
NF18 - Rapport

Sujet 3 : Gestion de comptes bancaires



Sommaire

Introduction	2
Phase 1 : Relationnel	3
Conception de l'UML	3
Passage au Modèle Logique de Données (MLD)	4
Classes	4
Types	5
Contraintes	5
Étude de normalisation	6
Phase II : Non relationnel	9
Création de tables	9
Insertion de données	10
Requêtes	11
Comparaison relationnel / non relationnel	14
Conclusion	15

Introduction

Dans le cadre de l'UV NF18, nous avons réalisé un projet nous faisant utiliser les notions sur les bases de données apprises au cours du semestre. Ces notions concernent le modèle conceptuel de données (MCD), en UML, le modèle logique de données (MLD), en algèbre relationnelle et l'implémentation, en SQL. Nous avons dans un premier temps fait une implémentation en relationnel avec PostgreSQL, puis en non relationnel, toujours avec PostgreSQL.

En ce qui concerne le sujet du projet en lui-même, nous devons réaliser un système de gestion de comptes bancaires. Le sujet présentait un certain nombre de contraintes et de spécifications, en voici l'énoncé :

Sujet 3 : Gestion de comptes bancaires

Son système d'information étant obsolète, une banque vous demande de l'aider à concevoir une base de données permettant de gérer ses clients, les comptes dont ils sont titulaires et les opérations effectuées sur ces comptes. Le compte rendu d'enquête ci-dessous vous indique les traitements effectués par la banque et les données dont elle a besoin pour ce faire. A l'arrivée d'un nouveau client, la banque enregistre son nom, son téléphone (qui est unique) et son adresse. Un client peut être titulaire de plusieurs comptes et un compte peut avoir plusieurs titulaires (mais au moins un) susceptibles d'y appliquer des opérations. Il existe trois types de comptes : les comptes courant, les comptes de revolving et les comptes d'épargne. Un compte courant est un compte sur lequel on peut déposer ou retirer de l'argent via différentes catégories d'opérations. Un compte revolving est un compte particulier correspondant à un crédit accordé par la banque à son titulaire. Ainsi un compte de revolving a toujours un solde négatif ou nul, à partir duquel sont calculés des intérêts payés par le titulaire du compte. Un compte d'épargne est destiné à favoriser l'épargne de son titulaire. Son solde doit toujours être positif et supérieur à un montant fixé de façon statutaire : 300€. Les clients peuvent réaliser tout type d'opérations sur les comptes courant et les comptes de revolving : opération au guichet (débit, crédit), opération de dépôt ou d'émission de chèque, opération de carte bleu, virement. Par contre les comptes d'épargne sont limités aux opérations au guichet et aux virements. Pour tout compte, on enregistre son type, sa date de création (unique), la balance actuelle du compte et son statut (ouvert, bloqué, fermé). On ne peut faire aucune opération sur un compte fermé et on ne peut faire que des opérations de crédit ou des opérations au guichet sur un compte bloqué. Un compte courant est un compte pour lequel on enregistre, en plus, le montant du découvert autorisé et la date depuis laquelle le compte est à découvert (si c'est le cas). Afin de faciliter les prises de décision, on maintient aussi la valeur minimum et la valeur maximum du solde de chaque compte courant, mois par mois. Un compte revolving fonctionne comme un compte, mais avec un solde négatif comme mentionné ci-dessus. A l'ouverture du compte de revolving, un montant minimum est négocié avec le client, ainsi qu'un taux d'intérêt journalier appliqué chaque jour en fonction du solde du compte. A chaque opération on répertorie le compte concerné, le client auteur de l'opération, le type de l'opération, le montant, la date et l'état de l'opération. Le type de l'opération peut être un débit ou un crédit effectué au guichet, un dépôt de chèque, une émission de chèque ou un débit en carte bleu. Une opération de débit ou de crédit effectuée au guichet ne nécessite aucune information spécifique. Par contre elle n'est enregistrée que si on a pu débiter ou créditer le compte correspondant et lui affecter l'état traité. Une opération de débit carte bleue ne nécessite pas non plus d'information spécifique.

N.B: pour réaliser l'UML de ce projet, il est fortement recommandé d'utiliser plantuml accessible via ce lien: <https://www.planttext.com/>

Exemples de requêtes à réaliser:

- connaître le montant d'un compte connaissant sa date de création.
- compter le nombre de chèques émis par un client identifié par son numéro de téléphone.

Figure 1 : Enoncé du sujet

Ce rapport présentera notre démarche pour mener le projet à bien, les difficultés que nous avons rencontrées, les solutions que nous avons trouvées et une comparaison entre les deux implémentations.

Phase 1 : Relationnel

★ Conception de l'UML

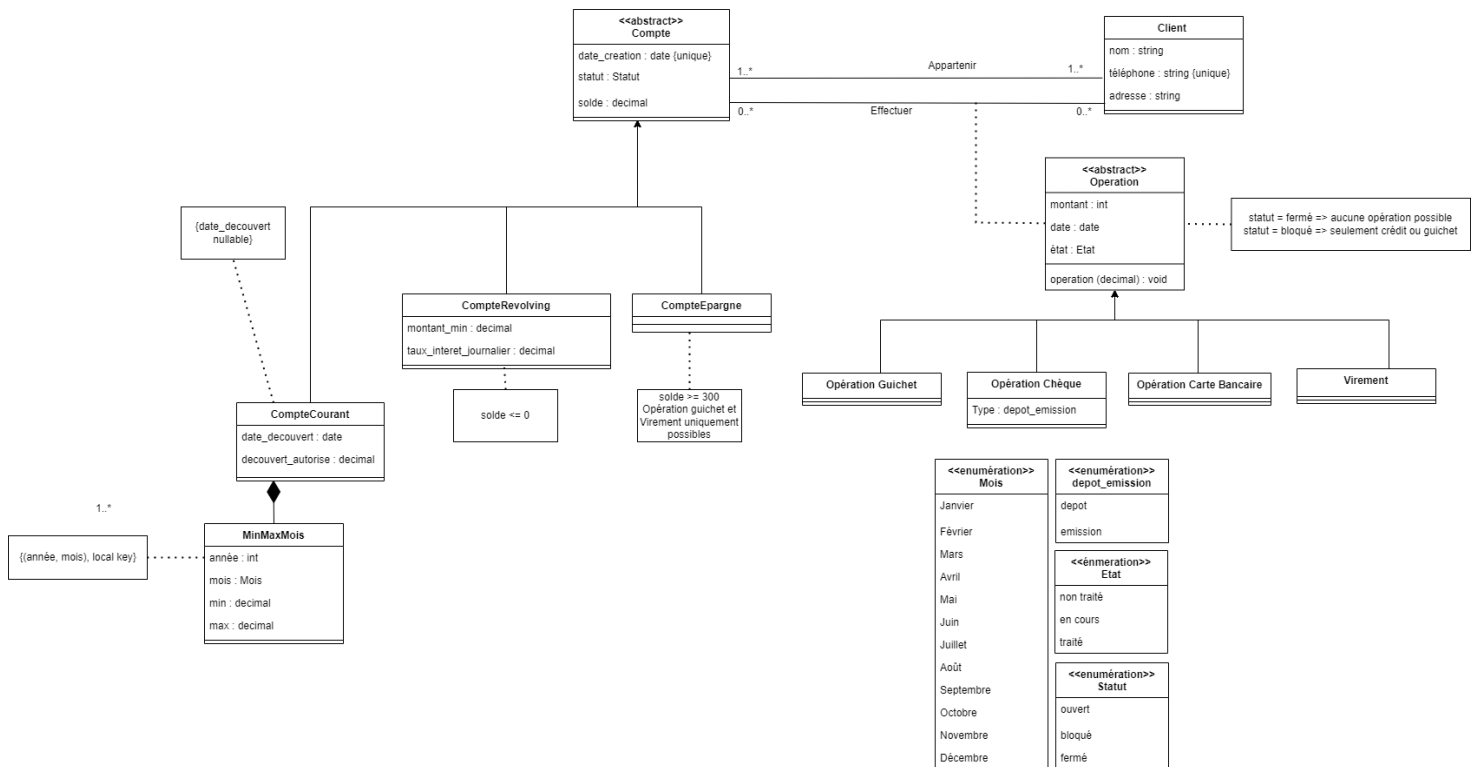


Figure 2 : UML du projet

Nous avons réalisé le MCD de notre projet afin de respecter les consignes données.

Un Compte peut être de 3 types différents :

- CompteCourant
- CompteRevolving
- CompteEpargne

Ces types de compte héritent donc de la classe Compte puisqu'ils en sont des spécifications.

Les comptes courant ont une spécificité qui est que le stockage des montants minimum et maximum autorisés du compte pour chaque mois. Nous avons donc introduit la table MinMaxMois qui compose CompteCourant. Chaque tuple va ainsi stocker les différents éléments voulus sur lesquels on pourra faire des vérifications lors du mois courant lorsqu'on voudra mettre à jour le solde avec une opération.

Un compte est lié à un client à la fois par sa possession, mais aussi par les opérations qu'il effectue dessus. La classe Association Opération est héritée par tous les différents types d'opérations. Elles ont toutes leurs contraintes, notamment par rapport aux comptes sur lesquelles elles peuvent être appliquées.

Ainsi, l'ensemble des informations et contraintes demandées peuvent être gérées. Le passage au MLD spécifiera ces dernières.

★ Passage au Modèle Logique de Données (MLD)

Afin de passer au modèle logique de données, il nous a fallu analyser les différents cas possibles pour nous faciliter la tâche au niveau applicatif.

Pour l'héritage de la classe Compte, il nous a paru impossible de réaliser un héritage par classe fille puisque nous rencontrons des difficultés par la suite pour les relations Appartenir et Opération. En effet, il faudrait vérifier de nombreuses contraintes au niveau applicatif, ce que nous ne souhaitons pas. Un héritage par la classe mère ne nous a pas semblé envisageable car il y aurait à nouveau des contraintes complexes à vérifier, comme pour la composition avec MinMaxMois. L'héritage par référence nous a donc semblé la meilleure option, alliant performance et simplicité.

Pour l'héritage de la classe Compte, nous avons directement remarqué que l'on était dans le cas d'un héritage presque complet, dans lesquelles les classes filles ont peu d'attributs et ne sont liées dans aucune relation. L'héritage par classe mère nous a donc semblé être la meilleure option.

Nous avons alors créé les tables comme suit :

Classes

Client(#id : int, nom : str, telephone : str, adresse : str) avec {telephone key, (telephone, adresse) NOT NULL}

Compte(#date_creation : date, statut : Statut, solde : decimal) avec {statut, solde NOT NULL}

Appartenir(#compte => Compte.date_creation, #client => Client.id)

CompteCourant(#compte => Compte.date_creation, date_decouvert : date, decouvert_autorise : decimal) avec {decouvert_autorise NOT NULL}

CompteRevolving(#compte => Compte.date_creation, montant_min : decimal, taux_interet_journalier : decimal) avec {(montant_min, taux_interet_journalier) NOT NULL}

CompteEpargne(#compte => Compte.date_creation)

MinMaxMois (#année : int, #mois : Mois, min : int, max : int, #compte => Compte.date_creation) avec {(min, max) NOT NULL}

Opération(#Client=>Client.id, #Compte=>Compte.date_creation, montant : int, #date : date, état : Etat, type_operation : TypeOpération, TypeChèque : DepotEmission) avec {(Client, montant, état, TypeOpération) NOT NULL}

Types

DepotEmission : enum{depot, emission}

TypeOpération : enum{Guichet, Chèque, CB, Virement}

Statut : enum{ouvert, fermé, bloqué}

Mois : enum{'janvier', 'février', 'mars', 'avril', 'mai', 'juin', 'juillet', 'août', 'septembre', 'octobre', 'novembre' et 'décembre'}

Etat : enum{'traité', 'non traité', 'en cours'}

Contraintes

- Opération :

NOT (TypeChèque != NULL AND TypeOpération IN {Guichet, CB , Virement})

NOT (Compte.Statut = fermé)

NOT (Compte.Statut = bloqué AND TypeOpération IN {CB, Chèque})

NOT (montant < 0 AND TypeChèque = {dépôt})

NOT (montant > 0 AND TypeChèque = {émission})

NOT (montant > 0 AND TypeOpération IN {Virement, CB})

- Comptes :

Compte épargne :

solde > 300 :

Projection(Jointure(Restiction(Compte,solde >=300),CompteEpargne,Compte.date_creation = CompteEpargne.compte), Compte.date_creation) = Projection(CompteEpargne, compte)

{seules des opérations au guichet et virements peuvent faits avec un compte épargne}

Compte revolving :

solde < 0

Projection(Jointure(Restiction(Compte, solde < 0),CompteRevolving,Compte.date_creation = CompteRevolving.compte), Compte.date_creation) = Projection(CompteRevolving, compte)

Classe mère abstraite :

Projection(Compte, id) = Union(Projection(CompteEpargne, compte), (Union(Projection(CompteRevolving, compte), Projection(CompteCourant, compte))))

Héritage exclusif :

Intersection(Projection(Comptecourant, compte), Intersection(Projection(CompteRevolving, compte), Projection(CompteEpargne, compte)))= {}

Pour la composition :

Projection(CompteCourant, compte) = Projection(MinMaxMois, compte)

- **Appartenir :**

Tous les clients ont au moins un client et un client possède au moins un compte :

$\text{Projection}(\text{Restriction}(\text{Appartenir}, \text{compte})) = \text{Projection}(\text{Compte}, \text{date_creation})$
 $\text{Projection}(\text{Restriction}(\text{Appartenir}, \text{client})) = \text{Projection}(\text{Client}, \text{id})$

★ Étude de normalisation

Nous avons ensuite étudié la normalisation fonctionnelle de notre modèle logique initial.

Rappel :

- 1 NF : Attribut atomique et clé
- 2 NF : Tout attribut non clé n'est pas déterminé par un sous ensemble de la clé
- 3 NF : Tout attribut non clé n'est pas déterminé par un attribut non clé
- BCNF : Tout attribut clé n'est pas déterminé par un attribut non clé

Etude de la relation Client :

On a : **Client**(#telephone, nom, adresse) avec {(nom, adresse) NOT NULL}

Dépendances fonctionnelles élémentaires :

- telephone \rightarrow nom
- telephone \rightarrow adresse

Niveau de normalisation fonctionnelle :

BCNF : toutes les dépendances fonctionnelles sont de la forme $K \rightarrow A$ avec K un attribut clé.

On procède de manière analogue pour les autres tables, et on tire les conclusions suivantes.

La base de données est en BCNF.

En effet :

- Toutes les tables sont en 1NF car elles comportent une clé et les attributs sont atomiques.
- Toutes les tables sont en 2NF parce que tous les attributs non clé sont déterminés par tous les attributs clé et non une partie seulement des attributs clé.
- Toutes les tables sont en 3NF puisqu'aucun attribut non clé ne détermine un autre attribut non clé
- Toutes les tables sont en BCNF puisque les dépendances fonctionnelles sont de la forme $K \rightarrow A$, avec K une clé.

★ Création et interrogation des tables

Le passage au MLD étant effectué, la partie principale du travail s'avérait déjà effectuée. Il nous a juste fallu transcrire nos idées en langage SQL.

Par exemple, pour la table Client, nous avons alors le code suivant :

```
CREATE TABLE Client(  
  id serial primary key,  
  nom varchar(30) not null,  
  telephone varchar(10) unique not null,  
  adresse varchar(50) not null  
);
```

NB : Les autres requêtes de création de tables sont disponibles sur le Gitlab, dans Partie 1/Rendu 3/ TABLE.SQL

Nous avons procédé de manière analogue pour les différentes tables.

En ce qui concerne l'interrogation de la base de données, nous avons conçu les requêtes demandées dans l'énoncé ainsi que des requêtes supplémentaires qui nous faciliteront le travail par la suite lors de la partie applicative. En utilisant les connaissances établies en cours, nous avons alors écrit les requêtes suivantes en SQL :

```
--Connaître le montant d'un compte connaissant sa date de création :  
SELECT solde FROM Compte WHERE Compte = date_de_création
```

```
--Compter le nombre de chèques émis par un client identifié par son  
numéro de téléphone :  
SELECT COUNT(*) FROM Opération INNER JOIN Client ON Operation.client =  
Client.id WHERE Client.telephone = TELEPHONE AND Opération.TypeOperation  
= 'Cheque' AND Opération.TypeChèque = 'emission'
```

NB : D'autres requêtes SQL sont disponibles sur le Gitlab, dans Partie 1/Rendu 3/ INTERROGATION.SQL

★ Application Python

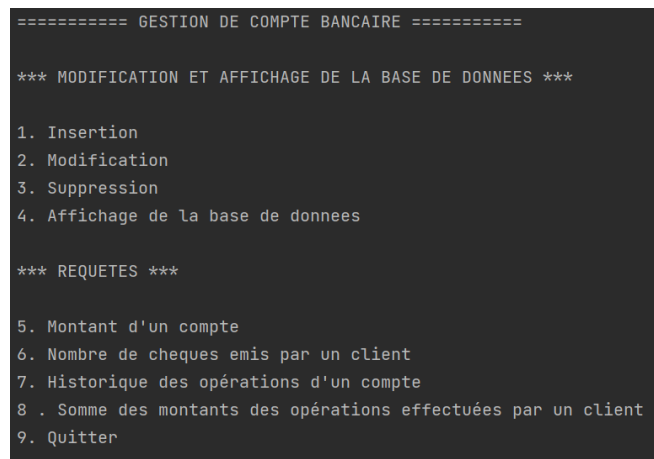
Le dernier temps de la phase 1 était consacré à la conception d'une application Python permettant de gérer les comptes bancaires. Cette application devait permettre d'ajouter des clients, des comptes de différents types, des opérations, permettre la modification des données ou leur suppression.

Afin d'améliorer la lisibilité du code, nous avons choisi de séparer le code en plusieurs fichiers. Par exemple, le fichier affichage.py se chargera de l'affichage en console, tandis que insertion.py se chargera des ajouts dans la base de données.

Le fichier `affichage.py` permet donc l'affichage à l'écran comme suit :

```
def afficheMenuPrincipal():  
    """Fonction qui affiche le menu principal"""  
    print("===== GESTION DE COMPTE BANCAIRE =====\n")  
    print("***** MODIFICATION ET AFFICHAGE DE LA BASE DE DONNEES ***\n"  
1. Insertion\n2. Modification\n3. Suppression\n4. Affichage de la base  
de donnees\n\n*** REQUETES ***\n  
5. Montant d'un compte\n6. Nombre de cheques emis par un client\n7.  
Historique des opérations d'un compte\n8 . Somme des montants des  
opérations effectuées par un client\n9. Quitter\n\n")
```

Ce qui donne le rendu suivant :



```
===== GESTION DE COMPTE BANCAIRE =====  
  
*** MODIFICATION ET AFFICHAGE DE LA BASE DE DONNEES ***  
  
1. Insertion  
2. Modification  
3. Suppression  
4. Affichage de la base de donnees  
  
*** REQUETES ***  
  
5. Montant d'un compte  
6. Nombre de cheques emis par un client  
7. Historique des opérations d'un compte  
8 . Somme des montants des opérations effectuées par un client  
9. Quitter
```

[Figure 3](#) : Application en console

Le fichier `application.py` interagit avec l'utilisateur, c'est le programme principal qui est exécuté. Il demande le choix de l'utilisateur et permet à l'utilisateur d'accomplir les tâches qu'il souhaite effectuer.

Les autres fichiers permettent quant à eux l'insertion, la suppression et la modification des données. Ainsi, nous avons conçu une application en console.

Phase II : Non relationnel

Nous avons effectué le passage au non relationnel en utilisant du relationnel-JSON avec PostgreSQL.

★ Création de tables

Nous avons rassemblé toutes les informations dans une unique table compte de la forme suivante :

```
CREATE TABLE Compte(  
  date_creation date primary key,  
  statut Statut not null,  
  solde decimal not null,  
  proprietaires JSON,  
  type_compte JSON  
);
```

L'attribut JSON proprietaires contiennent les informations suivantes :

```
{  
  "id" :  
  "nom" :  
  "prenom" :  
  "operations" :  
  {  
    "montant" :  
    "date" :  
    "etat" :  
    "typeOperation" :  
  }  
}
```

On y stocke l'ensemble des informations du client ainsi que toutes les opérations qu'il a effectué dans l'attribut JSON "operations", qui ne seront donc pas amenés à évoluer au fur et à mesure du temps, ce qui n'est pas adapté à un usage bancaire comme vu en cours.

L'attribut type_compte comporte les informations suivantes :

```
{  
  "type" :  
  "decouvert_autorise" :  
  "MinMaxMois" :  
  {
```

```

        "annee" :
        "mois" :
        "min" :
        "max" :
    }
}

```

On y stocke l'ensemble des informations liées au type de compte. `decouvert_autorise` et le sous attribut JSON `MinMaxMois` ne seront renseignés que pour un type de compte `Courant`.

En d'autres termes, en non relationnel, un compte possède un ensemble de propriétaires, qui effectuent des opérations sur le compte. Egalement, on peut tout de suite s'apercevoir d'un problème au niveau de la cohérence des données. En effet, on peut avoir plusieurs fois un propriétaire avec l'id 3 et comme nom "Dupont" ou "Durant" à différents endroits de la base de données.

★ Insertion de données

Voici un exemple d'insertion de données :

```

INSERT INTO Compte VALUES (
'2022-05-03',
'ouvert',
-650,
'[
    {
        "id" : 4,
        "nom" : "Garcia",
        "prenom" : "Fanny",
        "operations" :
        [
            {
                "montant" : 100,
                "date" : "2022-05-12",
                "etat" : "traite",
                "typeOperation" : "Virement"
            },
            {
                "montant" : 150,
                "date" : "2022-05-13",
                "etat" : "traite",
                "typeOperation" : "Guichet"
            }
        ]
    }
],

```

```

{
  "id" : 3,
  "nom" : "Jean",
  "prenom" : "Nemar",
  "operations" :
  [
    {
      "montant" : 600,
      "date" : "2022-05-18",
      "etat" : "traite",
      "typeOperation" : "Cheque",
      "typeCheque" : "Emission"
    }
  ]
},
'{'
  "type" : "CompteRevolving",
  "montant_min" : -1500,
  "taux_interet_journalier" : 0.6
}'
)

```

Il est à noter que le modèle relationnel-JSON ne vérifie pas la cohérence des données au sein d'un attribut JSON. Ainsi, on peut avoir un même identifiant de propriétaire ayant deux noms différents dans deux fichiers JSON.

★ Requêtes

Comme lors de la réalisation du modèle relationnel, il nous a fallu concevoir des requêtes pour accéder aux données. Nous avons donc créé les requêtes suivantes, selon les normes établies en cours magistraux :

```

--Connaître les solde de tous les comptes de la base de données
SELECT date_creation, solde
FROM Compte;

```

```

--Connaître le statut d'un compte créé à une date donnée (date_voulue)
SELECT statut
FROM Compte
WHERE date_creation = date_voulue;

```

```

-- Nombre de comptes dans la base de données

```

```

SELECT Count(*)
FROM Compte;

-- Nombre de compte ayant un solde supérieur à un seuil
SELECT COUNT(*)
FROM Compte
WHERE solde > seuil;

-- Nombre de compte ayant un solde inférieur à un seuil
SELECT COUNT(*)
FROM Compte
WHERE solde < seuil;

-- Nombre de compte créé après une date donnée
SELECT COUNT(*)
FROM Compte
WHERE date_creation > date_voulue;

-- Nombre de compte ayant le statut ouvert
SELECT COUNT(*)
FROM Compte
WHERE statut = 'ouvert';

-- Nombre de compte ayant le statut ouvert
SELECT COUNT(*)
FROM Compte
WHERE statut = 'ferme';

-- Nombre de compte ayant le statut ouvert
SELECT COUNT(*)
FROM Compte
WHERE statut = 'bloque';

-- Nombre de personne ayant un compte créé avant une certaine date
SELECT COUNT(*)
FROM Compte
WHERE date_creation < date_voulue;

-- Comptes bloqués depuis une date donnée
SELECT COUNT(*)
FROM Compte
WHERE statut = 'bloque' AND solde < seuil;

```

```
--Connaître la somme totale des montants effectués par un client id par
numtel en fonction d'un type d'opération
SELECT SUM(proprietaires->operations->>montant)
FROM Compte C, JSON_ARRAY_ELEMENTS(C.proprietaires) proprietaires
WHERE date_creation = date AND proprietaires->operations->>typeOperation
= type_voulu;
```

```
--Connaître l'opération effectuée et le montant de celle-ci par un
client à une date précise
```

```
SELECT                                proprietaires->operations->>'montant',
proprietaires->operations->>'typeOperation'
FROM Compte C, JSON_ARRAY_ELEMENTS(C.proprietaires) proprietaires
WHERE C.date_creation = date
AND proprietaires->operations->>date = date_voulue;
```

```
--Historique des opérations réalisées par un client
```

```
SELECT props->>'operations'
FROM Compte C, JSON_ARRAY_ELEMENTS(C.proprietaires) props
WHERE props->>id = id_voulu;
```

```
--Nom et prénom des propriétaires des comptes
```

```
SELECT date_creation, props->>'nom', props->>'prenom'
FROM Compte C, JSON_ARRAY_ELEMENTS(C.proprietaires) props;
```

Ces requêtes permettent d'accéder aux informations contenues dans les attributs de type JSON. Nous avons utilisé “->” pour nous déplacer à l'intérieur de l'arborescence d'un fichier JSON, et “->>” pour accéder à la valeur finale d'un attribut qui n'est pas de type JSON. Egalement, nous avons utilisé les agrégats et fonctions SQL afin d'améliorer la qualité de nos requêtes.

Comparaison relationnel / non relationnel

En ce qui concerne le relationnel, celui-ci dispose de plusieurs tables liées entre elles et disposant de contraintes. Le système est bien ordonné, il n'a pas de redondance et permet une vérification simple de la validité des données. Les mises à jour sont également simples à effectuer. Une critique que nous pouvons faire des contraintes imposées par le sujet est le fait que la date de création de compte est unique, ce qui limite la banque à ne pouvoir créer qu'un seul compte par jour.

Pour la partie non relationnelle, elle ne contient qu'une seule table contenant elle-même les autres tables. Le non relationnel n'est pas conçu pour ce type d'application bancaire, et ce, pour de multiples raisons. Le non relationnel ne nous permet pas d'assurer la cohérence des données et d'assurer le bon déroulement des transactions. On doit rentrer les opérations futures d'un client au moment de la création du compte, la consigne n'est donc pas fidèle à la réalité. Une autre raison est la moins grande fiabilité des données : il peut y avoir de la redondance et des données incorrectes ou incomplètes, en effet, il n'y a pas ou peu de vérification au niveau de la base de données. S'il doit y avoir des vérifications, elles doivent se faire au niveau applicatif, ce qui diminue la fiabilité des données. Cependant, un avantage du non relationnel est qu'il est plus rapide étant donné qu'il n'y a pas de jointure à faire pour accéder aux données.

Ces différentes analyses nous ont amené à réaliser le tableau synthétique suivant des avantages et inconvénient des deux différents modèles :

	Avantages	Inconvénients
Relationnel	<ul style="list-style-type: none">- Cohérence des données- Vérification des données- Fiabilité des données	<ul style="list-style-type: none">- Temps important pour les requêtes (jointures)
Non relationnel	<ul style="list-style-type: none">- Rapidité des requêtes- Souplesse du modèle	<ul style="list-style-type: none">- Données redondantes- Vérification des données au niveau applicatif

[Figure 4](#) : Avantages et inconvénients des modèles

Conclusion

Ce projet nous a permis de mettre en pratique les connaissances acquises lors des cours magistraux et travaux dirigés. Il nous a permis de concevoir intégralement une application bancaire liée à une base de données. À travers ce projet, nous avons appris à analyser de manière rigoureuse l'énoncé afin de prendre en compte les différentes contraintes et demandes imposées.

La réalisation d'un modèle en relationnel et d'un autre en non relationnel nous a permis de comprendre les avantages et inconvénients de ces différents modèles. Le modèle relationnel nous a semblé plus adapté pour la gestion de comptes bancaires. En effet, il permet la fiabilité et la cohérence des données.