TAHIRI Mohamed
KEBLI Soulaymane
CHAPIRO Louka
YU Jieni

utc
Université de Technologie
Compiègne

# Rapport de projet
# NF18

## Introduction :

L'objectif du projet de groupe de NF18 est de réussir à développer une base de données ainsi que son environnement applicatif dans un contexte concret. En effet, nous avons reçu un cahier des charges que nous avons dû suivre pour la mise en place de notre travail. Notre sujet concerne une société de chemin de fer qui a besoin de gérer des trains, des lignes, des réservations ou des comptes clients avec différents types d'utilisateurs.

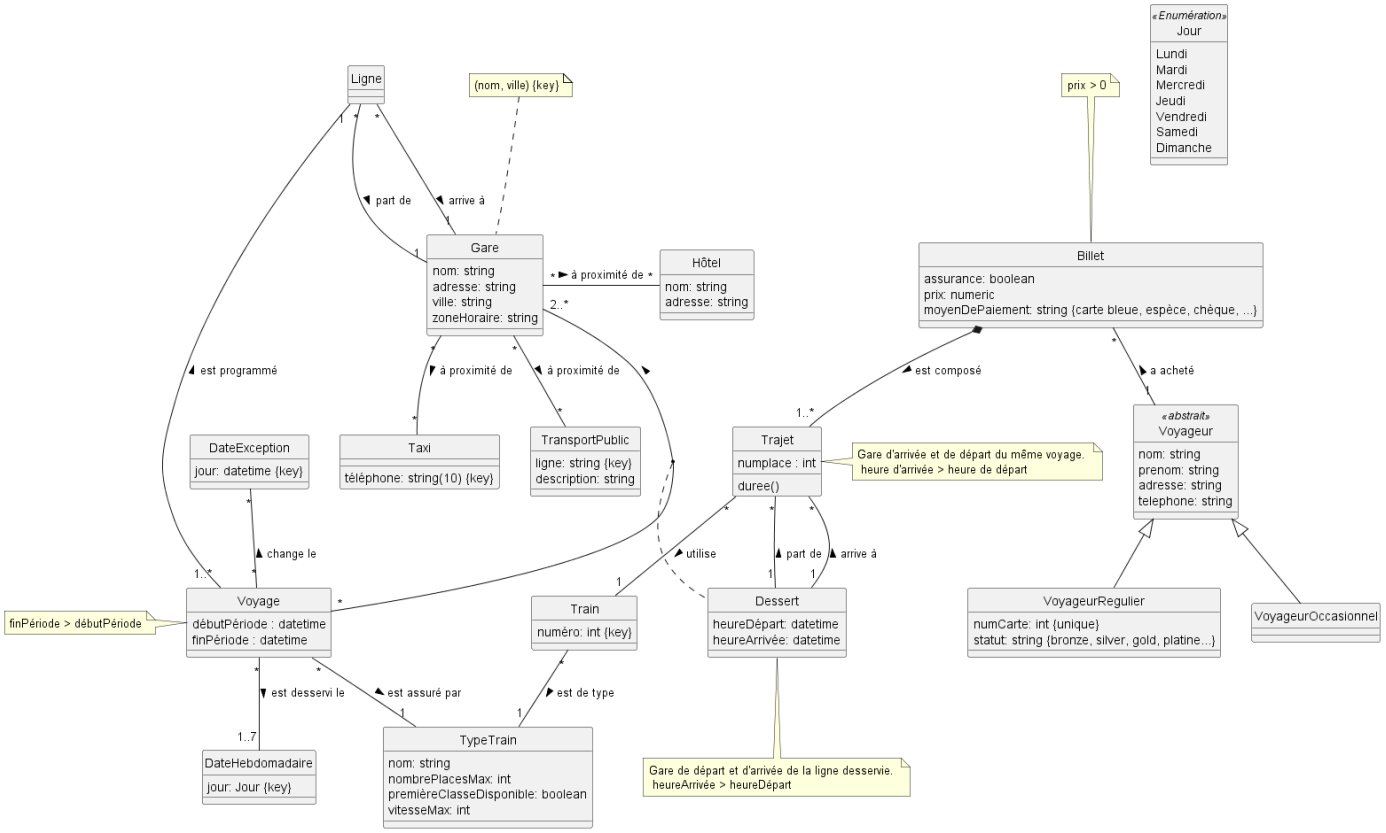
Notre rôle est de répondre aux besoins et contraintes du client à l'aide des notions que nous avons étudiées dans le cadre de l'uv. Cela comprend dans un premier temps la mise en place d'un modèle relationnel, comprenant la création du modèle conceptuel puis logique de la base de données, que nous adapterons ensuite en SQL, pour enfin développer une application python utilisable par le client.

Dans un second temps, nous avons transformé notre base de données d'origine en modèle non relationnel ce qui implique la modification de la structure de notre travail.
Ce rapport a pour but d'expliciter notre démarche à chaque étape du projet, de justifier les choix d'implémentation que nous avons réalisés et de synthétiser notre travail. De nombreuses figures viendront illustrer nos propos pour permettre une meilleure compréhension des sujets abordés.

# Sommaire :

# Partie I : Le modèle relationnel

## Le modèle conceptuel de données




La principale difficulté que nous avons rencontrée lors de la réalisation de notre modèle conceptuel est l'ambiguïté entre certaines notions du cahier des charges. En effet, il est important de bien définir les nuances entre une Ligne, un Voyage et un Trajet afin d'avoir une modélisation qui correspond à la situation décrite.

Une ligne est composée d'une gare de départ et d'une gare d'arrivée. Elle n'est définie que par ces dernières. Par exemple, dans un contexte réel on pourrait retrouver la ligne Paris-Marseille. Cependant, la ligne ne donne aucune indication sur les gares desservies entre le départ et le terminus.

C'est le rôle d'un voyage d'indiquer ces gares-là. Un voyage fait partie d'une ligne et est fait d'une succession de gares. Il est obligatoire qu'il contienne la gare de départ et d'arrivée de la ligne comme indiqué dans la contrainte de la relation Dessert. Le voyage contient les horaires. Pour reprendre l'exemple de Paris-Marseille, un voyage sur cette ligne pourrait être Paris-Lyon-Marseille ou bien Paris-Grenoble-Lyon-Marseille ou plus simplement Paris-Marseille si le train est direct.

Un trajet quant à lui représente le parcours d'un voyageur. Le trajet récupère une gare d'arrivée et une gare de départ parmi toutes les gares d'un même voyage. Cette condition est importante sur la table trajet. Un trajet est forcément lié à un billet et un

billet à un voyageur. Ainsi un trajet n'est pas prédéfini par un administrateur de la base de données, mais est créé par le client selon son besoin. Pour finir l'exemple que nous avions commencé, un trajet pourrait être Grenoble-Lyon du voyage Paris-Grenoble-Lyon-Marseille de la ligne Paris-Marseille.
Voici les trois éléments clés de la modélisation de notre base de données. Ensuite nous avons décidé de représenter la relation voyageur par un héritage. En effet, cela s'y porte particulièrement, car il existe deux types de voyageurs différents, les voyageurs occasionnels et réguliers avec comme seule différence une carte numérotée et un statut pour les derniers. Il s'agit donc d'un héritage exclusif.

Enfin, le MCD que nous avons mis au point contient de nombreuses contraintes complexes qu'il sera nécessaire de retranscrire dans le passage au modèle logique afin de garder la cohérence dans les données.

## Le modèle logique de données

L'idée de ce rapport n'est pas de revenir sur chaque détail des rendus précédents, c'est pour cela que dans cette partie nous allons nous focaliser sur les éléments les plus importants du passage du MCD au MLD.

Le premier est la relation d'héritage concernant les voyageurs.

### Définition d'un voyageur

> Pour l'héritage avec Voyageur, on décide de faire un héritage par classe Mère avec un attribut type.

- **Voyageur** (#id : int, nom : str, prénom : str, adresse : str, téléphone : str, type : str, num_carte : int, statut : str)

> **Avec :**
>
> - type = {Régulier, Occasionnel),
> - Statut = {bronze, silver, gold, platine} et NULLABLE,
> - UNIQUE (num_carte) et NULLABLE,
> - NOT (type = Occasionnel AND num_carte AND statut),
> - NOT (type = Régulier AND num_carte = NULL AND statut = NULL).

Nous avons choisi de faire une transformation par classe mère. Il s'agit dans ce cas de la meilleure solution, car l'héritage est quasiment complet : VoyageurOccasionnel ne contient aucun attribut et VoyageurRégulier seulement deux. Les classes filles n'ont également aucune relation ce qui permet de procéder facilement par héritage par classe mère. Étant donné que l'héritage est exclusif, il est nécessaire d'ajouter certaines contraintes.

D’abord il faut ajouter un attribut type qui indique si le voyageur est régulier ou occasionnel. Ensuite il faut vérifier que selon son type il ait ou non les attributs réservés aux voyageurs réguliers.

Nous avons décidé ensuite de nous intéresser aux tables Trajet et Dessert.

## Définition d'un trajet

- **Trajet**(#id : int, #billet => Billet.id, num_place : int, gare_départ => Dessert.gare, heure_départ => Dessert.heure_départ, gare_arrivée => Dessert.gare, heure_arrivée => Dessert.heure_arrivée, train => Train.num, durée : time)

### *Relation avec Trajet*

- **Dessert** (#voyage => Voyage.id, #gare => Gare.id, heure_départ : Date, heure_arrivée : Date)

> **Avec :**
>
> - heure_arrivée < heure_départ et gare_départ ≠ gare_arrivée, un train arrive dans une gare avant d'en partir dans la classe dessert
> - Durée = heure_arrivée - heure_départ > 0 (car heure_arrivée > heure_départ dans la classe trajet),
> - numPlace > 0.

Ce qui rend la table Trajet particulière est qu’elle contient de nombreuses références étrangères. En effet, elle récupère de la table Dessert, une gare de départ avec son heure de départ et une gare d’arrivée avec son heure d’arrivée. Comme expliqué précédemment, les relations entre Voyage, Dessert et Trajet entraînent des contraintes plutôt complexes. Il faut s’assurer par exemple que toutes les références étrangères évoquées appartiennent au même voyage. Nous obtenons les contraintes suivantes :

- Relations **1 : 0..N** entre Trajet et Dessert
  - Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_départ), voyage) = Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_arrivée), voyage)
  - Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_départ), heure_départ) = Projection( Trajet, heure_départ)
  - Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_départ), heure_arrivée) = Projection( Trajet, heure_arrivée)

> **↑ La première condition fait que la gare de départ et d'arrivée dans trajet sont du même voyage.**
>
> Les deux autres conditions servent à s'assurer que l'heure d'arrivée du trajet correspond à la gare d'arrivée, et pareil pour la gare et l'heure de départ.

Ces contraintes sont essentielles puisque sans elles les gares pourraient ne pas correspondre aux horaires. Elles permettent également qu’un voyageur ne puisse pas réserver un trajet entre Compiègne et Grenoble alors qu’aucun voyage ne les dessert directement, cela demanderait plusieurs trajets.
La cohérence de la table Trajet repose sur la cohérence de la table Voyage, nous allons donc nous intéresser aux contraintes liées à la relation entre Voyager et Gare.

- Relation Dessert **0..N : 2..N** entre Voyage et Gare
  - Projection (Voyage, id) = Projection (Dessert, voyage)
  - Projection (Jointure (Voyage, Ligne, Voyage.Ligne = Ligne.id), gare_départ) ⊆ Projection (Jointure (Dessert, Voyage, Voyage.id = Dessert.voyage), gare)
  - Projection (Jointure (Voyage, Ligne, Voyage.Ligne = Ligne.id), gare_arrivée) ⊆ Projection (Jointure (Dessert, Voyage, Voyage.id = Dessert.voyage), gare)

> **↑ Tous les voyages sont représentés dans la table Dessert et ils desservent au moins deux gares : la gare de départ et d'arrivée de la ligne à laquelle ils sont associés.**

Les deux inclusions permettent à ce qu'un voyage desserve forcément le départ et le terminus de la ligne. Effectivement un voyage sur la ligne Paris-Marseille qui s'arrête à Lyon n'aurait pas de sens. Dans ce cas-là, il fait partie de la ligne Paris-Lyon.

Notre MLD contient encore de nombreuses tables et contraintes, cependant ces dernières sont bien plus triviales et ne seront donc pas étudiées dans ce rapport.

## Implémentation en SQL

L'implémentation de la base de données en SQL est la première étape qui permet de la créer concrètement. L'idée est de construire chaque table du modèle logique de données à l'aide de commandes SQL que nous utiliserons avec PostgreSQL.
Reprenons les tables étudiées précédemment :

```
CREATE TYPE Statut AS ENUM ('bronze', 'silver', 'gold', 'platine');
CREATE TYPE Type_voyageur AS ENUM ('Regulier', 'Occasionnel');

CREATE TABLE Voyageur(
        id SERIAL,
        nom VARCHAR NOT NULL,
        prenom VARCHAR NOT NULL,
        adresse VARCHAR NOT NULL,
        telephone VARCHAR NOT NULL,
        type Type_voyageur NOT NULL,
        num_carte INTEGER UNIQUE NULL,
        statut Statut NULL,
        PRIMARY KEY (id),
        CHECK (NOT (type='Occasionnel' AND num_carte is NOT NULL AND statut is not NULL)),
        CHECK (NOT (type='Regulier' AND num_carte is NULL AND statut is NULL))
);
```

Le type SERIAL que l'on attribue à l'attribut id permet l'auto incrémentation sans que le champ soit renseigné. Cela s'adapte parfaitement à une clé primaire puisque chaque numéro sera unique. Nous avons choisi de créer deux TYPE enum pour le statut et le type de voyageur. Cela évite d'avoir à rajouter des contraintes. Ensuite nous avons ajouté les contraintes liées à l'héritage en utilisant CHECK.
Cependant toutes les contraintes ne sont pas réalisables avec des checks. Notamment les contraintes complexes étudiées dans la partie précédente. Afin de

vérifier la cohérence de notre base de données, nous avons décidé de créer des vues qui reprennent ces différentes contraintes du MLD.

```
/*Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_départ), voyage) =
  Projection (Jointure (Trajet, Dessert, Dessert.gare = Trajet.gare_arrivée), voyage)
*/

/*on relève les voyage depuis trajet selon gare de depart*/
CREATE VIEW Trajet_voyage_depart AS
  SELECT Trajet.voyage
  FROM Trajet
  INNER JOIN Dessert ON Dessert.voyage = Trajet.voyage AND Dessert.gare = Trajet.gare_depart;

/*on relève les voyages depuis trajet selon gare d'arrive*/
CREATE VIEW Trajet_voyage_arrivee AS
  SELECT Trajet.voyage
  FROM Trajet
  INNER JOIN Dessert ON Dessert.voyage = Trajet.voyage AND Dessert.gare = Trajet.gare_arrive;

/*on joint les même voyages selons les 2 précédentes vues*/
CREATE VIEW Trajet_voyage_correspondant AS
  SELECT DISTINCT Trajet_voyage_depart.voyage
  FROM Trajet_voyage_depart
  INNER JOIN Trajet_voyage_arrivee ON Trajet_voyage_depart.voyage = Trajet_voyage_arrivee.voyage;
```

```
/*tester l'égalité des projections, si la vue est null alors il n'y a pas de problème*/
CREATE VIEW check_voyage_trajet AS
  SELECT *
  FROM Trajet_voyage_correspondant
  EXCEPT
  SELECT *
  FROM (SELECT Trajet.voyage
        FROM Trajet
        INNER JOIN Dessert ON Dessert.voyage = Trajet.voyage AND Dessert.gare = Trajet.gare_arrive) t;
```

Ces vues permettent de vérifier si deux gares du même trajet n'appartiennent pas au même voyage. Contrairement à un check, il ne s'agit pas d'une vérification automatique, elle requiert la vérification d'un humain. Mais nous ne connaissons pas de méthode permettant d'appliquer des contraintes de cette sorte à notre base de données. Nous avons réalisé toutes les contraintes du MLD de façon similaire.

L'intérêt d'une base de données est évidemment de stocker des données, mais d'également pouvoir l'interroger en fonction de nos besoins. Nous avons donc créé différentes requêtes SQL permettant d'obtenir des informations pertinentes selon les besoins du client.

```sql
/*Requête qui retourne la liste des voyages qui desservent deux gare. (Donc avec deux gares données) */

SELECT v.id as Voyage, g1.nom as Gare_depart, d1.heure_depart, g2.nom as Gare_arrivee, d2.heure_arrive, STRING_AGG(CAST(jv.jour AS VARCHAR), ', ')
FROM Voyage v
JOIN Dessert d1 ON v.id = d1.voyage
JOIN Gare g1 ON d1.gare = g1.id
JOIN Dessert d2 ON v.id = d2.voyage
JOIN Gare g2 ON d2.gare = g2.id
JOIN Jour_Voyage jv ON v.id = jv.voyage
WHERE g1.nom = 'Paris gare de Lyon' /*[Gare de départ]*/
AND g2.nom = 'Gare de Marseille' /*[Gare d'arrivée]*/
AND d1.heure_depart < d2.heure_arrive
GROUP BY v.id, g1.nom, d1.heure_depart, g2.nom, d2.heure_arrive
ORDER BY d1.heure_depart;
```

Cette première requête permet d'obtenir tous les voyages entre deux gares avec les horaires de départ et d'arrivée ainsi que les jours concernés. Cette requête correspond à ce qu'un client pourrait rechercher afin de réserver ses billets de train. Afin de rendre l'affichage plus lisible, nous avons utilisé la fonction STRING_AGG permettant d'afficher les jours dans une seule et même colonne. Les voyages sont également triés par ordre croissant d'heure de départ afin de simplifier la compréhension du tableau.

| voyage | gare_depart | heure_depart | gare_arrivee | heure_arrive | jours |
|---|---|---|---|---|---|
| 3 | Paris gare de Lyon | 06:00:00 | Gare de Marseille | 09:20:00 | Mercredi, Vendredi |
| 2 | Paris gare de Lyon | 11:00:00 | Gare de Marseille | 14:10:00 | Mardi, Jeudi |

(2 rows)

Nous avons donc créé plusieurs requêtes destinées aux clients de l'entreprise, mais dans un contexte réel il est important de penser aux professionnels de l'entreprise. Nous avons ajouté des requêtes qui pourraient répondre aux besoins des gestionnaires. Voici deux exemples :

| id_ligne | nombre_voyageurs |
|---|---|
| 2 | 2 |
| 1 | 1 |

(2 rows)

| train | nombre_voyageurs | nb_places_restantes |
|---|---|---|
| 7250 | 1 | 539 |
| 5238 | 1 | 239 |
| 1876 | 1 | 539 |

(3 rows)

Ce sont des requêtes permettant d'obtenir des statistiques utiles à l'entreprise. La première permet d'obtenir le nombre de voyageurs prévu par ligne trié par ordre croissant. Et la seconde, le nombre de voyageurs par train programmé. Il est donc possible de savoir quelles sont les lignes les plus populaires, ainsi que le taux de remplissage des trains.

## Application Python

L'application python est un produit qui a pour vocation d'être directement utilisé par les différents acteurs de l'entreprise. Il a fallu dans un premier temps définir les différents types d'utilisateurs qui peuvent accéder à l'application. Nous avons implémenté les trois types d'utilisateur suivant :

- Administrateur : il a un contrôle total sur la base de données. Il peut ajouter des éléments, en supprimer et avoir une vue d'ensemble sur la bdd.
- Gestionnaire : il a un contrôle partiel de la base de données. Par exemple, ajouter des dates exceptionnelles dans la programmation des trains ou contrôler les informations des clients
- Client : il peut contrôler l'horaire des trains, réserver des billets ou gérer son compte.

```
Vous êtes dans l'espace ADMIN. Choisissez
==========AJOUT DE DONNEES==========      ========SUPPRESSION DE DONNEES=======    ========CONSULTATION DE DONNEES=======
1.Ajouter une gare                       11.Supprimer une gare                    20.Afficher les gare
2.Ajouter un taxi                        12.Supprimer une ligne                   21.Afficher les ligne
3.Ajouter un hotel                       13.Supprimer un type de train            22.Afficher les trains
4.Ajouter un transport public            14.Supprimer un train                    23.Afficher les voyages
5.Ajouter une ligne                      15.Supprimer un voyage                   24.Afficher les voyageurs
6.Ajouter un type de train               16.Supprimer un voyageur                 25.Afficher les billets
7.Ajouter un train                       17.Supprimer un billet                   26.Afficher les dessertes
8.Ajouter une date exception             18.Supprimer une desserte
9.Ajouter un voyage                      19.Supprimer un trajet
10.Ajouter une desserte                  ===================================
====================================
```

```
Vous êtes dans l'espace GESTIONNAIRE. Choisissez
==========AJOUT DE DONNEES==========
1.Relier un hotel à une gare
2.Relier un transport public à une gare
3.Relier un taxi à une gare
4.Associer une date hebdomadaire à un voyage
5.Associer une date exception à un voyage
====================================
========MODIFICATION DE DONNEES=======
6.Modififer le jour d'un voyage
7.Modifier le type d'un voyageur
8.Modifier l'assurance d'un billet
9.Modifier les horaires d'une desserte
10.Modifier le numéro de siège
====================================
```

```
Vous êtes dans l'espace CLIENT. Choisissez le numéro
==========CONSULTER DES DONNEES==========
1.Consulter les différents voyages entre deux gares
2.Chercher un trajet
3.Consulter vos trajets
====================================
========ESPACE CLIENT=======
6.Afficher vos informations
7.Modifier votre statut
8.Réserver un billet
9.Annuler une réservation
10.Modifier la date d'une reservation
11.Modifier l'assurance d'un billet
12.Supprimer votre compte
====================================
```

Pour utiliser l'application, il faut se connecter à l'aide d'un identifiant et d'un mot de passe. Afin de gérer le système d'identification, nous avons ajouté à notre base de données une table Utilisateur. Ainsi il est possible de créer des comptes et de leur associer un statut parmi les trois disponibles.

```sql
CREATE TYPE Role_utilisateur AS ENUM ('admin', 'gestionnaire', 'client');

CREATE TABLE Utilisateur
(
    id              SERIAL,
    nom_utilisateur VARCHAR UNIQUE   NOT NULL,
    mot_de_passe    VARCHAR          NOT NULL,
    role            Role_utilisateur NOT NULL,
    PRIMARY KEY (id)
);
```

Ce système permet de sauvegarder les comptes clients et de notamment de sauvegarder les données des clients avec leurs trajets et leurs informations personnelles. Ainsi un client peut retrouver facilement ses prochains trajets comme il pourrait le faire sur son compte SNCF.

```
Votre information de voyageur :
Nom : Jean
Prénom : Dupont
Adresse : rue des potiers
Téléphone : 0765325678
Type de voyageur : Occasionnel
Trajet prévu le 2023-09-10 au départ de Gare du Nord à 08:00:00 et à l'arrivée de Gare de Lille à 09:30:00 pour une durée totale de 1:30:00.
Trajet prévu le 2023-09-14 au départ de Paris gare de Lyon à 06:00:00 et à l'arrivée de Gare de Grenoble à 08:00:00 pour une durée totale de 2:00:00.
```

Concernant les gestionnaires, ils peuvent interagir avec les réservations des clients comme avec l'exemple ci-dessous, où un gestionnaire modifie l'assurance sur un billet du client Jean Dupont.

```
Choisissez le voyageur dont vous voulez modifier le billet :
2. Michelle Hernandez, Téléphone : 0953678497, Type : Regulier, Numéro de carte:192873 , Statut:platine
3. Martin Marie, Téléphone : 0145678901, Type : Regulier, Numéro de carte:123456 , Statut:bronze
4. Garcia Pedro, Téléphone : 0123456789, Type : Occasionnel
1. Jean Dupont, Téléphone : 0765325678, Type : Occasionnel
Numéro du voyageur : 1
1. Gare du Nord 08:00:00 - Gare de Lille 09:30:00 le 2023-09-10, place n°35. 50€ , Assurance : False
2. Paris gare de Lyon 06:00:00 - Gare de Grenoble 08:00:00 le 2023-09-14, place n°42. 140€ , Assurance : False
Numéro du billet : 1
Vous n'avez pas décidé de prendre une assurance sur votre billet. Voulez vous changer (oui/non) ? oui
La modification a bien été prise en compte.
```

L’objectif avec l’application Python est d’offrir une expérience la plus proche du réel avec les outils qui sont à notre disposition. Nous avons pour cela proposé une grande variété de commandes pour chaque utilisateur en nous basant sur leurs besoins. Cela conclut la partie relationnelle de la base de données. Cependant, notre travail ne s’est pas arrêté là.

# PARTIE 2 : NON-RELATIONNEL

## Pourquoi le non-relationnel ?

Par la suite, nous avons mis en place un modèle comportant des éléments non-relationnels. Cela nous permet de simplifier les requêtes, surtout au niveau des jointures. Nous avons donc décidé de remplacer la composition Trajet avec Billet (Un trajet n'existe que pour un billet donné). Cela nous permet de simplifier grandement les requêtes car nous utilisons souvent la table Billet à l'aide de jointures avec Trajet dans nos requêtes, ceci rend les requêtes plus compliquées et demandent beaucoup plus de temps à être traitées par le serveur qu'une simple lecture d'un JSON.

## Changements CREATE SQL

Ceci implique des changements conséquents dans nos requêtes SQL car nous avons supprimé la table Trajet et nous avons dû adapter toutes les tables en relation avec la table Trajet. Par exemple nous avons dû modifier la relation (Trajet *..1 Train) en intégrant cette dépendance dans un attribut JSON nommé "trajet" de la table Billet.

De plus, toutes les requêtes INSERT/SELECT/UPDATE ont dû être adaptées dans le code SQL et dans l'application Python car étudier un JSON change la syntaxe de toutes les  requêtes utilisant la table Billet.

Cela a aussi des inconvénients car par exemple si un voyage change d'heure de départ, nous devons modifier cette date dans tous les JSON de tous les trajets. Ce qui implique une énorme quantité de requêtes UPDATE.
Voici les anciennes tables Billet et Trajet :

```sql
CREATE TABLE Billet
(
    id              SERIAL,
    prix            NUMERIC         NOT NULL CHECK (prix > 0),
    moyen_paiement Moyen_paiement NOT NULL,
    assurance       BOOLEAN,
    voyageur        INTEGER         NOT NULL,
    FOREIGN KEY (voyageur) REFERENCES Voyageur (id) ON DELETE CASCADE,
    PRIMARY KEY (id)
);
```

```sql
CREATE TABLE Trajet
(
    id           SERIAL,
    voyage       INTEGER,
    date         DATE CHECK (date > CURRENT_DATE),
    billet       INTEGER,
    num_place    INTEGER NOT NULL CHECK (num_place > 0),
    gare_depart  INTEGER NOT NULL,
    heure_depart TIME    NOT NULL,
    gare_arrive  INTEGER NOT NULL,
    heure_arrive TIME    NOT NULL,
    train        INTEGER NOT NULL,
    FOREIGN KEY (voyage, gare_depart, heure_depart)
    REFERENCES Dessert (voyage, gare, heure_depart)
    ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (voyage, gare_arrive, heure_arrive)
    REFERENCES Dessert (voyage, gare, heure_arrive)
    ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (billet) REFERENCES Billet (id) ON DELETE CASCADE,
    FOREIGN KEY (train) REFERENCES Train (num) ON DELETE CASCADE,
    PRIMARY KEY (id, billet),
    CHECK (gare_depart ≠ gare_arrive)
);
```

Voici les changements effectués à la table Billet après avoir inséré les trajets directement dans cette table :

```sql
CREATE TABLE Billet
(
    id             SERIAL,
    prix           NUMERIC        NOT NULL CHECK (prix > 0),
    moyen_paiement Moyen_paiement NOT NULL,
    assurance      BOOLEAN,
    voyageur       INTEGER        NOT NULL,
    trajet JSON NOT NULL,
    FOREIGN KEY (voyageur) REFERENCES Voyageur (id) ON DELETE CASCADE,
    PRIMARY KEY (id),

    CHECK (trajet->>'voyage' IS NOT NULL),
    CHECK (trajet->>'date' IS NOT NULL),
    CHECK (trajet->>'num_place' IS NOT NULL),
    CHECK (trajet->>'gare_depart' IS NOT NULL),
    CHECK (trajet->>'gare_arrive' IS NOT NULL),
    CHECK (trajet->>'heure_depart' IS NOT NULL),
    CHECK (trajet->>'heure_arrive' IS NOT NULL),
    CHECK (trajet->>'train' IS NOT NULL),

    CHECK (CAST(trajet->>'date' AS date) > CURRENT_DATE),
    CHECK (CAST(trajet->>'num_place' AS integer) > 0),
    CHECK (trajet->>'gare_depart' ≠ trajet->>'gare_arrive'),
    CHECK (trajet->>'heure_depart' < trajet->>'heure_arrive')
);
```

## Changements des autres reqûetes SQL :

Ce changement de table implique aussi des changements sur les vues et les requêtes.

Par exemple, sur la vue Duree_Trajet :
Avant :

```
CREATE VIEW Duree_Trajet AS
SELECT id, heure_arrive - heure_depart AS duree
FROM Trajet;
```

Après :

```
CREATE VIEW Duree_Trajet AS
SELECT id, CAST(trajet->>'heure_arrive' as TIME) - CAST(trajet->>'heure_depart' as TIME) as duree
FROM Billet;
```

Ceci a aussi modifié les requêtes INSERT, une nouvelle requête INSERT se fait comme cela désormais :

```
insert into Billet (prix, moyen_paiement, assurance, voyageur, trajet)
VALUES (20,
'carte bleue',
TRUE,
2,
'{
  "voyage": 2,
  "date": "2023/09/20",
  "num_place": 21,
  "gare_depart": 6,
  "heure_depart": "11:00:00",
  "gare_arrive": 5,
  "heure_arrive": "14:10:00",
  "train": 5238
}'::json);
```

De plus, dans l'application Python, il y avait aussi toutes les requêtes avec la table Trajet qui étaient à modifier. Comme par exemple la requête qui permet d'afficher tous les trajets d'un voyageur :

```
    sql2 = f"""SELECT b.id as billet, b.trajet->>'date' as date, g1.nom as Gare_depart,
b.trajet->>'heure_depart' as heure_depart, g2.nom as Gare_arrivee, b.trajet->>'heure_arrive' as heure_arrive, d.duree,
        CASE WHEN b.assurance = TRUE THEN 'Oui' ELSE 'Non' END AS assurance
        FROM Billet b
        INNER JOIN duree_trajet d ON d.id = b.id
        JOIN Gare g1 ON CAST(b.trajet->>'gare_depart' AS INTEGER) = g1.id
        JOIN Gare g2 ON CAST(b.trajet->>'gare_arrive' AS INTEGER) = g2.id
        WHERE b.voyageur = {voyageur[0]}
        ORDER BY CAST(b.trajet->>'date' AS DATE);
        """
```

Nous retrouvons bien le même résultat que dans la partie précédente :

```
Votre choix : 3
Trajet prévu le 2023/09/10 au départ de Gare du Nord à 08:00:00 et à l'arrivée à Gare de Lille à 09:30:00 pour une durée totale de 1:30:00.
Trajet prévu le 2023/09/14 au départ de Paris gare de Lyon à 06:00:00 et à l'arrivée à Gare de Grenoble à 08:00:00 pour une durée totale de 2:00:00.
```

Ainsi la transformation d'une base de données relationnelle en base non relationnelle est une tâche très lourde qui impacte toute la structure précédente. Il est important d'adapter chaque requête en utilisant la syntaxe du JSON et d'avoir des requêtes d'insertion cohérentes entre elles.

# Conclusion

Ce projet fut l'occasion d'appliquer concrètement les notions étudiées durant le semestre. Lors de chaque étape de la création de la base de données, nous nous sommes efforcé de répondre aux besoins du client et de respecter au mieux le cahier des charges afin d'offrir une application cohérente avec les besoins et exigences des utilisateurs. Enfin, le fait de réaliser une version relationnelle et non relationnelle de notre base de données nous a permis de prendre conscience des différentes utilisations et applications de ces deux méthodes.