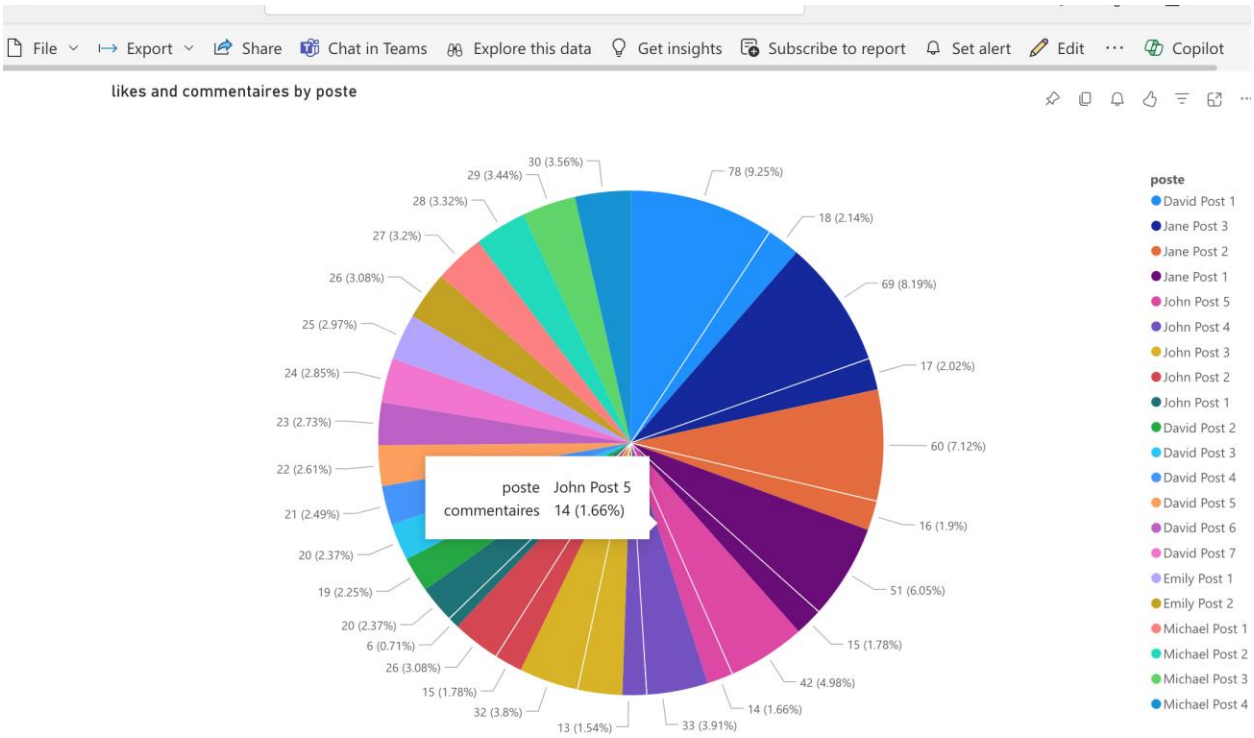
- **Visualisation 5 : Analyse du Nombre de Notifications par Jour**

Cette visualisation analyse le nombre de notifications reçues par jour. Elle permet de visualiser et de comparer l'activité des utilisateurs en fonction de la fréquence à laquelle ils interagissent avec le contenu publié, offrant ainsi des insights sur les moments où l'activité et l'engagement sont les plus élevés sur la plateforme.



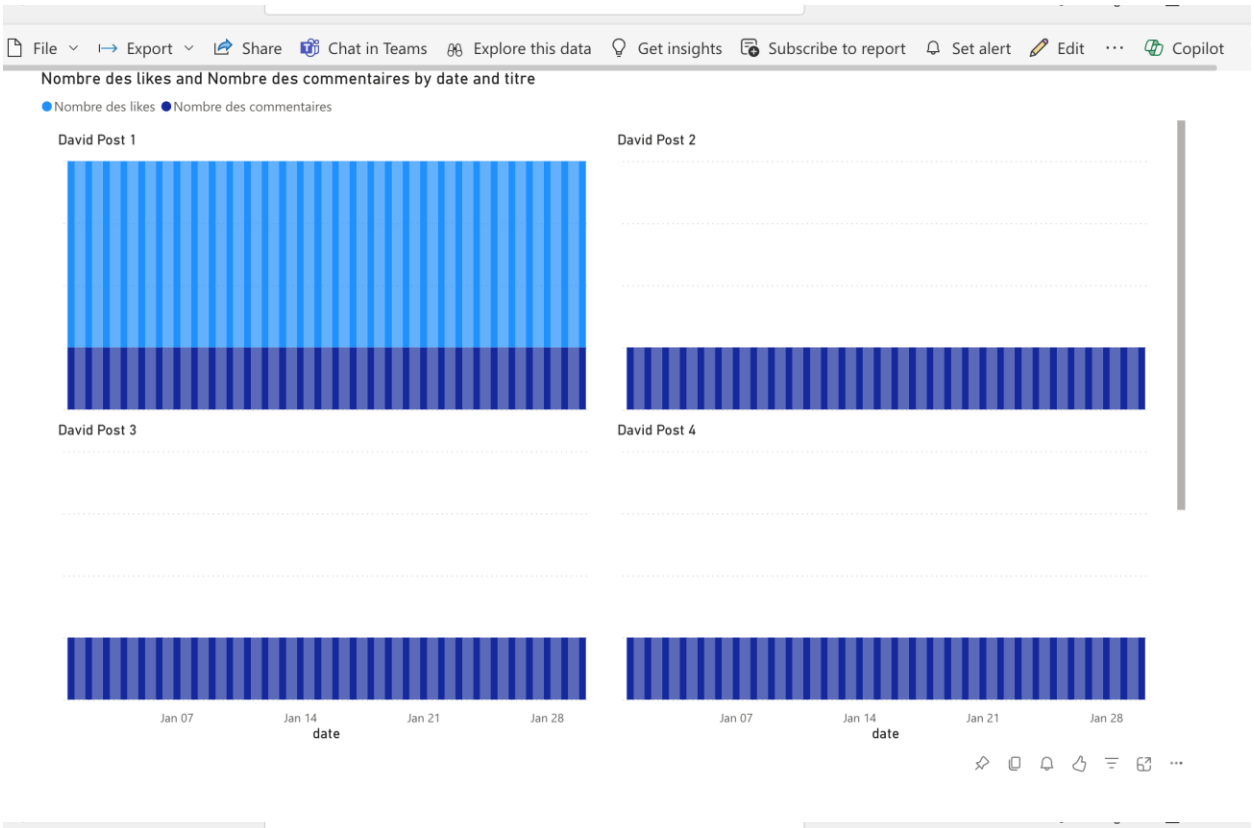
- **Visualisation 6 en Secteurs : Distribution des Likes et des Commentaires par Poste**

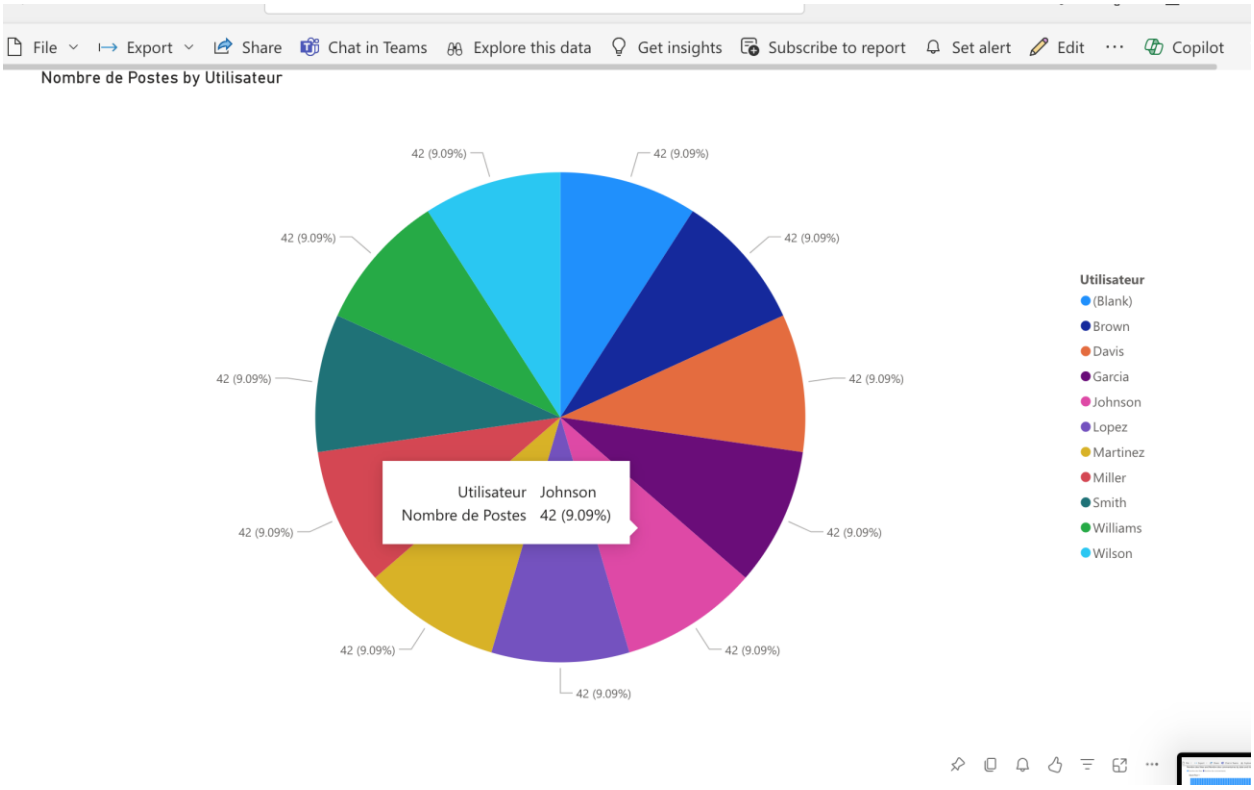
Cette visualisation présente la répartition des interactions générées par les posts en deux catégories principales : les likes et les commentaires. Chaque secteur du graphique représente la proportion relative de ces deux types d'interaction pour chaque poste publié par les utilisateurs. Cela permet de visualiser rapidement quelle proportion de l'engagement provient des likes par rapport aux commentaires, offrant ainsi une vue d'ensemble de l'interaction qualitative et quantitative autour du contenu partagé.



- **Visualisation 7 : Nombre de Postes par Utilisateur**

Cette visualisation montre le nombre de postes publiés par chaque utilisateur. Elle permet de comparer l'activité de publication des utilisateurs, offrant ainsi des insights sur la contribution individuelle à la plateforme en termes de volume de contenu partagé.


File
Export
Share
Chat in Teams
Explore this data
Get insights
Subscribe to report
Set alert
Edit
Copilot
Nombre des likes and Nombre des commentaires by date and titre
Nombre des likes
Nombre des commentaires
David Post 1
David Post 2
David Post 3
David Post 4
Jan 07
Jan 14
Jan 21
Jan 28
date


File
Export
Share
Chat in Teams
Explore this data
Get insights
Subscribe to report
Set alert
Edit
Copilot
Nombre de Postes by Utilisateur
42 (9.09%)
Utilisateur Johnson
Nombre de Postes 42 (9.09%)
Utilisateur
(Blank)
Brown
Davis
Garcia
Johnson
Lopez
Martinez
Miller
Smith
Williams
Wilson

FONCTIONNALITES POWER BI UTILISEES

Pour dériver des insights exploitables, nous avons utilisé des fonctionnalités avancées de Power BI, y compris :

- **Mesures DAX et Colonnes Calculées** : Implémentées pour calculer des métriques clés telles que les likes totaux, le nombre moyen de commentaires par post et le taux d'engagement des utilisateurs, offrant une meilleure compréhension des interactions des utilisateurs et de la performance du contenu.
- **Filtres et Tranches de Données** : Permettent une exploration dynamique des données en appliquant des filtres basés sur les périodes de temps, les segments d'utilisateurs et les régions géographiques, assurant que les parties prenantes puissent se concentrer sur des aspects spécifiques d'intérêt.

INSIGHTS ET SUPPORT A LA DECISION

Les capacités OLAP de Power BI ont considérablement amélioré nos capacités analytiques.

- **Prise de Décision Stratégique** : En fournissant aux cadres et aux parties prenantes des insights exploitables issus de relations de données complexes, facilitant la planification stratégique informée.
- **Analyse Opérationnelle** : Permettant aux équipes opérationnelles de surveiller étroitement les métriques de performance et d'identifier des opportunités pour améliorer l'expérience utilisateur et la fonctionnalité de la plateforme.
- **Prévision et Planification** : Utilisant les tendances historiques des données pour prévoir le comportement futur des utilisateurs et la performance de la plateforme, guidant l'allocation des ressources et les stratégies de croissance.

BILAN OLAP

En conclusion, l'intégration de l'analyse OLAP à travers Power BI a été cruciale pour extraire des insights significatifs de nos données de réseau social. Elle a approfondi notre compréhension du comportement des utilisateurs, de la dynamique de l'engagement avec le contenu et de la performance de la plateforme, permettant ainsi une prise de décision éclairée, une planification stratégique et une amélioration continue.

CONCLUSION

En conclusion, nous avons pu réaliser une grande partie de ce que nous voulions faire pour ce projet. Nous pourrions cependant améliorer de nombreux points comme… .

Avant de commencer ce projet, nous n'étions tous les 3 pas très familiers avec les bases de données. Cependant, au fur et à mesure de l'avancée du projet, nous avons pu nous familiariser avec l'environnement des bases de données ainsi que des logiciels associés. Nous avons acquis de nombreuses compétences qui nous serons sans aucun doutes très utiles pour notre parcours futur.

Concernant notre orientation, ce projet a permis d'éclaircir l'idée que nous nous faisions des bases de données, nous dirigeant pour certains d'entre nous (Cyprien et Victor) vers un stage ST40 orienté base de donnée.

Ainsi, ce projet restera pour nous une expérience très positive et riche en apprentissage.

ANNEXES

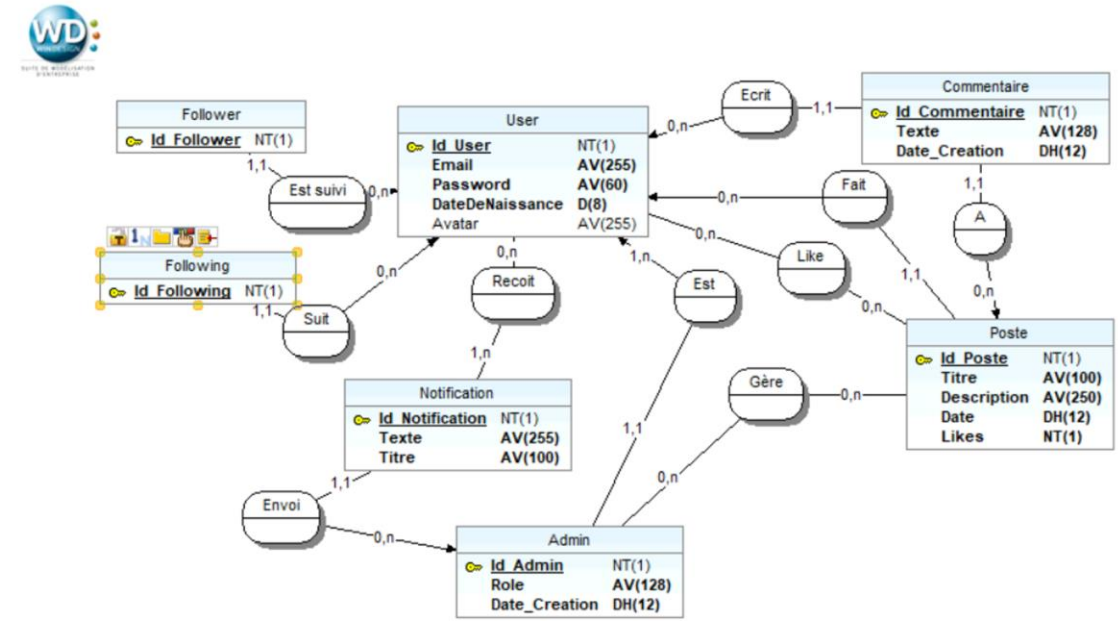


Figure 40: Annexe Première Version du diagramme MCD (mauvaise)

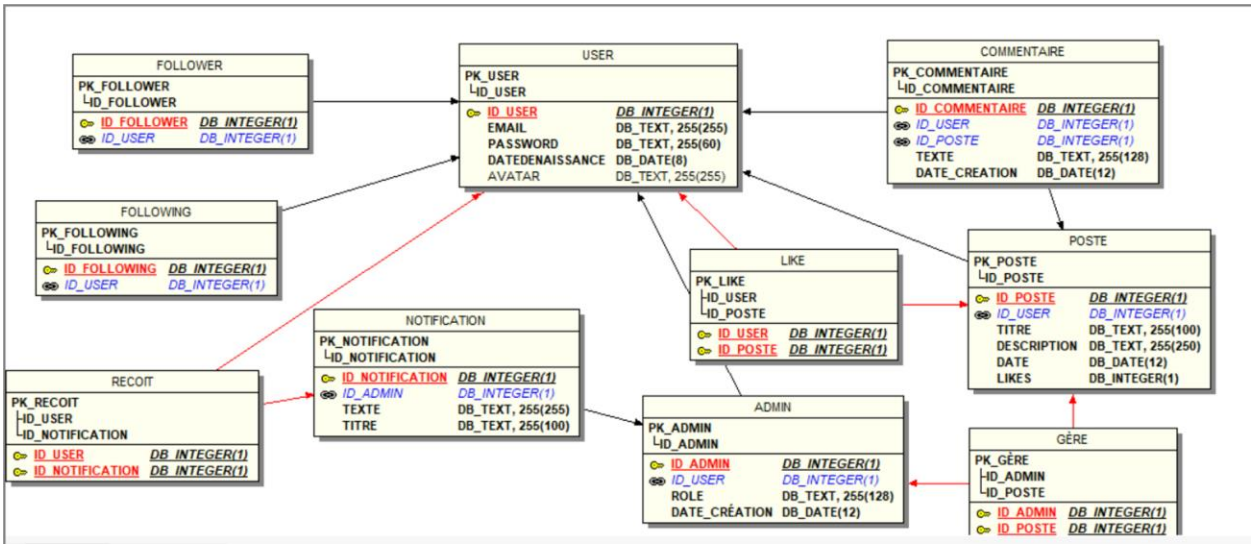


Figure 41: Annexe Première Version du diagramme MLD (mauvaise)