SAE 2.01 : Rapport :

Projet Base de données bancaire

Réalisé par :

Moustapha Ndiaye

Enseignant :

Kevin Atighehchi

Année universitaire 2024/2025

1. Introduction

La banque de Clermont a décidé de moderniser la gestion de ses données, jusqu'ici enregistrées dans un simple fichier CSV. Ce format n'étant plus adapté à une utilisation fiable ou à une augmentation du volume de données, la banque souhaite mettre en place une base de données relationnelle plus solide.
L'objectif est de construire un système sous PostgreSQL capable de :

- Gérer les informations clients et leur lien avec un agent référent,
- Représenter les comptes bancaires, y compris ceux partagés par plusieurs clients,
- Enregistrer tous les mouvements financiers effectués sur les comptes,
- Suivre les parrainages entre clients.

Ce projet vise donc à structurer les données de manière durable, sécurisée et facilement exploitable.

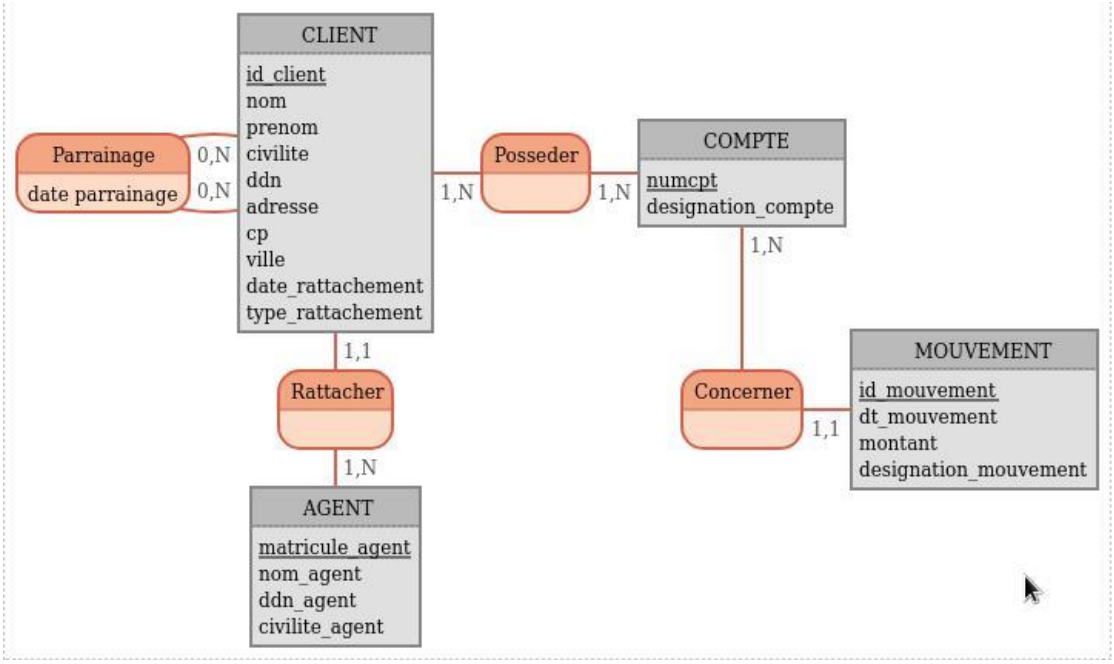
2. Modèle Conceptuel de Données (MCD)

Le modèle conceptuel de données permet de représenter les différentes entités de l'application ainsi que leurs relations. Il a été réalisé à l'aide de l'outil Mocodo Online.
Les entités retenues sont les suivantes :

- CLIENT : id_client, nom, prenom, civilite, ddn, adresse, cp, ville, date_rattachement, type_rattachement.
- AGENT matricule_agent, nom_agent, ddn_agent, civilite_agent.
- COMPTE : numcpt, designation_compte.
- MOUVEMENT : id_mouvement, dt_mouvement, montant, designation_mouvement.

Les associations définies sont les suivantes :

Rattacher , Posséder, Concerner, Parrainage



3. Modèle Logique de Données (MLD)

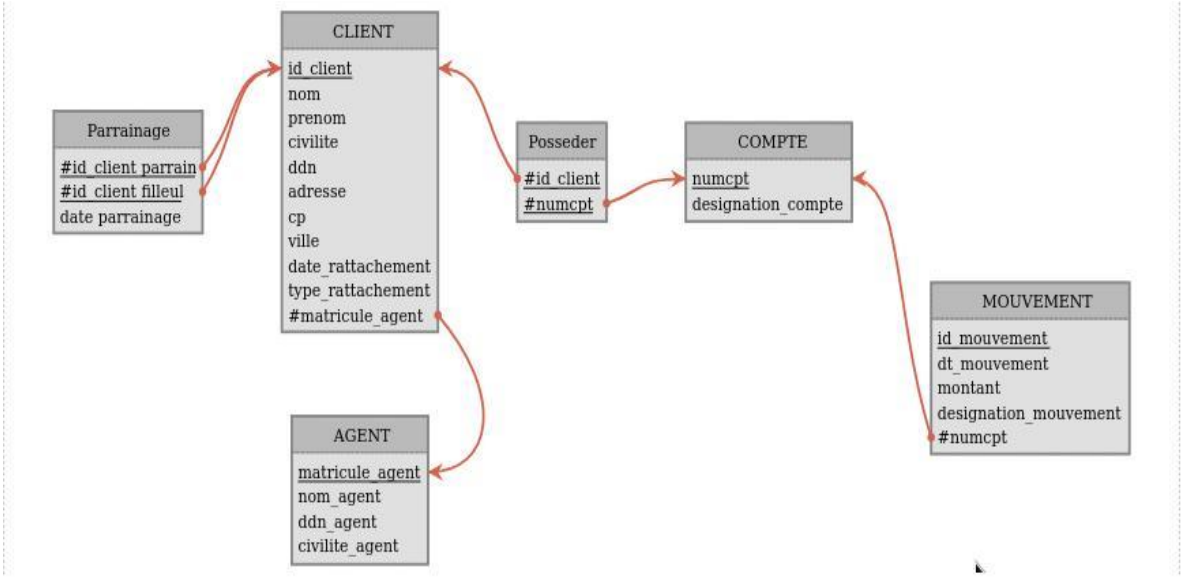
Après le MCD, le MLD traduit chaque entité en table avec ses attributs (noms conservés :

%%mocodo
:

Parrainage: #id_client parrain > CLIENT > id_client, _#id_client filleul > CLIENT > id_client, date parrainage
:
CLIENT: id_client, nom, prenom, civilite, ddn, adresse, cp, ville, date_rattachement, type_rattachement, #matricule_agent > AGENT > matricule_agent
:
Posseder: #id_client > CLIENT > id_client, _#numcpt > COMPTE > numcpt
:
COMPTE: numcpt, designation_compte
:::

:::::::::
MOUVEMENT: id_mouvement, dt_mouvement, montant, designation_mouvement, #numcpt > COMPTE > numcpt
:

:::
AGENT: matricule_agent, nom_agent, ddn_agent, civilite_agent
:::::::



4. Nettoyage des données

Avant de pouvoir insérer les données dans la base, il a fallu nettoyer le fichier CSV, car il contenait des erreurs, surtout dans les colonnes avec des dates.Problèmes rencontrés :

Certaines dates étaient invalides, comme le 29 février 2017 (2017 n'étant pas une année bissextile).

Certains champs étaient erronés ou incomplets, tels que 31/04 ou 00/00/0000.

Le format des dates manquait d'uniformité, avec des écritures différentes comme 2024-06-16 (AAAA-MM-JJ) et 16/06/2024 (JJ/MM/AAAA).

Solution mise en place :
Un script Python a été créé pour nettoyer tout ça.
Il utilise la bibliothèque pandas et gère les erreurs pour repérer les dates incorrectes, corriger ou ignorer celles qui posent problème ou encore convertir toutes les dates dans le même format : JJ/MM/AAAA.

5. Script SQL, Insertion des données et Vues

A) Mise en place de la structure de base de la base de données

Avant d'insérer les données, on a d'abord construit la structure de la base de données à l'aide de scripts SQL pour PostgreSQL.
Les tables ont été créées dans le bon ordre, en respectant les liens entre elles (clés étrangères, dépendances hiérarchiques).
Voici les principales tables créées :

```
CREATE TABLE AGENT (
   matricule_agent INT PRIMARY KEY,
   nom_agent TEXT NOT NULL,
   ddn_agent DATE NOT NULL,
   civilite_agent CHAR(1) NOT NULL CHECK (civilite_agent IN ('M', 'F'))
);

CREATE TABLE CLIENT (
   id_client INT PRIMARY KEY,
   nom TEXT NOT NULL,
   prenom TEXT NOT NULL,
   civilite CHAR(1) NOT NULL CHECK (civilite IN ('M', 'F')),
   ddn DATE NOT NULL,
   adresse TEXT NOT NULL,
   cp INT CHECK (cp BETWEEN 1000 AND 99999),
   ville TEXT NOT NULL,
   date_rattachement DATE,
   type_rattachement TEXT,
   matricule_agent INT REFERENCES AGENT(matricule_agent) ON DELETE SET NULL
);

CREATE TABLE COMPTE (
   numcpt BIGINT PRIMARY KEY,
   designation_compte TEXT NOT NULL
);

CREATE TABLE COMPTE_CLIENT (
   id_client INT REFERENCES CLIENT(id_client) ON DELETE CASCADE,
   numcpt BIGINT REFERENCES COMPTE(numcpt) ON DELETE CASCADE,
   PRIMARY KEY (id_client, numcpt)
);

CREATE TABLE MOUVEMENT (
   id_mouvement INT PRIMARY KEY,
   dt_mouvement DATE NOT NULL,
   montant FLOAT NOT NULL,
   designation_mouvement TEXT,
   numcpt BIGINT REFERENCES COMPTE(numcpt) ON DELETE CASCADE
);

CREATE TABLE PARRAINAGE (
```

```
    id_client_parrain INT REFERENCES CLIENT(id_client) ON DELETE CASCADE,
    id_client_filleul INT REFERENCES CLIENT(id_client) ON DELETE CASCADE,
    date_parrainage DATE NOT NULL,
    PRIMARY KEY (id_client_parrain, id_client_filleul),
    CHECK (id_client_parrain <> id_client_filleul)
);

CREATE TABLE TABLE_TEMPORAIRE (
    id_client INT,
    nom TEXT,
    prenom TEXT,
    civilite CHAR(1),
    ddn DATE,
    adresse TEXT,
    cp INT,
    ville TEXT,
    date_rattachement DATE,
    type_rattachement TEXT,
    numcpt BIGINT,
    designation_compte TEXT,
    id_mouvement INT,
    dt_mouvement DATE,
    montant FLOAT,
    designation_mouvement TEXT,
    parrain TEXT,
    date_parrainage DATE,
    matricule_agent INT,
    nom_agent TEXT,
    ddn_agent DATE,
    civilite_agent CHAR(1)
);
```

REMARQUE :

Certains numéros de compte dépassent la limite d'un INT, donc on utilise BIGINT pour éviter les erreurs.

B) Importation des données brutes

La table temporaire, appelée TABLE_TEMPORAIRE, contient toutes les données nettoyées du fichier CSV.
On utilise la commande \copy pour importer le fichier sae_donnees_bancaires_nettoyee.csv dans cette table, avec les données séparées par
des virgules et la première ligne comme en-tête.
Cette étape sert de zone tampon sans contraintes, ce qui facilite et sécurise le chargement initial des données sans risquer d'erreurs ou de blocages.

C) Répartition des données dans les tables finales

```
-- Agents uniques
INSERT INTO AGENT (matricule_agent, nom_agent, ddn_agent, civilite_agent)
SELECT DISTINCT matricule_agent, nom_agent, ddn_agent, civilite_agent
FROM TABLE_TEMPORAIRE;

-- Clients sans doublon
INSERT INTO CLIENT (id_client, nom, prenom, civilite, ddn, adresse, cp, ville, date_rattachement,
```

```
type_rattachement, matricule_agent)
SELECT DISTINCT ON (id_client) id_client, nom, prenom, civilite, ddn, adresse, cp, ville,
date_rattachement, type_rattachement, matricule_agent
FROM TABLE_TEMPORAIRE;

-- Comptes distincts
INSERT INTO COMPTE (numcpt, designation_compte)
SELECT DISTINCT numcpt, designation_compte
FROM TABLE_TEMPORAIRE;

-- Liaisons compte-client
INSERT INTO COMPTE_CLIENT (id_client, numcpt)
SELECT DISTINCT id_client, numcpt
FROM TABLE_TEMPORAIRE;

-- Mouvements valides
INSERT INTO MOUVEMENT (id_mouvement, dt_mouvement, montant, designation_mouvement,
numcpt)
SELECT DISTINCT ON (id_mouvement) id_mouvement, dt_mouvement, montant,
designation_mouvement, numcpt
FROM TABLE_TEMPORAIRE
WHERE id_mouvement IS NOT NULL;

-- Parrainages
INSERT INTO PARRAINAGE (id_client_parrain, id_client_filleul, date_parrainage)
SELECT
   parrain.id_client AS id_client_parrain,
   filleul.id_client AS id_client_filleul,
   tmp.date_parrainage
FROM TABLE_TEMPORAIRE tmp
JOIN CLIENT parrain ON LOWER(parrain.nom) = LOWER(tmp.parrain)
WHERE tmp.parrain IS NOT NULL;
```

Chaque insertion active les contraintes d'intégrité définies dans le schéma (PRIMARY KEY, FOREIGN KEY, CHECK), assurant ainsi que les données importées restent cohérentes et valides.

1. comptes_dormants : extrait les comptes sans mouvement depuis le début de l'année, et exporte les résultats vers un fichier CSV.
   \copy (SELECT * FROM comptes_dormants) TO 'comptes_dormants.csv' WITH CSV HEADER;

2. etat_parrainages : récupère les parrainages enregistrés depuis le 1er janvier, et les enregistre dans un fichier CSV

   \copy (SELECT * FROM etat_parrainages) TO 'etat_parrainages.csv' WITH CSV HEADER;

Conclusion

Cette démarche a permis de mener l'ensemble du processus, de la modélisation à la normalisation (3FN), jusqu'au scripting SQL.
Les vues dynamiques offrent aux décideurs un accès direct aux données à jour, tout en restant facilement exportables en CSV.
Les difficultés rencontrées (valeurs numériques, relations plusieurs-à-plusieurs, dates incorrectes) ont été traitées de manière rigoureuse à l'aide de PostgreSQL et Python.
Le livrable final fournit ainsi une base de données bancaire fiable, évolutive, et conforme aux attentes du cahier des charges de la SAÉ.