République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle et Sciences des Données

Rapport du projet BDA SQL3-Oracle et NoSQL (MongoDB)

MODULE : Base de données avancé

Réalisé par :

TAOURIRT Hamza

SAADA Samir

Année Universitaire : 2023/2024

Tableau des matières

Partie I : Relationnel-objet

Partie I : NoSQL – Modèle orienté « documents »

Table des figures

Partie I : Relationnel-Objet

1. Introduction

Dans le cadre de ce rapport, nous abordons une étude approfondie sur la modélisation et la manipulation des données, en mettant en œuvre deux paradigmes de gestion de bases de données : relationnel-objet et NoSQL orienté documents. Notre objectif principal est de concevoir et de mettre en œuvre une base de données répondant à des exigences spécifiques, en utilisant ces deux approches distinctes et en évaluant leurs avantages et inconvénients respectifs.

La première partie de ce travail se concentre sur le modèle relationnel-objet. Nous commençons par modéliser une base de données à l'aide d'un schéma relationnel, que nous transformons ensuite en un schéma objet représenté sous forme de diagramme de classes UML. Ensuite, nous décrivons le processus de création des tablespaces et des utilisateurs, ainsi que la définition des données et des associations nécessaires. Nous terminons cette partie en peuplant la base de données avec des instances et en répondant à des requêtes spécifiques à l'aide du langage SQL.

La deuxième partie explore le modèle NoSQL orienté documents. Nous proposons une modélisation de la base de données décrite dans la première partie sous forme de documents, en mettant en évidence les avantages de cette approche pour les requêtes spécifiques sur les prêts. Nous décrivons également le processus de remplissage de la base de données et répondons à des requêtes en utilisant le langage de requête spécifique au NoSQL.

Enfin, nous analysons en détail les requêtes posées pour chaque modèle et comparons les performances ainsi que la flexibilité des deux approches. Nous discutons des avantages et des inconvénients de chaque modèle en fonction des exigences de notre application et formulons des recommandations pour le choix de modèle dans des contextes similaires.

2. Modélisation orientée objet

2.1.Schéma UML

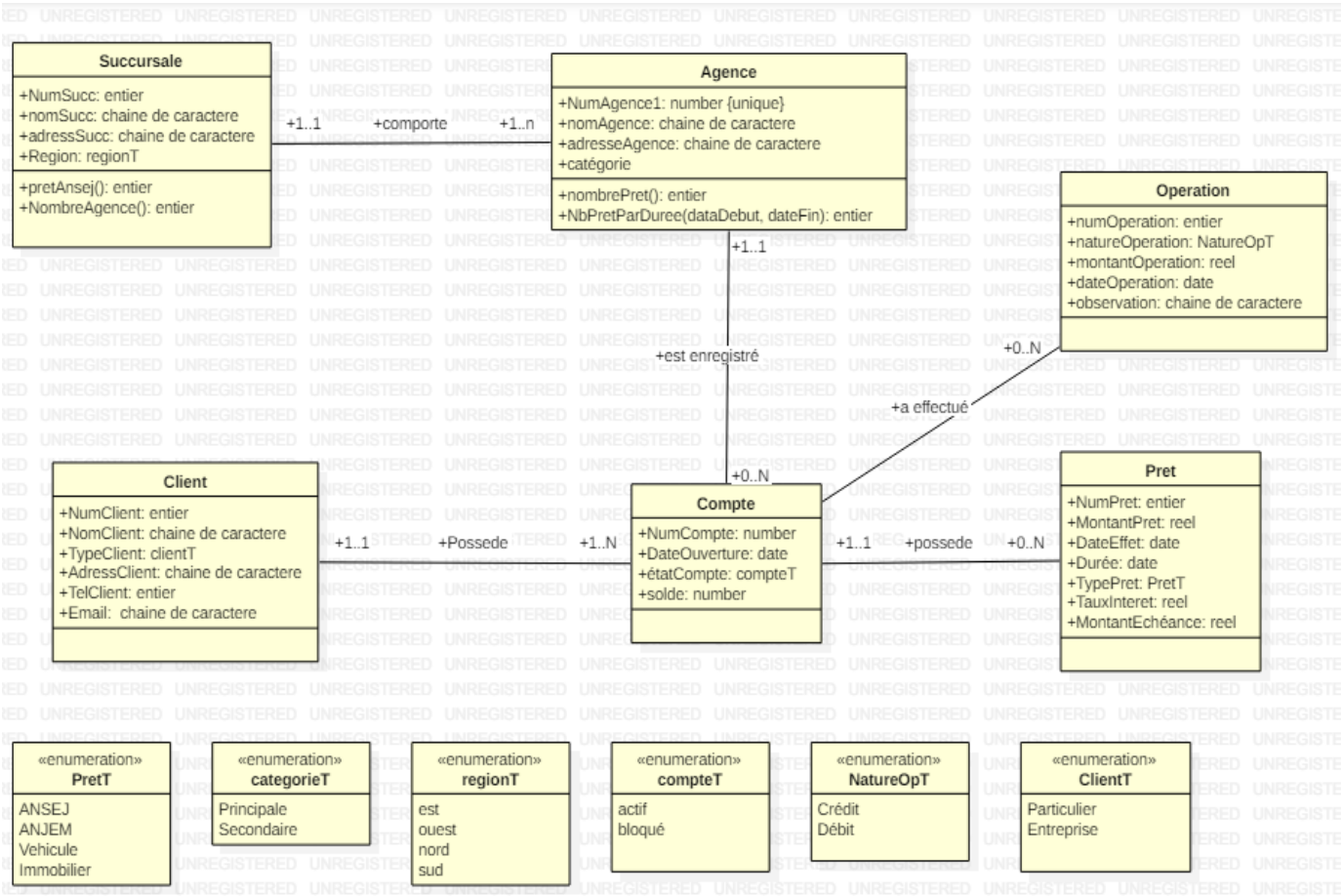


Figure 1: Modèle UML d'un diagramme de classe pour la base de données banque

Dans cette modélisation UML, Nous avons crée la classe Succursale qui possède quatre attributs : NumSucc, nomSucc, adresseSucc, région de plus il possede au minimum deux méthodes. Succursale comporte plusieurs agences et sont lié par un lien d'association, une agence n'appartient qu'a une seul succursale. Les attributs de succursale sont NumAgence, nomAgence, adresseAgence, catégorie. Plusieurs comptes sonts enregistré sur une et une seul agence c'est-à-dire un compte n'est register que sur une agence. Les attributs de la classe compte sont Numcompte, dateOuverture, étatCompte, Solde. Un compte possede de zero à plusieurs opérations de la classe opération et aussi de zero à plusieurs prêts de la classe prêt. La classe client a les attributs : NumOpération, NatureOp, montantOp, DateOp, Observation. La classe prêt possede les attributs NumPrêt, montantPrêt, dateEffet, durée, typePrêt, tauxIntérêt,

montantEchéance. Enfin un client représenté par la classe client avec les attributs NomClient, TypeClient, AdresseClient, NumTel.

2.2. Création des TableSpaces et utilisateur

Les requêtes ci-dessous (figure 2) sont des instructions SQL permettant de créer deux tablespaces dans une base de données Oracle.

La première requête crée un tablespace nommé SQL3_TBS avec un fichier de données initial de taille 100 Mo, pouvant s'étendre automatiquement de 10 Mo à chaque extension, sans limite maximale de taille.

La seconde requête crée un tablespace temporaire nommé SQL3_TempTBS avec un fichier temporaire de taille initiale de 50 Mo, également extensible automatiquement de 10 Mo à chaque extension, sans limite maximale de taille.

```
1
2  CREATE TABLESPACE SQL3_TBS
3    DATAFILE 'sql3_tbs_datafile.dbf' SIZE 100M
4    AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITES;
5
6  CREATE TEMPORARY TABLESPACE SQL3_TempTBS
7    TEMPFILE 'sql3_temptbs_tempfile.dbf' SIZE 50M
8    AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
9    EXTEND MANAGEMENT LOCAL UNIFORM SIZE 1M;
10
11 CREATE USER SQL3 IDENTIFIED BY PASSWORD
12   DEFAULT TABLESPACE SQL3_TBS
13   TEMPORARY TABLESPACE SQL3_TempTBS;
14
15 GRANT ALL PRIVILEDGES TO SQL3;
```

Figure 2:Creation des tables espaces, de l'utilisateur SQL3

2.3. Créer un utilisateur SQL3 en lui attribuant les deux tableSpaces créés précédemment

La requête dans la ligne 11 de figure 2 crée un utilisateur nommé SQL3 dans une base de données Oracle. Cette commande spécifie le tablespace par défaut SQL3_TBS pour le stockage des données de l'utilisateur, ainsi que le tablespace temporaire SQL3_TempTBS pour les opérations temporaires. L'utilisateur est identifié par un mot de passe.

Et les exécution des commandes dans la figure 2 sont illutré dans les figure 3 et 4

```
SQL> CREATE TABLESPACE SQL3_TBS
  2    DATAFILE 'sql3_tbs_datafile.dbf' SIZE 100M
  3    AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED;

Tablespace created.

SQL> CREATE TEMPORARY TABLESPACE SQL3_TempTBS
  2    TEMPFILE 'sql3_temptbs_tempfile.dbf' SIZE 50M
  3    AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
  4    EXTENT MANAGEMENT LOCAL UNIFORM SIZE 1M;

Tablespace created.
```

Figure 3: Creation des tablesSpaces

2.4. Donner tous les privilèges à cet utilisateur.

On crée ll'utilisateur en lui assignants les tablespaces crées précedemment puis on lui donne tout les prévilèges et enfin on se connect à l'utilisateur (figure 5)

```
SQL> CREATE USER SQL3 IDENTIFIED BY password
  2    DEFAULT TABLESPACE SQL3_TBS
  3    TEMPORARY TABLESPACE SQL3_TempTBS;

User created.

SQL> GRANT ALL PRIVILEGES TO SQL3;

Grant succeeded.
```

Figure 4: Creation de l'utilisateur SQL3

```
SQL> connect sql3@orclpdb/password;
Connected.
SQL>
```

Figure 5: Connection à l'utilisateur SQL3

3. Langage de définition de données

3.1. Définition des types et des associations

```
CREATE TYPE SuccursaleType;
/

CREATE TYPE AgenceType;
/

CREATE TYPE ClientType;
/

CREATE TYPE CompteType;
/

CREATE TYPE OperationType;
/

CREATE TYPE PretType;
/
```

Figure 6: Creation des signatures des types principaux

Les commandes dans la figure 6 créent plusieurs types de données utilisateur dans une base de données Oracle, notamment les types Succursale, Agence, Client, Compte, Opération et Prêt, utilisés pour modéliser les différentes entités de l'application bancaire. Leur exécution est montrée dans la figure 7

```
SQL> CREATE TYPE SuccursaleType;
  2  /

Type created.

SQL>
SQL> CREATE TYPE AgenceType;
  2  /

Type created.

SQL>
SQL> CREATE TYPE ClientType;
  2  /

Type created.

SQL>
SQL> CREATE TYPE CompteType;
  2  /

Type created.

SQL>
SQL> CREATE TYPE OperationType;
  2  /

Type created.

SQL>
SQL> CREATE TYPE PretType;
  2  /

Type created.
```

Figure 7: Declaration des types

Les requêtes Dans la figure 8 sont des types de données tableaux faisant référence aux types crées précédemment (figure6), respectivement pour les références aux types Compte, Prêt, Opération et Agence, permettant de stocker des ensembles de références vers ces types spécifiques.

Elles représentent les associations de type 0…N du modèle UML

```
1
2 create type t_set_ref_Compte as table of ref CompteType;
3 /
4 create type t_set_ref_pret as table of ref PretType;
5 /
6 create type t_set_ref_Operation as table of ref OperationType;
7 /
8 create type t_set_ref_Agence as table of ref AgenceType;
9 /
```

Figure 8: Création des tables de références

```
SQL> create type t_set_ref_Compte as table of ref CompteType;
  2  /

Type created.

SQL> create type t_set_ref_pret as table of ref PretType;
  2  /

Type created.

SQL> create type t_set_ref_Operation as table of ref OperationType;
  2  /

Type created.

SQL> create type t_set_ref_Agence as table of ref AgenceType;
  2  /

Type created.
```

Figure 9: Execution de la requête de creation des tables de réference

Maintenant on peut créer les types avec leurs attributs comme il est montré dans la figure 10 et 11 pour le type SuccursaleType.

```
CREATE OR REPLACE TYPE SuccursaleType AS OBJECT (
  NumSucc NUMBER,
  nomSucc VARCHAR2(50),
  adresseSucc VARCHAR2(100),
  region VARCHAR2(20),
  Agences t_set_ref_Agence,

  MEMBER FUNCTION countMainAgences RETURN NUMBER,
  MEMBER FUNCTION pretANSEJ RETURN NUMBER
);
/
```

Figure 10: Code SQL creation type succursale Type

Ce type d'objet représente une succursale bancaire et comprend plusieurs attributs tels que le numéro de succursale, le nom, l'adresse et la région. De plus, il contient un ensemble de références vers les agences associées à cette succursale. En outre, deux fonctions membres sont définies par leur signature : countMainAgences, qui retourne le nombre d'agences principales liées à cette succursale, et pretANSEJ, qui retourne le nombre de prêts ANSEJ effectués par cette succursale. Figure 11 montre l'exécution :

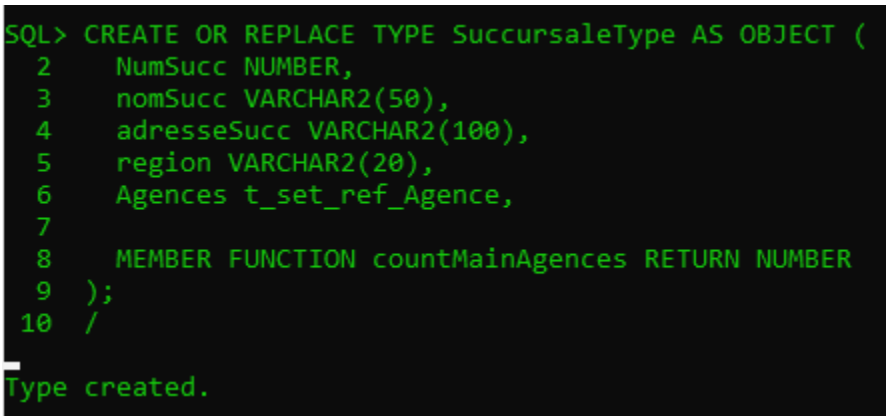


Figure 11: execution code pour la creation du type succursaleType

Pareille pour AgenceType ou on déclare les attributs et la signature des fonctions membres. De plus il est à noté que l'attribut Succursale fait référence à la succursale à laquelle l'agence est lié ; c'est donc une relation de type 1..1 dans le modèle UML, tandis que l'attribut ComptesAgence est une table imbriquée de référence aux objets comptes lié à l'instance de l'objet.

```sql
CREATE OR REPLACE TYPE AgenceType AS OBJECT (
  NumAgence NUMBER,
  nomAgence VARCHAR2(50),
  adresseAgence VARCHAR2(100),
  categorie VARCHAR2(20),
  Succursale REF SuccursaleType,
  ComptesAgence t_set_ref_Compte,
  MEMBER FUNCTION countPret RETURN NUMBER,
  MEMBER FUNCTION montantGlobalPret(dateDebut DATE, dateFin DATE)
    RETURN NUMBER

) cascade;
```

Figure 12: Code SQL pour le type Agence

Son éxecution est motrer dans la figure 13.

```
SQL> CREATE OR REPLACE TYPE AgenceType AS OBJECT (
  2    NumAgence NUMBER,
  3    nomAgence VARCHAR2(50),
  4    adresseAgence VARCHAR2(100),
  5    categorie VARCHAR2(20),
  6    Succursale REF SuccursaleType,
  7    ComptesAgence t_set_ref_Compte,
  8    MEMBER FUNCTION countPret RETURN NUMBER,
  9    MEMBER FUNCTION montantGlobalPret(dateDebut DATE, dateFin DATE) RETURN
 NUMBER,
 10    MEMBER FUNCTION pretANSEJ RETURN NUMBER
 11  );
 12
 13  /
```

Figure 13: Création type agence

Les autres types suivent la même logique que les types succursales et agence. Afin de concrétisé les association tel quelle sont décrite dans le modèle UML, on utilise une attribut référence avec REF pour une association 1..1, ou une table imbriquée de référence pour une association 0..N. Les figures 14, 15 et 16 montre le reste des types.

```
CREATE OR REPLACE TYPE ClientType AS OBJECT (
  NumClient NUMBER,
  NomClient VARCHAR2(50),
  TypeClient VARCHAR2(20),
  AdresseClient VARCHAR2(100),
  NumTel VARCHAR2(20),
  Email VARCHAR2(50),
  ComptesClient t_set_ref_Compte
```

Figure 14: Type Client

```
CREATE OR REPLACE TYPE CompteType AS OBJECT (
  NumCompte NUMBER,
  dateOuverture DATE,
  etatCompte VARCHAR2(20),
  Solde NUMBER,
  Client REF ClientType,
  Agence REF AgenceType,
  prets t_set_ref_pret,
  Operations t_set_ref_Operation
);
/
```

Figure 15: Type compte

```
SQL> create TABLE pret of PretType(
  2    constraint pk_pret primary key(NumPret),
  3    constraint ck_pret_type check (UPPER(typePret) in ('ANSEJ', 'VEHICULE'
, 'IMMOBILIER', 'ANJEM'))
  4  )
  5  ;

Table created.
```

Figure 16: Creation type pret

```
SQL> create TABLE operation of OperationType(
  2    constraint pk_operation primary key(NumOperation)
  3  );
```

Avant de pouvoir crée la corp des fonctions membres si elles doivent retourner un type complexe tel un tableau on doit créer un type spécial pour cette occasion comme le montre la figure 17 qui est un type spécialement crée pour la fonction PrêtAnsej dont le corp (body) est décrit dans la figure 18.

```
CREATE OR REPLACE TYPE pretANSEJType AS OBJECT (
  numagence NUMBER,
  numsucc NUMBER
);

CREATE OR REPLACE TYPE pretANSEJTableType AS TABLE OF pretANSEJType
    ;
```

Figure 17: Type pretANSEJ

3.2. Définition des méthodes

La figure 18 et 19 illustre le corps du type SuccursaleType dans une base de données Oracle. Ce corps de type définit deux fonctions membres : countMainAgences et pretANSEJ.

La fonction countMainAgences permet de compter le nombre d'agences principales associées à une succursale. Elle effectue une requête SQL pour sélectionner et compter les agences principales dans l'ensemble des agences associées à la succursale.

La fonction pretANSEJ retourne un tableau contenant les prêts ANSEJ (Agence Nationale de Soutien à l'Emploi des Jeunes) associés à une succursale donnée. Elle parcourt les comptes de la succursale et sélectionne les prêts de type ANSEJ pour les agences secondaires. Le résultat est retourné sous forme d'un tableau de type pretANSEJTableType.

Ces fonctions membres permettent d'effectuer des opérations spécifiques sur les objets de type SuccursaleType, fournissant ainsi une fonctionnalité utile pour l'analyse et la gestion des données dans le contexte d'une application bancaire.

```
CREATE OR REPLACE TYPE pretANSEJTableType AS TABLE OF pretANSEJType
    ;

CREATE OR REPLACE TYPE BODY SuccursaleType AS

  MEMBER FUNCTION countMainAgences RETURN NUMBER IS
    nb NUMBER;
  BEGIN

      SELECT count(VALUE(b).numagence) into nb
      FROM TABLE(self.agences) b
      WHERE VALUE(b).categorie = 'PRINCIPALE';
   return nb;
  END countMainAgences;

  MEMBER FUNCTION pretANSEJ RETURN pretANSEJTableType IS
```

Figure 18: Fonction countMainAgence

```
    v_result pretANSEJTableType := pretANSEJTableType();
  BEGIN
    FOR rec IN (
      SELECT a.numagence, deref(a.succursale).numsucc
      FROM self a, table(a.comptesagence) b, table(value(b).prets)
    c
      WHERE value(c).typepret = 'ANSEJ' AND a.categorie = '
    SECONDAIRE'
    ) LOOP
      v_result.extend;
      v_result(v_result.count) := pretANSEJType(rec.numagence, rec.
    numsucc);
    END LOOP;

    RETURN v_result;
  END pretANSEJ;

END;
/
```

Figure 19: fonction pretANSEJ

La figure 18 présente le corps du type AgenceType dans une base de données Oracle. Ce corps de type définit deux fonctions membres :

La fonction countPret permet de compter le nombre total de prêts effectués par l'agence. Elle réalise une requête SQL pour compter le nombre de prêts présents dans l'ensemble des comptes de l'agence.

La fonction montantGlobalPret calcule le montant total des prêts effectués par l'agence dans une période spécifiée entre deux dates. Elle utilise une requête SQL pour calculer la somme des montants des prêts dans l'ensemble des comptes de l'agence, en filtrant les prêts en fonction de leurs dates d'effet comprises entre les dates spécifiées.

Ces fonctions membres fournissent des fonctionnalités essentielles pour l'analyse des données relatives aux prêts effectués par une agence dans une application bancaire.

```
CREATE OR REPLACE TYPE BODY AgenceType AS
  -- Function to count loans made by the agency
  MEMBER FUNCTION countPret RETURN NUMBER IS
    v_total NUMBER;
  BEGIN
    SELECT
      COUNT(value(c).numpret) INTO v_total
    FROM
      TABLE(self.comptesAgence) b,
      TABLE(value(b.prets)) c;
    RETURN v_total;
  END countPret;

  -- Function to calculate the total amount of loans made by the
    agency between specified dates
  MEMBER FUNCTION montantGlobalPret(dateDebut DATE, dateFin DATE)
    RETURN NUMBER IS
    v_total NUMBER;
  BEGIN
    SELECT
      SUM(value(c).montant) INTO v_total
    FROM
      TABLE(self.comptesAgence) b,
      TABLE(value(b.prets)) c
    WHERE
      c.dateEffet BETWEEN dateDebut AND dateFin;
    RETURN v_total;
  END montantGlobalPret;

END;
/
```

Figure 20: fonctions countPret et montantGlobalPret

3.3. Création des tables avec les contraintes

La figure 21 présente la création des tables de la base de données pour les types SuccursaleType, AgenceType, ClientType, CompteType, OperationType et PretType.

La table "succursale" est définie pour stocker des objets de type SuccursaleType, avec une contrainte de clé primaire sur l'attribut NumSucc et une contrainte de vérification sur l'attribut région pour s'assurer que seules les régions spécifiées ('EST', 'OUEST', 'NORD', 'SUD') sont valides. Les références vers les agences sont stockées dans une table imbriquée nommée "tab_ref_Agence".

La table "agence" est créée pour stocker des objets de type AgenceType, avec une contrainte de clé primaire sur l'attribut NumAgence et une contrainte de vérification sur l'attribut catégorie pour garantir que seules les catégories spécifiées ('PRINCIPALE', 'SECONDAIRE') sont valides. Les comptes associés à chaque agence sont stockés dans une table imbriquée nommée "tab_comptes".

La table "client" est définie pour stocker des objets de type ClientType, avec une contrainte de clé primaire sur l'attribut NumClient et une contrainte de vérification sur l'attribut TypeClient pour s'assurer que seuls les types spécifiés ('PARTICULIER', 'ENTREPRISE') sont valides. Les comptes associés à chaque client sont stockés dans une table imbriquée nommée "tab_comptes_client".

La table "compte" est créée pour stocker des objets de type CompteType, avec une contrainte de clé primaire sur l'attribut NumCompte et une contrainte de vérification sur l'attribut etatCompte pour garantir que seuls les états spécifiés ('ACTIF', 'BLOQUEE') sont valides. Les prêts et les opérations associés à chaque compte sont stockés dans des tables imbriquées nommées "tab_prets" et "tab_operations", respectivement.

La table "operation" est définie pour stocker des objets de type OperationType, avec une contrainte de clé primaire sur l'attribut NumOperation.

La table "pret" est créée pour stocker des objets de type PretType, avec une contrainte de clé primaire sur l'attribut NumPret et une contrainte de vérification sur l'attribut typePret pour garantir que seuls les types spécifiés ('ANSEJ', 'VEHICULE', 'IMMOBILIER', 'ANJEM') sont valides.

```sql
create TABLE succursale of SuccursaleType(
  constraint pk_succursale primary key(NumSucc),
  constraint ck_region check (UPPER(region) in ('EST', 'OUEST', '
    NORD', 'SUD'))
)
nested TABLE Agences store as tab_ref_Agence;

create TABLE agence of AgenceType(
  constraint pk_agence primary key(NumAgence),
  constraint ck_categorie_agence check (UPPER(categorie) in ('
    PRINCIPALE', 'SECONDAIRE'))
)
nested TABLE ComptesAgence store as tab_comptes;

create TABLE client of ClientType(
  constraint pk_client primary key(NumClient),
  constraint ck_type_client check (UPPER(TypeClient) in ('
    PARTICULIER', 'ENTREPRISE'))
)
nested TABLE ComptesClient store as tab_comptes_client;

create TABLE compte of CompteType(
  constraint pk_compte primary key(NumCompte),
  constraint ck_etat_compte check (UPPER(etatCompte) in ('ACTIF', '
    BLOQUEE'))
)
nested TABLE prets store as tab_prets
nested TABLE Operations store as tab_operations;

create TABLE operation of OperationType(
  constraint pk_operation primary key(NumOperation)
);

create TABLE pret of PretType(
  constraint pk_pret primary key(NumPret),
  constraint ck_pret_type check (UPPER(typePret) in ('ANSEJ', '
    VEHICULE', 'IMMOBILIER', 'ANJEM'))
)
;
```

Figure 21: Creation des tables

4. Création des instances dans les tables

Création des valeurs pour succursale

La figure 22 illustre l'insertion d'une nouvelle entrée dans la table "succursale" de la base de données. Cette insertion crée une nouvelle succursale avec les caractéristiques suivantes :

- NumSucc : 002
- nomSucc : CitiBank-Alger-ouest
- adresseSucc : 15 Rue Belouizdad
- région : OUEST
- Agences : un ensemble vide de références vers des agences (t_set_ref_Agence())

Cette opération ajoute une nouvelle entrée à la table "succursale" représentant une succursale de la banque CitiBank située à Alger dans la région OUEST, sans agence associée pour le moment.

```
INSERT INTO Succursale
VALUES (SuccursaleType(002, 'CitiBank-Alger-ouest', '15 Rue
    Belouizdad', 'OUEST', t_set_ref_Agence()));
```

Figure 22: Insertion dans la table succursale

Son execution est montrée dans la figure 23

```
SQL> insert into succursale values(SuccursaleType(001, 'CitiBank-Alger-est', '28 Rue Didouche', 'EST', t_set_ref_Agence()));

1 row created.
```

Figure 23: execution de l'insertion

On repète cette operation six fois pour crée les six succursale en repectant la contrainte que la clé pincipale soit être de type 001….002….etc.

Création des valeurs pour Agence

Dans la figure 24, nous voyons une opération d'insertion dans la table "agence" de la base de données. Cette insertion crée une nouvelle agence avec les caractéristiques suivantes :

- NumAgence : 101
- nomAgence : CitiBank-agence-alger-01
- adresseAgence : 9 Rue Colonel Mentouri
- catégorie : PRINCIPALE
- succursale : une référence à la succursale avec le numéro 001 de la banque CitiBank
- ComptesAgence : un ensemble vide de références vers des comptes (t_set_ref_Compte())

Cette opération ajoute une nouvelle entrée à la table "agence", représentant une agence principale de la banque CitiBank située à Alger, avec le numéro d'agence 101 et associée à la succursale numéro 001. Son execution est illustré par la figure 25.

```
INSERT INTO agence
VALUES (AgenceType(101, 'CitiBank-agence-alger-01', '9 Rue Colonel
    mentouri', 'PRINCIPALE', (select ref(a) from succursale a where
     a.NumSucc = 001),t_set_ref_Compte()));
```

Figure 24: Insertion dans la table agence

```
SQL> INSERT INTO agence
  2  VALUES (AgenceType(102, 'CitiBank-agence-alger-02', '12 Rue Belouizdad', 'SECONDAIRE', (SELECT REF(a) FROM succursale a WHERE a.NumSucc = 001), t_set_re
f_Compte()));

1 row created.
```

Figure 25: execution de l'insertion dans agence

Maintenant il est necessaire de connecté l'agence avec sa succursale avec l'opération de mise à jour suivante (figure 26)

```
UPDATE Succursale
SET Agences = t_set_ref_Agence(SELECT REF(v)
FROM agence v WHERE TO_CHAR(v.NumAgence) LIKE '1%')
WHERE NumSucc = 001;

insert into table (select l.Agences from succursale l where numsucc
    = 3)
(select ref(c) from agence c where TO_CHAR(v.NumAgence) LIKE '3%');
```

Figure 26: Mise à jour de la table succursale

Dans la figure 26, nous observons une instruction UPDATE sur la table "succursale" de la base de données. Cette instruction met à jour les références aux agences associées à la succursale ayant le numéro NumSucc égal à 001.

L'opération consiste à sélectionner les références aux agences de la table "agence" où le numéro d'agence commence par le chiffre 1 (LIKE '1%'), puis à les convertir en références et les placer dans l'ensemble Agences de la succursale correspondante. L'execution de l'insertion de la reference d'une agence dans une succursale est montrer dans la figure 27.

On refet ses opérations pour les 24 agences restantes.

```
SQL> insert into table (select l.Agences from succursale l where numsucc= 2)
  2  (select ref(c) from agence c where TO_CHAR(c.NumAgence) LIKE '2%');

4 rows created.
```

Figure 27: execution de la requete d'insertion d'une reference

Création des valeurs pour client et compte

Dans la figure 28, nous observons deux opérations d'insertion dans les tables de la base de données.

La première insertion concerne la table "client". Elle crée une nouvelle entrée représentant un client avec les caractéristiques suivantes :

- NumClient : 50045
- Nom : Hamza Taourirt
- TypeClient : PARTICULIER
- Adresse : 299 Rue des Bananes Trop Mûr
- Téléphone : 0565500022
- Email : hamzatrt@gmail.com
- ComptesClient : un ensemble vide de références vers des comptes (t_set_ref_Compte())

La seconde insertion concerne la table "compte". Elle crée un nouveau compte avec les caractéristiques suivantes :

- NumCompte : 1010050045
- DateOuverture : 30 Avril 2022
- EtatCompte : ACTIF
- Solde : 350000
- Client : une référence au client avec le numéro 50045
- Agence : une référence à l'agence avec le numéro 101
- Prêts : un ensemble vide de références vers des prêts (t_set_ref_pret())
- Opérations : un ensemble vide de références vers des opérations (t_set_ref_Operation())

```
INSERT INTO client
VALUES (ClientType(50045, 'hamza taourirt', 'PARTICULIER', '299 RUE
     DES BANANES TROP MUR', '0565500022', 'hamzatrt@gmail.com',
    t_set_ref_Compte()));

INSERT INTO compte
VALUES (CompteType(1010050045, TO_DATE('30-04-2022', 'DD-MM-YYYY'),
     'ACTIF', 350000, (SELECT REF(a) FROM client a WHERE a.
    NumClient = 50045),(SELECT REF(b) FROM agence b WHERE b.
    NumAgence = 101), t_set_ref_pret(), t_set_ref_Operation()));
```

Figure 28: Insertion dans compte et client

On insert plus de 100 clients et comptes de cette manière.

Dans la figure 29, nous avons une opération d'insertion dans une table imbriquée de la base de données. Cette opération insère des références vers des comptes dans la table imbriquée "ComptesAgence" de l'agence ayant le numéro NumAgence égal à 102. Les références sont sélectionnées à partir de la table "compte" où le numéro de compte commence par "102". Cela signifie que tous les comptes dont le numéro commence par "102" sont associés à l'agence ayant le numéro 102. Cela concrétise la relation d'association entre une agence et ses

```
INSERT INTO TABLE (SELECT l.ComptesAgence FROM agence l WHERE
    NumAgence = 102)
(SELECT REF(c) FROM compte c WHERE TO_CHAR(c.NumCompte) LIKE '102%'
    );
```

Figure 29: insertion des references de comptes dans l'agence lié

```
SQL> -- For Agence 102
SQL> INSERT INTO TABLE (SELECT l.ComptesAgence FROM agence l WHERE NumAgence = 102)
  2  (SELECT REF(c) FROM compte c WHERE TO_CHAR(c.NumCompte) LIKE '102%');

4 rows created.
```

Figure 30: Connecté les comptes avec la table agence lié

Insertion dans les tables opération et prêt :
Avant de pouvoir inséré les instances, il est nécessaire de crée certaines procédures pour faciliter les mises à jour dans les tables lié au tables opération et prêt.

Pour la table operations on a deux procedure :
a figure 31 et 32 présente deux procédures stockées en langage PL/SQL.

La première procédure, "**update_solde_after_insert**", est conçue pour mettre à jour le solde d'un compte après l'insertion d'une opération. Elle prend deux paramètres en entrée : p_NumCompte, qui est le numéro de compte concerné, et p_NumOperation, qui est le numéro de l'opération nouvellement insérée. La procédure récupère le montant de l'opération en fonction de sa nature

(retrait ou dépôt) à partir de la table "operation", puis met à jour le solde du compte correspondant dans la table "compte".

La deuxième procédure, "**ajoutCmptProcedure**", ne contient pas de logique de traitement spécifique. Elle est fournie avec un commentaire indiquant que votre logique de traitement doit être implémentée à cet endroit. La procédure prend deux arguments en entrée, arg1 et arg2, et comprend des déclarations de variables pour afficher les arguments et pour effectuer un traitement supplémentaire.

Ces procédures stockées peuvent être appelées et exécutées depuis une application cliente ou depuis d'autres parties de votre système de base de données pour automatiser des tâches de gestion et de traitement des données.

L'utilisation des deux procédures est illustrée dans la figure 33.

```
CREATE OR REPLACE PROCEDURE update_solde_after_insert(
    p_NumCompte IN NUMBER,
    p_NumOperation IN NUMBER
) AS
    v_amount NUMBER;
BEGIN
    -- Determine the amount to be added or subtracted based on the
    nature of the operation
    SELECT CASE
               WHEN UPPER(NatureOp) = 'RETRAIT' THEN -1 * montantOp
     -- Subtract the amount for a retrait
               WHEN UPPER(NatureOp) = 'DEPOT' THEN montantOp -- Add
     the amount for a depot
               ELSE 0
           END INTO v_amount
    FROM operation
    WHERE NumOperation = p_NumOperation;

    -- Print the value of v_amount for debugging
    DBMS_OUTPUT.PUT_LINE('v_amount: ' || v_amount);

    -- Update the Solde column in the corresponding Compte row
    UPDATE compte
    SET Solde = Solde + v_amount
    WHERE NumCompte = p_NumCompte;

    -- Print a message indicating successful update
    DBMS_OUTPUT.PUT_LINE('Solde updated successfully');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        -- Print a message for no data found
        DBMS_OUTPUT.PUT_LINE('No data found for NumOperation: ' ||
    p_NumOperation);
    WHEN OTHERS THEN
        -- Print the error message
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

Figure 31: Procedure pour mettre à jour le solde d'un compte

```
PROCEDURE ajoutCmptProcedure(arg1 IN NUMBER, arg2 IN NUMBER) AS
BEGIN

    INSERT INTO TABLE (SELECT p.operations FROM compte p WHERE p.numCompte = arg

    (SELECT REF(c) FROM operation c WHERE c.numoperation = arg1);

    DBMS_OUTPUT.PUT_LINE('Processing operation ' || arg1);
    DBMS_OUTPUT.PUT_LINE('Argument 1: ' || arg1);
    DBMS_OUTPUT.PUT_LINE('Argument 2: ' || arg2);
END ajoutCmptProcedure;


-- NumCompte 2030010303
INSERT INTO operation
VALUES (
    OperationType(4, 'Depot', 60000, TO_DATE('01-05-24', 'DD-MM-YY'), 'Dépôt argent',
                  (SELECT REF(c) FROM compte c WHERE c.NumCompte = 2030010303))
);
```

Figure 32: Predure pour connecter une operation au compte lié

```
BEGIN
    ajoutCmptProcedure(4, 2030010303);
END;
/

BEGIN
    update_solde_after_insert(2030010303, 4);
END;
/
```

Figure 33: Utlisation des deux procédures

Une exécution de ces code est montrée dans la figure 34

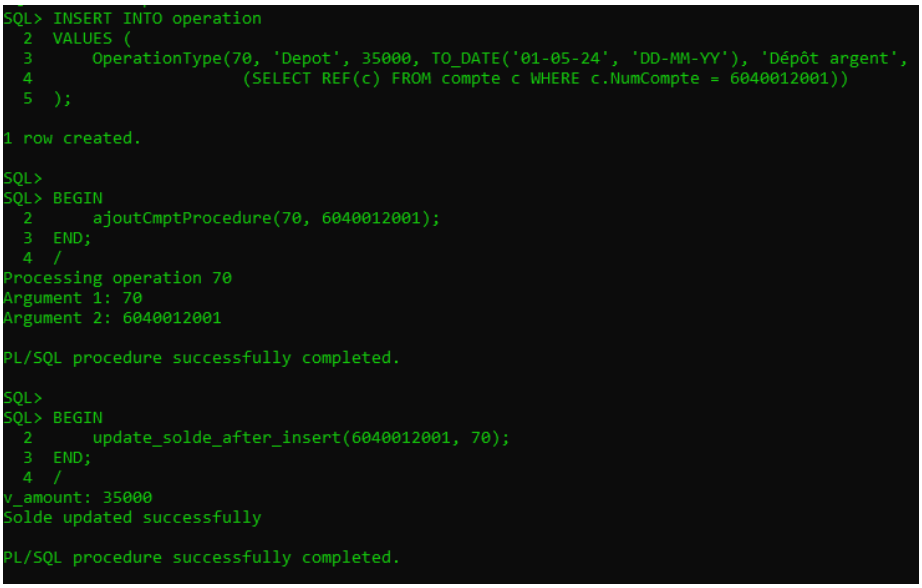


Figure 34: Execution code lié au opérations et comptes

En ce qui concerne les prêts la table aussi a besoin d'opérations automatique pour mettre à jour les références entre compte et prêts ainsi que la mise à jour de la solde après l'acceptance d'un emprunt.

Dans la figure 35, une procédure stockée nommée "update_solde_after_borrow" est présentée. Cette procédure est conçue pour mettre à jour le solde d'un compte après un emprunt. Elle prend deux paramètres en entrée : p_NumCompte, qui est le numéro de compte concerné, et p_NumPret, qui est le numéro du prêt effectué. La procédure récupère le montant emprunté à partir de la table "pret" en fonction du numéro de prêt spécifié, puis met à jour le solde du compte correspondant dans la table "compte". Des messages de débogage sont affichés pour indiquer le montant emprunté et le succès de la mise à jour.

```
CREATE OR REPLACE PROCEDURE update_solde_after_borrow(
    p_NumCompte IN NUMBER,
    p_NumPret IN NUMBER
) AS
    v_amount NUMBER;
BEGIN
    -- Determine the borrowed amount based on the specified Pret
    SELECT MONTANTPRET INTO v_amount
    FROM pret
    WHERE NUMPRET = p_NumPret;

    -- Print the borrowed amount for debugging
    DBMS_OUTPUT.PUT_LINE('Borrowed amount: ' || v_amount);

    -- Update the Solde column in the corresponding Compte row
    UPDATE compte
    SET Solde = Solde + v_amount
    WHERE NumCompte = p_NumCompte;

    -- Print a message indicating successful update
    DBMS_OUTPUT.PUT_LINE('Solde updated successfully');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        -- Print a message for no data found
        DBMS_OUTPUT.PUT_LINE('No data found for NumPret: ' ||
    p_NumPret);
    WHEN OTHERS THEN
        -- Print the error message
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

Figure 35: Procedure de mise à jour d'un compte aprés emprunt

Dans la figure 36, une autre procédure stockée appelée "ajoutCmptPretProcedure" est présentée. Cette procédure ne contient pas de logique de traitement spécifique, mais elle est conçue pour être utilisée dans le contexte de l'ajout d'un prêt à un compte. Elle prend deux arguments en entrée, arg1 et arg2, qui représentent respectivement le numéro du prêt et le numéro du compte. La procédure insère une référence au prêt dans la table imbriquée "prets" du compte spécifié. Des messages de débogage sont inclus pour afficher les arguments passés à la procédure.

```
CREATE OR REPLACE PROCEDURE ajoutCmptPretProcedure(arg1 IN NUMBER,
    arg2 IN NUMBER) AS

BEGIN
    -- Your logic to process the operation goes here
    -- You can also use the arguments passed to the procedure
    INSERT INTO TABLE (SELECT p.prets FROM compte p WHERE p.
    numCompte = arg2)

    (SELECT REF(c) FROM pret c WHERE c.numpret = arg1);

    -- Optionally, you can include DBMS_OUTPUT statements for
    debugging purposes

    DBMS_OUTPUT.PUT_LINE('Processing pret ' || arg1);
    DBMS_OUTPUT.PUT_LINE('Argument 1: ' || arg1);
    DBMS_OUTPUT.PUT_LINE('Argument 2: ' || arg2);

    -- Your processing logic here
END ajoutCmptPretProcedure;
/
```

Figure 36: procedure pour lié un compte au pêt

Figure 37 montre l'execution du code dans les figures 36 et 37 :

```
SQL> INSERT INTO pret
  2  VALUES (
  3      pretType(
  4          2,
  5          6000,
  6          TO_DATE('2002-11-30', 'YYYY-MM-DD'),
  7          24,
  8          'ANJEM',
  9          6.0,
 10          0.00,  -- This signifies that it was repaid
 11          (SELECT REF(c) FROM compte c WHERE c.NumCompte = 4010010904) -- TYPE
 12      )
 13  );

1 row created.

SQL>
SQL> BEGIN
  2      ajoutCmptPretProcedure(2, 4010010904);
  3  END;
  4  /
Processing pret 2
Argument 1: 2
Argument 2: 4010010904

PL/SQL procedure successfully completed.

SQL>
SQL> BEGIN
  2      update_solde_after_borrow(4010010904, 2);
  3  END;
  4  /
Borrowed amount: 6000
Solde updated successfully

PL/SQL procedure successfully completed.
```

Figure 37: insertion d'un prêt

On insère plus de quarante prêts et un nombre supérieur d'opérations.

5. Langage d'interrogation des données

5.1. Liste des comptes d'une agence appartenant à des entreprises

Figure 38 : Cette requête vise à déterminer le nombre de comptes détenus par des entreprises au sein d'une agence spécifique, identifiée par son numéro d'agence (101). Elle recherche dans la table des agences pour les comptes associés à l'agence donnée, puis vérifie le type de chaque client pour s'assurer qu'il s'agit d'une entreprise. En comptant le nombre de comptes répondant à ces critères, elle fournit un indicateur de l'activité commerciale avec des entreprises dans cette agence particulière.

```
select a.numagence, deref(a.succursale).numsucc
  from agence a, table(a.comptesagence) b, table(value(b).prets) c
  where value(c).typepret in 'ANSEJ'and a.categorie in 'SECONDAIRE'
    ;
```

Figure 38: Liste des comptes entreprise d'une agence donnée

```
SQL> select count(deref(value(a).client).typeclient) from agence b, table(b.comptesagence) a where numagence = 101 and upper(deref(value(a).client).typeclien
t) = 'ENTREPRISE';

COUNT(DEREF(VALUE(A).CLIENT).TYPECLIENT)
----------------------------------------
                                       1
```

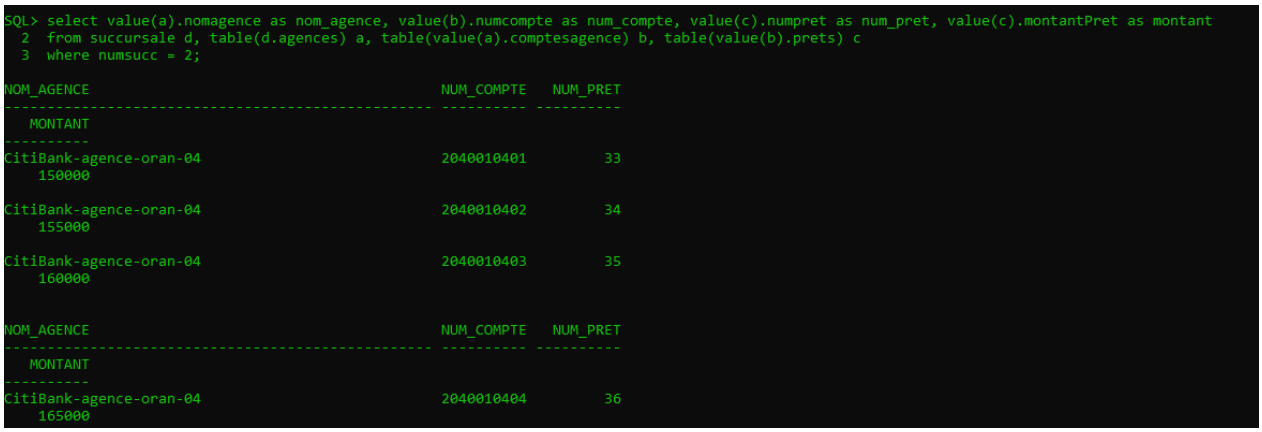
Figure 39: exécution de la requête

5.2. Liste des prêts effectués auprès des agences rattachées à une succursale spécifique

La requête dans la figure 40 récupère des informations détaillées sur les prêts associés aux agences rattachées à une succursale spécifique (numéro de succursale 002). Elle joint les tables des succursales, des agences, des comptes d'agence et des prêts pour extraire le nom de l'agence, le numéro de compte, le numéro de prêt et le montant de chaque prêt. Cette requête fournit une vue d'ensemble des activités de prêt dans les agences associées à une succursale donnée.

```
select value(a).nomagence as nnom_agence, value(b).numcompte, value
    (c).numpret as num_pret, value(c).montantPret as montant
from succusale d, table(d.agences) a, table(value(a).comptesagence)
     b, table(value(b).prets) c
where numsucc = 002;
```

Figure 40: prêts dans les agences d'une succursale donnée (002)



```
SQL> select value(a).nomagence as nom_agence, value(b).numcompte as num_compte, value(c).numpret as num_pret, value(c).montantPret as montant
  2  from succursale d, table(d.agences) a, table(value(a).comptesagence) b, table(value(b).prets) c
  3  where numsucc = 2;

NOM_AGENCE                                         NUM_COMPTE   NUM_PRET
-------------------------------------------------- ---------- ----------
   MONTANT
----------
CitiBank-agence-oran-04                            2040010401         33
    150000

CitiBank-agence-oran-04                            2040010402         34
    155000

CitiBank-agence-oran-04                            2040010403         35
    160000


NOM_AGENCE                                         NUM_COMPTE   NUM_PRET
-------------------------------------------------- ---------- ----------
   MONTANT
----------
CitiBank-agence-oran-04                            2040010404         36
    165000
```

Figure 41: éxcution de la requête

5.3. Comptes sans opérations de débit entre 2000 et 2022

Dans la figure 42 cette requête identifie les comptes sur lesquels aucun dépôt n'a été effectué entre les années 2000 et 2022. Elle sélectionne les numéros de compte à partir de la table des comptes et exclut ceux pour lesquels des opérations de dépôt ont été enregistrées dans la période spécifiée. Cela peut aider à détecter les comptes inactifs ou peu utilisés pendant cette période.

```
select numcompte
from compte
where numcompte not in(
  select a.numcompte
  from compte a, table(a.operations) b
  where value(b).natureop = 'Depot'
  and value(b).dateop BETWEEN TO_DATE('01-01-2000', 'DD-MM-YYYY')
    AND TO_DATE('12-31-2022', 'DD-MM-YYYY')
  );
```

Figure 42: Requête compte débit

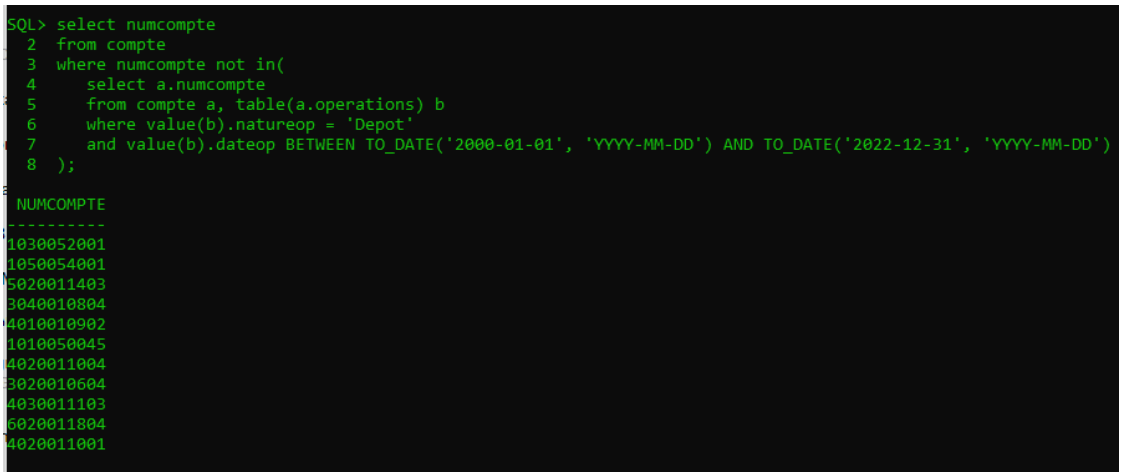


Figure 43: éxecution de la requête

5.4. Montant total des crédits effectués sur un compte spécifique en 2023

Figure 44 : Cette requête calcule la somme totale des prêts sur un compte spécifique (numéro de compte 4020011002). Elle parcourt les prêts associés à ce compte et additionne les montants de tous les prêts. Cela peut être utile pour évaluer le niveau d'endettement ou l'utilisation des services de prêt par un client particulier.

```
select sum(value(b).montantpret) as somme_totale_prets
  from compte a, table(a.prets) b
  where numcompte = 4020011002;
```

Figure 44: requête montant des credit

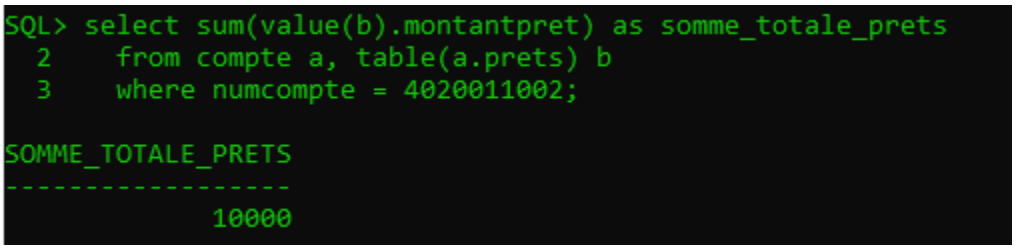


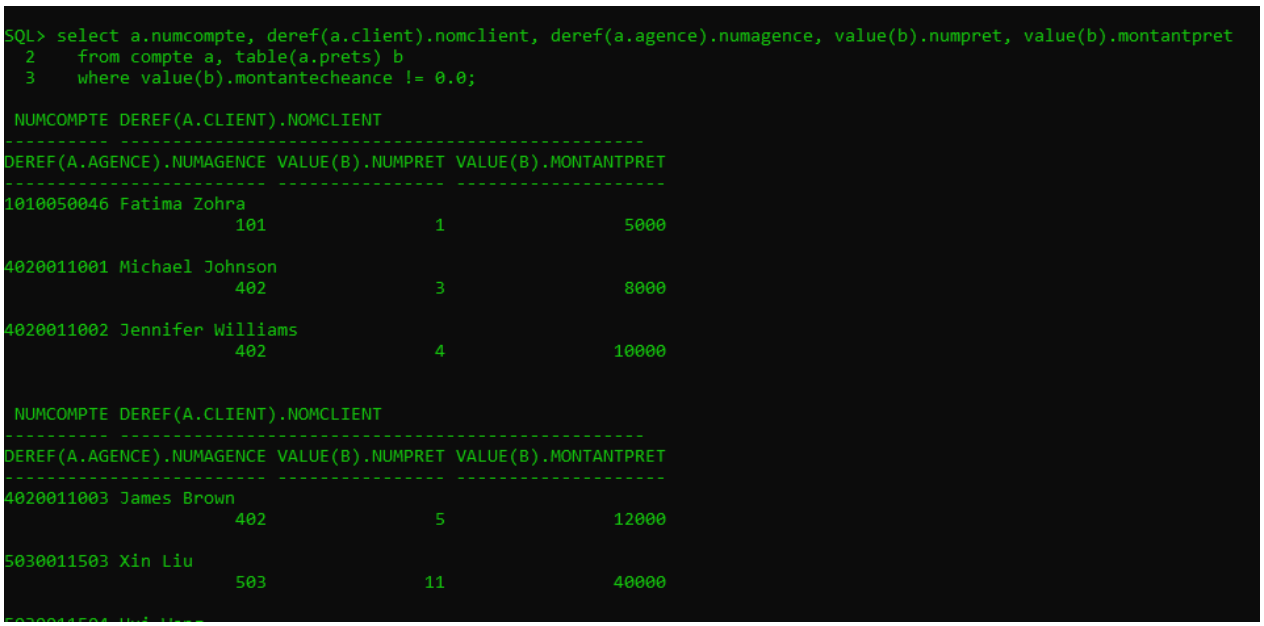
Figure 45: exécution de la requête montant des credits

5.5. Prêts non encore soldés

Figure 46 : Cette requête récupère les détails des prêts qui n'ont pas encore été entièrement remboursés. Elle sélectionne les informations pertinentes sur les prêts à partir de la table des comptes, en excluant ceux pour lesquels le montant restant à rembourser est nul. Cela fournit une vue des prêts en cours et de leur état de remboursement.

```
select a.numcompte, deref(a.client).nomclient, deref(a.agence).
    numagence, value(b).numpret, value(b).montantpret
  from compte a, table(a.prets) b
  where value(b).montantecheance != 0.0;
```

Figure 46: requête des prêts non encore solde



```
SQL> select a.numcompte, deref(a.client).nomclient, deref(a.agence).numagence, value(b).numpret, value(b).montantpret
  2    from compte a, table(a.prets) b
  3    where value(b).montantecheance != 0.0;

 NUMCOMPTE DEREF(A.CLIENT).NOMCLIENT
---------- --------------------------------------------------
DEREF(A.AGENCE).NUMAGENCE VALUE(B).NUMPRET VALUE(B).MONTANTPRET
------------------------- ---------------- --------------------
1010050046 Fatima Zohra
                      101                1                 5000

4020011001 Michael Johnson
                      402                3                 8000

4020011002 Jennifer Williams
                      402                4                10000


 NUMCOMPTE DEREF(A.CLIENT).NOMCLIENT
---------- --------------------------------------------------
DEREF(A.AGENCE).NUMAGENCE VALUE(B).NUMPRET VALUE(B).MONTANTPRET
------------------------- ---------------- --------------------
4020011003 James Brown
                      402                5                12000

5030011503 Xin Liu
                      503               11                40000
```

Figure 47: execution de la requete

5.6. Compte le plus actif en 2023

```
SELECT NumCompte
FROM (
    SELECT c.NumCompte, COUNT(*) AS operation_count
    FROM Compte c
    JOIN TABLE(value(c).Operations) o ON 1=1
    WHERE o.NatureOp IN ('retrait', 'depot')
    GROUP BY c.NumCompte
    ORDER BY COUNT(*) DESC
)
WHERE ROWNUM = 1;
```

Figure 48: Requête pour le compte le plus actif

```
SQL> select numCompte, nbOperations from(
  2  Select a.NumCompte, count(value(b).natureOp) as nbOperations
  3  from compte a, table(a.operations) b
  4  where value(b).natureOp IN ('Retrait', 'Depot')
  5  group by a.NumCompte
  6  order by COUNT(*) DESC
  7  )
  8  WHERE ROWNUM = 1;

 NUMCOMPTE NBOPERATIONS
---------- ------------
2030010304            6
```

Figure 49: exécution de la requête

Partie II : NoSQL – Modèle orienté «documents »

6. Modélisation orientée document

6.1. Proposer une modélisation orientée document de la base de données décrite dans la partie I, dans ce cas.

Afin de transformer le model orienté objet-relationel en un model document centré sur prêt, on analyse le schéma UML.

Du schéma on extrait les associations en commençant par la classe prêt. On remarque que la class prêt est lié par association 1..1 a un compte, ceci ce traduit par un objet représentant les données compte à l'intérieur de l'objet prêt. Par la suite, en remarque que compte est lié par une relation 1..1 a une agence, une relation N…M avec des opérations et une relation 1..1 avec un client. Les objets client et agence seront des objets alors qu'opération devient un tableau d'objets tous trois imbriquée à l'intérieur de l'objet compte. De plus, la classe agence est lié par une association 1..1 avec une succursale donc un nouveau niveau d'imbrication s'impose pour représenté les attributs de succursale.
De cette analyse on peut représenté le schéma relationnel de la partie I comme dans la figure 50 ;

les inconvénients de cette conception

La principale faiblesse de cette modélisation est lié au nombreux niveau d'imbrications qui causa un excès de consommation de ressource si les requêtes se porte sur les données positionnées dans de des niveaux profond. On particulier, on note que cette approche est mal adapté pour géré les opérations can comm les prêts, elles sont liées a la classe compte par une relation N..M ; cela causera une difficulté dans l'insertion de nouveaux prets lie à un compte que un déjà des prets. Il serait probablement préférable de sauvegarder les opérations dans une collection séparé.

```
const examples =
    [{

        "NumPret": "1",
        "montantPret": 5000,
        "dateEffet": "2024-05-02",
        "duree": 36,
        "typePret": "ANSEJ",
        "tauxInteret": 5,
        "montantEcheance": 150,

      "compte": {
        "NumCompte": "1010012311",
        "dateOuverture": "2023-01-15",
        "etatCompte": "ACTIF",
        "Solde": 10000,
        "client": {
          "NumClient": "CL00167851",
          "NomClient": "John Doe",
          "TypeClient": "Particulier",
          "AdresseClient": "123 Main St, City",
          "NumTel": "+1234567890",
          "Email": "john.doe@example.com"
        },
        "agence": {
          "NumAgence": "101",
          "nomAgence": "Main Street Branch",
          "adresseAgence": "123 Main St, City",
          "categorie": "PRINCIPALE",
          "succursale": {
            "NumSucc": "1",
            "nomSucc": "CityBank-Branch-capital",
            "adresseSucc": "456 Elm St, City",
            "region": "NORD"
          }
        },
        "Operations": [
          {
            "NumOperation": "OP001",
            "NatureOperation": "Depot",
            "montantOp": 200,
            "DateOp": "2024-05-02",
            "Observation": "Deposit to account"
          }
        ]

      }
  }
  ]
```

Figure 50: - Illustration de la modélisation sur un exemple

7. Remplir la base de données (via un script, ajouter d'autres données afin d'augmenter le volume de la base)

D'abord on crée une nouvelle base de données 'bank'. Qui contient la collection « prêt », puis on remplit cette collections avec des document de la structure prêt (figure 51). Il est à noter que cette étape a été réalisé sur un script javascript.

```
const conn = new Mongo();
  const db = conn.getDB('bank');
  const collection= db.createCollection('pret');
```

Figure 51 : connexion a la DB bank et création d'une collection prête

On ajoute les exemples dans une variable qui elle-même sera chargé dans la DB Bank en utilisant la requête comme illustré dans la figure

```
const result = db.pret.insertMany(examples);
  print(`${result.insertedCount} documents inserted into the
     collection`);
```

Figure 52: insetion des documents dans la collection prête

Figure 53: résultat de l'insertion (les deux premiers documents sur 40)

8. Réponse aux requêtes

8.1. Prêts effectués auprès de l'agence numéro 102

Cette requête utilise la méthode find() pour rechercher les prêts dans la collection "pret" où le numéro de l'agence est "102".

```
const operatio1 = db.pret.find({ "compte.agence.NumAgence": "102" })
    ;
```

Figure 54: Prêts effectués sur l'agence 102

Figure 55: Affichage de l'éxecution

8.2. Prêts effectués auprès des agences rattachées aux succursales de la région "Nord"

Cette requête utilise également la méthode find(), mais avec une condition plus complexe pour rechercher les prêts associés aux agences rattachées à des succursales de la région "NORD". Elle projette également certains champs spécifiques à afficher dans le résultat à l'aide du paramètre projection.

```
const operatio2 = db.pret.find({
    "compte.agence.succursale.region": "NORD"
  }, {
    "NumPret": 1,
    "compte.agence.NumAgence": 1,
    "compte.NumCompte": 1,
    "compte.client.NumClient": 1,
    "montantPret": 1,
    "_id": 0
  }).pretty();
```

Figure 56: Prêts dans les agences Nord

```
bank> db.pret.find({
...   "compte.agence.succursale.region": "NORD"
... }, {
...   "NumPret": 1,
...   "compte.agence.NumAgence": 1,
...   "compte.NumCompte": 1,
...   "compte.client.NumClient": 1,
...   "montantPret": 1,
...   "_id": 0
... }).pretty();
[
  {
    NumPret: '1',
    montantPret: 5000,
    compte: {
      NumCompte: '1010012311',
      client: { NumClient: 'CL00167851' },
      agence: { NumAgence: '101' }
    }
  },
  {
    NumPret: '2',
    montantPret: 7000,
    compte: {
      NumCompte: '1010022311',
      client: { NumClient: 'CL00234567' },
      agence: { NumAgence: '101' }
    }
  },
```

Figure 57: affichage partiel du résultat

8.3.Nombre total des prêts par agence dans une nouvelle collection ordonnée

Cette requête utilise l'opération d'agrégation aggregate() pour regrouper les prêts par le numéro de l'agence et calculer le nombre total de prêts par agence. Ensuite, elle trie les résultats par ordre décroissant et les écrit dans une nouvelle collection appelée "Agence-NbPrêts".

```
const operatio3 = db.pret.aggregate([
  {
    $group: {
      _id: "$compte.agence.NumAgence",
      NbPrets: { $sum: 1 }
    }
  },
  {
    $sort: { NbPrets: -1 }
  },
  {
    $out: "Agence-NbPrets" // Creez une nouvelle collection pour
       stocker les resultats
  }
]);

// Affichez le contenu de la nouvelle collection
db.getCollection("Agence-NbPrets").find();
```

Figure 58: Les trois stages de l'opération d'aggregation en pipeline

```
bank> db.getCollection("Agence-NbPrêts").find();
[
  { _id: '504', NbPrets: 4 },
  { _id: '203', NbPrets: 4 },
  { _id: '503', NbPrets: 3 },
  { _id: '604', NbPrets: 3 },
  { _id: '601', NbPrets: 3 },
  { _id: '105', NbPrets: 3 },
  { _id: '602', NbPrets: 3 },
  { _id: '101', NbPrets: 2 },
  { _id: '404', NbPrets: 2 },
  { _id: '204', NbPrets: 1 },
  { _id: '304', NbPrets: 1 },
  { _id: '102', NbPrets: 1 },
  { _id: '403', NbPrets: 1 },
  { _id: '301', NbPrets: 1 },
  { _id: '401', NbPrets: 1 }
]
```

Figure 59: Résultat de l'agregation

8.4. Prêts liés à des dossiers ANSEJ dans une collection dédiée

Cette requête utilise la méthode find() pour rechercher les prêts avec le type "ANSEJ". Elle projette certains champs spécifiques à inclure dans le résultat et stocke ces prêts dans une nouvelle collection appelée "pret-ansej" à l'aide des méthodes createCollection() et insertMany().

```
let pretsANSEJ = db.pret.find({
    "pret.typePret": "ANSEJ"
  }, {
    "pret.NumPret": 1,
    "pret.compte.client.NumClient": 1,
    "pret.montantPret": 1,
    "pret.dateEffet": 1,
    "_id": 0
  }).toArray();
```

Figure 60: Recherche des prets ANSEJ

```
// Creez une nouvelle collection pret-ansej et inserez les prets
   recuperes
  const operatio4Creation =db.createCollection("pret-ansej");
  const operatio4Insertion =db.getCollection("pret-ansej").
     insertMany(pretsANSEJ);
```

Figure 61: Création de la nouvelle collection pretsAgence

```
bank> db.getCollection("pret-ansej").find();
[
  {
    _id: ObjectId('6634ee48b6b7dbaa5646b7a2'),
    NumPret: '1',
    montantPret: 5000,
    dateEffet: '2024-05-02',
    compte: { client: { NumClient: 'CL00167851' } }
  },
  {
    _id: ObjectId('6634ee48b6b7dbaa5646b7a3'),
    NumPret: '16',
    montantPret: 9200,
    dateEffet: '2025-09-25',
    compte: { client: { NumClient: 'CL01678901' } }
  },
  {
    _id: ObjectId('6634ee48b6b7dbaa5646b7a4'),
    NumPret: '20',
    montantPret: 9200,
    dateEffet: '2026-01-20',
    compte: { client: { NumClient: 'CL02012345' } }
  },
  {
    _id: ObjectId('6634ee48b6b7dbaa5646b7a5'),
    NumPret: '26',
    montantPret: 9200,
    dateEffet: '2007-09-25',
```

Figure 62: Affichage partiel de la nouvelle collection

8.5.Prêts effectués par des clients de type "Particulier"

Cette requête utilise la méthode find() pour rechercher les prêts associés à des clients de type "Particulier". Elle projette certains champs spécifiques à inclure dans le résultat.

```
const operatio5 = db.pret.find(
   { "compte.client.TypeClient": "Particulier" },
   { "compte.client.NumClient": 1, "compte.client.NomClient": 1, "
     NumPret": 1, "montantPret": 1, "_id": 0 }
 );
```

Figure 63: prêts effectués par les clients particuliers

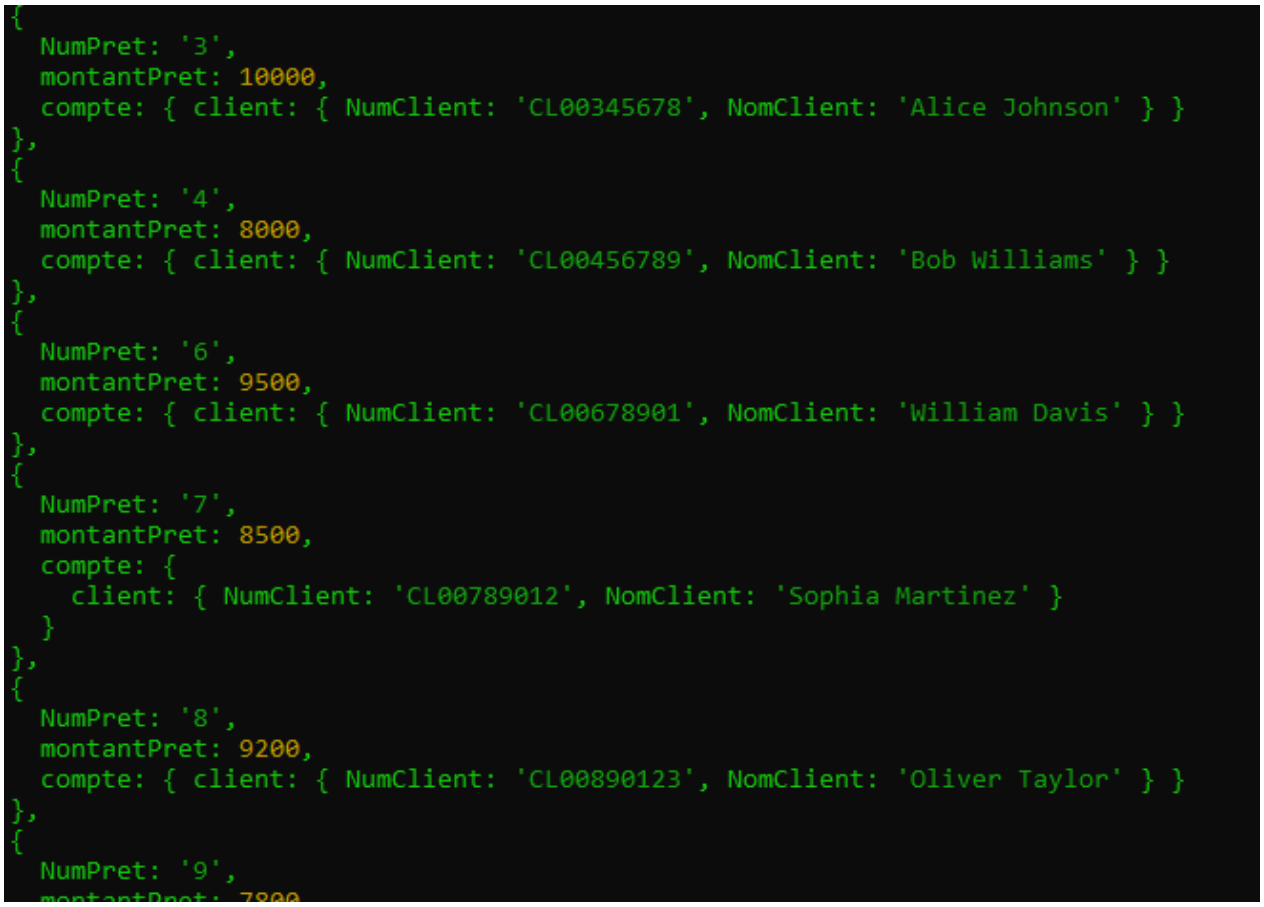


Figure 64: Affichage du résultat

8.6.Augmentation des échéances des prêts non soldés antérieurs à janvier 2021

Cette requête utilise la méthode updateMany() pour mettre à jour les documents de la collection "pret" où la date d'effet est antérieure à janvier 2021 et le montant de l'échéance n'est pas nul. Elle utilise l'opérateur $inc pour augmenter le montant des échéances de 2000.

```
const operatio6 = db.pret.updateMany(
    { "dateEffet": { $lt: "2021-01-01" }, "montantEcheance": { $ne:
       0 } },
    { $inc: { "montantEcheance": 2000 } }
  );
```

Figure 65: Mise à jour montant Echeance

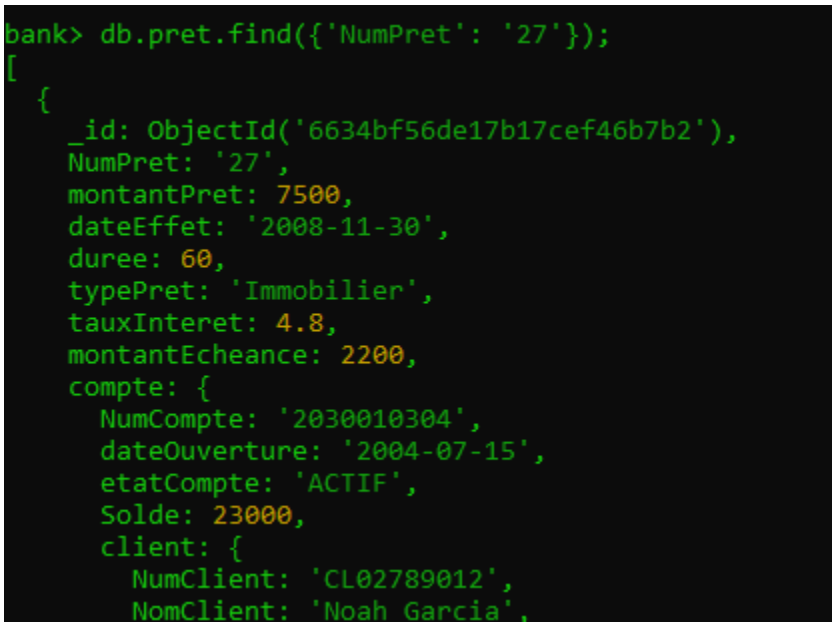


Figure 66: Avant mise à jour

Figure 67: Aprés mise à jour

8.7.Requête 3 avec le paradigme Map-Reduce

Cette requête utilise l'opération d'agrégation aggregate() pour regrouper les prêts par le numéro de l'agence et calculer le nombre total de prêts par agence. Ensuite, elle trie les résultats par ordre décroissant et les écrit dans une nouvelle collection appelée "Agence-NbPrêts".

```
// Definition de la fonction de mappage
var mapFunction = function() {
  emit(this.compte.agence.NumAgence, 1);
};

// Definition de la fonction de reduction
var reduceFunction = function(key, values) {
  return Array.sum(values);
};
```

Figure 68: Code du map-reduce

```
// Execution de l'operation Map-Reduce
db.pret.mapReduce(
  mapFunction,
  reduceFunction,
  { out: "Agence-NbPrets2" }
);
```

Figure 69: execution du map-reduce

```
// Trier les resultats par ordre decroissant du nombre total de
   prets
db["Agence-NbPrets2"].find().sort({ value: -1 });

// Afficher le contenu de la collection
db["Agence-NbPrets2"].find();
//8.
```

Figure 70: triage des résultats

Figure 71: affichage des résultats

8.8. Possibilité de répondre à la requête sur les opérations de crédit des clients de type "Entreprise" en 2023

On commence par filtrer les documents dans la collection **pret** en fonction des critères suivants :

1. La nature de l'opération dans le champ **NatureOperation** dans le tableau **Operations** du champ **compte** doit être égale à "Retrait".
2. La date de l'opération dans le champ **DateOp** dans le tableau **Operations** du champ **compte** doit être comprise entre le 1er janvier 2023 (inclus) et le 1er janvier 2024 (non inclus).
3. Le type du client dans le champ **TypeClient** dans le champ **client** du champ **compte** doit être égal à "Entreprise".

Une fois ces conditions de filtrage appliquées, les documents correspondants sont sélectionnés pour l'agrégation ultérieure.(figure 72)

```
db.pret.aggregate(
  {

    $match: {
      "compte.Operations.NatureOperation": "Retrait",
      "compte.Operations.DateOp": { $gte: "2023-01-01", $lt: "2024-01-01" },
      "compte.client.TypeClient": "Entreprise"
    },


  });
```

Figure 72: Agrégation pour trouver les opérations de crédit(retrait)

```
bank> db.pret.aggregate(
...   {
...
...       $match: {
...         "compte.Operations.NatureOperation": "Retrait",
...         "compte.Operations.DateOp": { $gte: "2023-01-01", $lt: "2024-01-01" },
...         "compte.client.TypeClient": "Entreprise"
...       },
...
...
...
...   });
[
  {
    _id: ObjectId('663d48689e3f89c5426874ac'),
    NumPret: '59',
    montantPret: 6500,
    dateEffet: '2011-02-15',
    duree: 48,
    typePret: 'ANSEJ',
    tauxInteret: 6,
    montantEcheance: 200,
    compte: {
      NumCompte: '5030011111',
      dateOuverture: '2000-08-10',
      etatCompte: 'ACTIF',
      Solde: 15000,
bank>
      NumClient: 'CL234900',
      NomClient: 'Hamza Hamzaoui',
      TypeClient: 'Entreprise',
      AdresseClient: '456 Elm St',
      NumTel: '+9876543210',
      Email: 'bob@example.com'
    },
```

Figure 73: exécution de l'agrégation

9. Analyse

Après avoir effectué les opérations de ce chapitre, il devient apparent que cette modélisation n'est optimale que lorsqu'on manipule les attributs directement liés au prêt. Cependant, les imbrications multiples qui constituent cette collection rendent l'accès aux autres "types" très difficile. Par conséquent, une telle modélisation n'est pas réaliste pour la gestion efficace des données d'une banque.

Deux alternatives peuvent être explorées : la première consiste à recentrer le modèle vers un élément plus central. Dans notre cas, une réorientation vers le compte semble plus judicieuse, car c'est vraisemblablement l'élément le plus peuplé de documents et possède le plus d'associations avec les autres types. Cela produira un document avec un niveau de profondeur d'un (une seule imbrication), comme illustré dans la figure 73.


```
"compte": {
    "NumCompte": "5030011502",
    "dateOuverture": "2000-08-10",
    "etatCompte": "ACTIF",
    "Solde": 15000,
    "client": {
      "NumClient": "CL234567",
      "NomClient": "Bob Bobson",
      "TypeClient": "Entreprise",
      "AdresseClient": "456 Elm St",
      "NumTel": "+9876543210",
      "Email": "bob@example.com",
      "Operations": [
        {
          "NumOperation": "OP202",
          "NatureOperation": "Retrait",
          "montantOp": 200,
          "DateOp": "2001-02-16",
          "Observation": "Withdraw from account"
        }
      ]
      "Prets": [
        "NumPret": "36",
    "montantPret": 6500,
    "dateEffet": "2011-02-15",
    "duree": 48,
    "typePret": "ANSEJ",
    "tauxInteret": 6,
    "montantEcheance": 200
      ]
      "agence": {
        "NumAgence": "503",
        "nomAgence": "Pine Street Branch",
        "adresseAgence": "789 Pine St, Village",
        "categorie": "PRINCIPALE",
        "succursale": {
          "NumSucc": "5",
          "nomSucc": "VillageBank-Branch-capital",
          "adresseSucc": "789 Pine St, Village",
          "region": "OUEST"
        }}
}
  }
```


Figure 74: Modelisation centrée vers le client

Une autre alternative serait de s'inspirer du modèle relationnel et de diviser le document orienté compte en plusieurs collections : une collection centrale pour les comptes, une collection pour les opérations, etc. Toutes ces collections possèdent des identifiants des autres collections liées (comme vu dans le modèle objet-relationnel).

Les documents d'une collection font référence aux documents d'une autre collection en utilisant des identifiants ou des clés étrangères. Par exemple, une collection de commentaires peut contenir des documents faisant référence au champ **_id** des documents d'une collection de publications. Cette approche est adaptée aux relations de type plusieurs-à-plusieurs ou un-à-plusieurs où les documents enfants ne sont pas directement liés au document parent.

10. Conclusion

Ce rapport a abordé deux aspects principaux de la gestion de bases de données : le modèle relationnel-objet et le modèle orienté documents. Dans la première partie, nous avons détaillé la modélisation orientée objet en transformant un schéma relationnel en un diagramme de classes UML. Nous avons ensuite créé les TableSpaces et l'utilisateur requis pour la base de données, défini les types abstraits et les associations nécessaires, ainsi que les méthodes pour effectuer divers calculs et requêtes.

Dans la seconde partie, nous avons exploré le modèle orienté documents en proposant une modélisation alternative de la base de données décrite précédemment. Nous avons justifié nos choix de conception, présenté les avantages et inconvénients de cette approche, puis répondu à diverses requêtes en utilisant MongoDB et en comparant certains résultats avec ceux obtenus dans la première partie.

L'approche relationnelle-objet offre une représentation plus proche du monde réel et facilite la modélisation des associations entre entités. Cependant, elle peut être complexe à gérer et nécessite des opérations de jointure coûteuses en cas de données volumineuses. En revanche, le modèle orienté documents est plus flexible, évolutif et performant pour les opérations de lecture et d'écriture massives, grâce à sa capacité de stockage hiérarchique et dénormalisée des données.

En examinant les requêtes effectuées dans les deux approches, nous constatons que chaque modèle a ses propres forces et faiblesses. Le modèle relationnel-objet est plus adapté aux opérations de jointure complexes et aux requêtes transactionnelles, tandis que le modèle orienté documents excelle dans la gestion de données non structurées et dans les cas d'utilisation nécessitant une évolutivité rapide et une distribution géographique.

Bien que le modèle orienté documents offre des avantages significatifs en termes de scalabilité et de flexibilité, il est important de reconnaître que les deux approches ont leur place dans le paysage des bases de données. Il est possible de tirer parti des principes relationnels pour simplifier certaines manipulations de données, tout en profitant des capacités de stockage et de requête avancées offertes par les bases de données NoSQL orientées documents. En fin de compte, le choix du modèle dépend des besoins spécifiques de l'application et des contraintes de performance auxquelles elle est confrontée.