Projet SGBD: Covoiturage du campus

Desbois-Renaudin Méo, Facen Théo, Gajic Maxime, Duchiron Jules

December 6, 2023

Contents

1 Introduction

L'objectif principal de ce projet est de mettre en place un système de covoiturage spécifiquement adapté à l'environnement campus. À travers une modélisation efficace des données, des opérations bien définies sur la base, et une implémentation pratique, le SGBD offre une solution complète pour répondre aux besoins de covoiturage des étudiants et du personnel du campus.

2 Modélisation des données

2.1 Description du contexte de l'application

L'application vise à gérer les informations liées aux voyages en voiture entre étudiants. Les entités principales incluent les étudiants, les voitures, les voyages, les arrêts, les inscriptions et les évaluations. Les associations sont établies entre ces entités pour modéliser les relations.

Les règles de gestion incluent la gestion des inscriptions aux arrêts, l'évaluation des étudiants pour les voyages, et la gestion des contraintes d'intégrité pour maintenir la cohérence des données.

2.2 Modèle Entité-Association

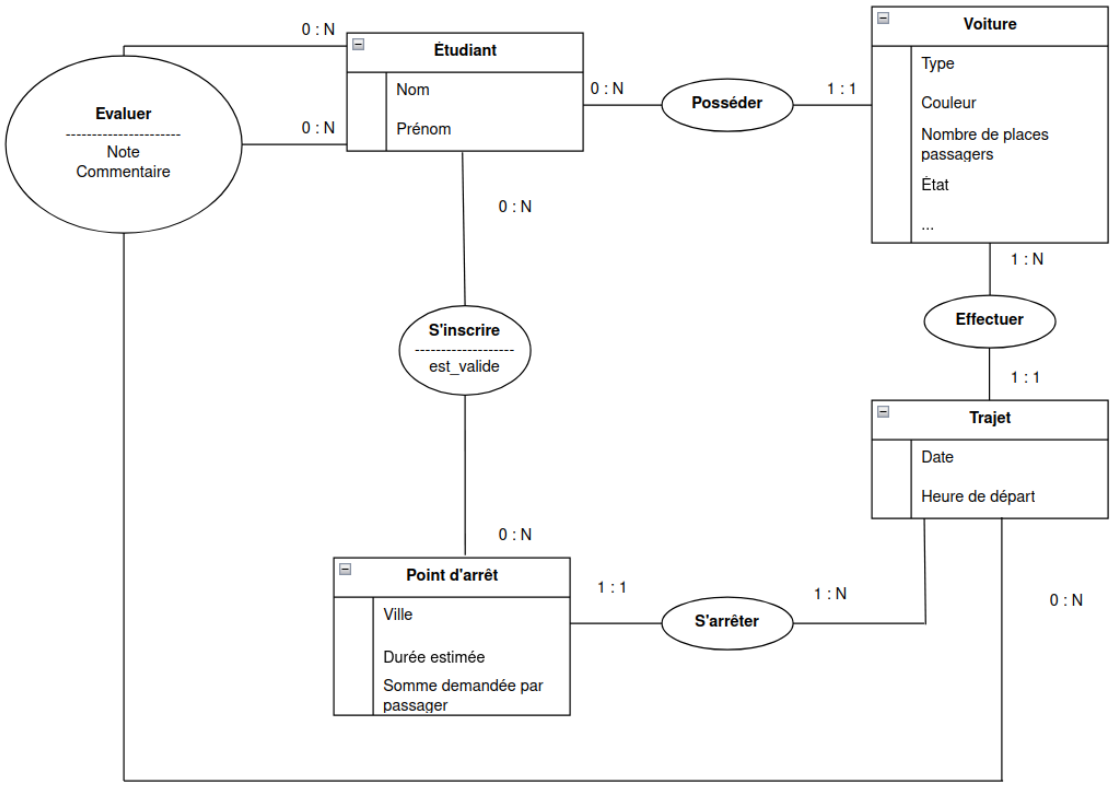

Figure 1: Modèle Entité-Association

2.3 Liste des Opérations Prévues sur la Base

Il est très **important** de noter que vu notre implémentation du server en nodejs, nous ne nous servons pas des fichier `select.sql` et `update.sql`, les requêtes utilisées par le site sont répértoriées dans le dossier `server/queries/`. Pour que vous puissiez y avoir accès, nous avons recopié les requêtes dans les fichiers `select.sql` et `update.sql`.

Nous avons ici répertorié toutes les opérations possibles sur la base de données

2.3.1 Consultations

- Consultation des étudiants (tous ou par ID)
- Consultation des voitures (toutes ou par ID)
- Consultation des voyages (tous ou par ID)
- Consultation des arrets proposés pour les voyages (tous ou par ID)
- Consultation des évaluations (toutes ou par ID)
- Consultation des inscriptions (toutes ou par ID d'étudiant et d'arrêt)
- Recherche d'étudiant par nom ou prénom
- La liste des véhicules non remplis pour une date donnée et une ville donnée donnée
- Consultations des arrêts pouvant être desservis dans un intervalle de jours donné
- Consultations des trajets pouvant desservir un arrêt donné dans un intervalle de temps

2.3.2 Modifications

- ajout, suppression et modification d'étudiant (par ID)
- ajout, suppression et modification de voitures (par ID)
- ajout, suppression et modification de voyage (par ID)
- ajout, suppression et modification d'arret (par ID)
- ajout, suppression et modification d'evaluation (par ID)
- ajout, suppression et modification d'inscription (par ID d'étudiant et d'arret)

2.3.3 Statistiques

Nous avons aussi réalisé des opérations statistiques. C'est-à-dire que nous pouvons afficher :

- Moyenne de passagers par trajet
- Distance parcourue en moyenne par jour
- 5 meilleurs conducteurs
- 5 meilleures villes selon le nombre de trajets qui les dessert

3 Schéma Relationnel

Nous allons expliquer comment nous avons passé notre modèle entité-association en un modèle relationnel, le résultat final etant donné en figure 2.

3.1 Passage au relationnel

Tout d'abord, chaque entité devient une table, et les associations comportant une cardinalité maximale 1 deviennent des clés étrangères dans la table du côté en question :

- L'association *Posséder* Devient une clé étrangère `id_propriétaire` dans la table Voiture.
- L'association *Effectuer* Devient une clé étrangère `id_voiture` dans la table Voyage.
- L'association *S'arrêter* Devient une clé étrangère `id_voyage` dans la table Arret.

Ensuite, les associations restantes deviennent des nouvelles tables.

Par exemple, l'association *S'inscrire* devient une table *Inscription* Ayant pour clé primaire les clés étrangères `id_etudiant` et `id_arret` faisant référence aux deux tables du même nom.

Pour l'association *évaluer*, nous avons décidé de faire autrement. En effet, nous avons d'abord pensé à donner à la nouvelle table évaluation pour clé primaire les clés étrangères référencant etudiant et Voyage. Cependant, nous voulions pouvoir conserver les entrées dans la table évaluation même si un étudiant ou voyage réféérencé par celle-ci venait à être supprimé. Nous avons donc ajouté à cette nouvelle table *Evaluation* une clé primaire propre id_evaluation comme montré en figure 2.

3.2 Contraintes d'Intégrité, Dépendances Fonctionnelles

Les associations devenues clés étrangère précédemment citées sont des dépendances fonctionnelles entre les clés étrangères et les clés primaires des tables qu'elles référencent.

3.3 Schéma Relationnel en 3e Forme Normale

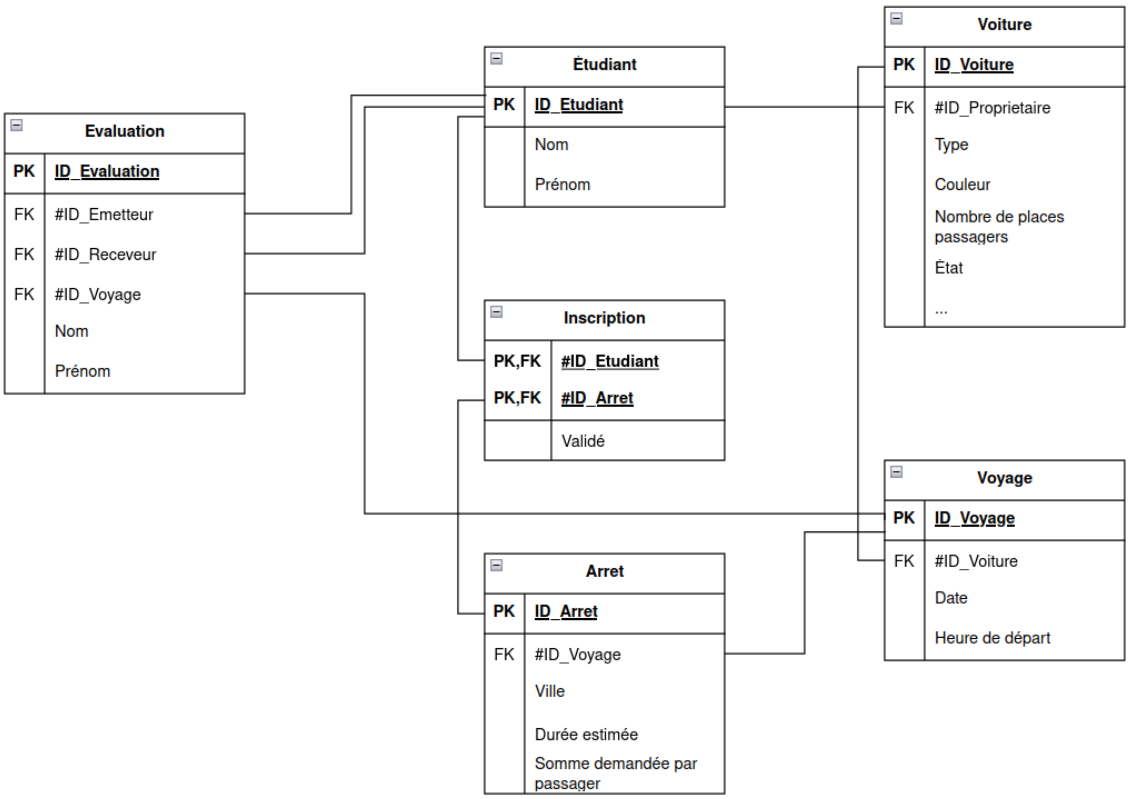

Figure 2: Modèle relationnel

4 Implantation (Base SQL au Choix)

4.1 Création de la Base de Données

Pour mettre en place la base de données, nous avons un fichier sql *create.sql* créant les tables, un fichier *insert.sql* insérant les données dans ces dernières et *drop.sql* permettant de supprimer les tables.

Dans le `script de création`, nous avons 6 tables à créer : *Evaluation, Inscription, Arret, Voyage, Voiture, Etudiant.* Leurs clés primaires sont créés avec la commande *PRIMARY KEY*, assurant `l'existence` et `l'unicité` de ces dernières.

Les clés secondaires sont quant-à elles créés avec la commande *NOT NULL*, permettant la contrainte `d'existence`. Ces attributs sont ensuite définies comme clés étrangères une fois que toutes les tables ont été créés en utilisant la commande *ALTER TABLE* afin de maintenir l'`intégrité réferentielle` entre les tables.

On doit également vérifier la cohérence des valeurs insérées dans les tables, comme par exemple qu'une inscription validée ne concerne pas un étudiant déja inscrit dans le voyage, ou un voyage plein. On doit également vérifier que les avis émis concernent des étudiants présents lors du voyage, et qu'il ne donnent pas un avis sur eux-mêmes. Pour cela, on a soit la solution de faire un *CHECK* lors de la création de la table (ce qui a été fait pour s'assurer qu'on ne puisse pas émettre d'avis sur soi-même), ou alors on peut créer un *déclencheur* vérifiant à chaque insertion ou mise à jour la cohérence des valeurs (ce qui a été utilisé pour le reste des contraintes).

Les tables *Voiture, Arret et Inscription* ont des contraintes *ON DELETE CASCADE* sur leurs *clés étrangères.* Ainsi, si des enregistrements référencés par ces clés sont supprimées, les enregistrements possèdant ces clés étrangère le sont également. Par exemple, lorsqu'un arrêt qui est référencé par son identifiant dans la table Inscription, toutes les inscriptions le concernant sont également supprimées. Cela permet de maintenir la `cohérence des données et l'intégrité réferentielle` entre les table de la base de données.

Enfin, il existe des tables telles que *Voyage* ou *Evalution* que nous souhaitons conserver même si un des enregistrement qu'ils référencent est supprimé (ex on souhaite conserver un voyage même si la voiture a été supprimée). Dans ce cas, pour conserver la `cohérence des données et l'intégrité réferentielle` entre les table de la base de données on utilise des `déclancheurs`. Ainsi lors de la suppression d'une voiture, les voyages concernant cette dernière sont mis à jour en remplaçant l'identifiant par une valeur nulle. De même pour la table évaluation lorsque des étudiants ou des voyages sont supprimés.

Dans le `script d'insertion`, les jeux de données sont définis de manière cohérente pour respecter les contraintes d'intégritées des tables.

Dans notre cas, on commence par insérer la table `Etudiant` qui ne possède pas de clé étrangère. Puis nous insérons `Voiture` qui ne référence que sur *Etudiant.* Comme les valeurs pour la table Etudiant ont bien été insérées, on pourra bien utiliser les valeurs de `id_etudiant` pour insérer les valeurs de la clé étrangère `id_proprietaire` dans la table `Voiture`. L'insertion des valeurs des tables n'auront pas été possibles dans l'autre sens car les valeurs de `id_etudiant` n'auront pas été insérées. Il n'aurait donc pas été possible d'insérer les valeurs de la clé étrangère `id_proprietaire` dans la table `Voiture`.

Nous continuons ainsi à insérer dans l'ordre les tables pour lesquelles les valeurs de leurs clés étrangères sont bien présentes : `Voyage` (qui dépend de `Voiture`), `Arret` (qui dépend de `Voyage`), `Inscription` (qui dépend de `Etudiant` et de `Arret`) pour finir par `Evalutation` (qui dépend de `Etudiant` et de `Voyage`).

dans le `script de suppression`, les tables sont supprimées dans un ordre précis permettant de

respecter les contraintes d'intégrité référentielles. Elles sont supprimées dans le `sens inverse du sens d'insertion`.

Ainsi, on commence par supprimer les tables `Evaluation et Inscription` qui ne sont référencées dans aucune autre table. Puis on supprime `Arret` qui n'était référencé que par *Inscription* qui a bien été supprimé précédemment. Nous continuons ainsi en supprimant `Voyage, Voiture puis Etudiant`.

4.2 Implémentation des Commandes SQL

Nous avons implémenté les requêtes afin de réaliser les opérations que nous avons détaillé auparavant. Nous allons nous arrêter sur 2 requêtes, une de consultations et une de statistiques.

- Pour la requête permettant de connaître les trajets partant un jour et d'une ville spécifique. A l'aide des '$1' et '$2' on peut avoir accès au jour et à la ville demandée par l'utilisateur, ce qui est nécessaire pour les requêtes de consultation.
- Pour la requête donnant le classement des meilleurs conducteurs. Nous selectionnons les noms et prenoms, ainsi que leur note moyenne des 5 meilleurs candidats. Pour cela, on effectue une jointure entre les tables Evaluation et Etudiant sur id_receveur et id_etudiant, ainsi que des jointures naturelles avec les tables Voyage et Voiture. Cela nous permet d'avoir d'avoir la moyenne des notes reçues, et ce uniquement pour les conducteurs. L'instructions 'Limit 5' permet de garder uniquement les 5 meilleurs.

5 Utilisation

5.1 Environnement d'Exécution

L'application est composée de trois parties distinctes :

5.1.1 Base de données

Postgresql est le système de gestion de base de données relationnelle utilisé pour le projet. Elle est hébergé localement.

5.1.2 Serveur

Le serveur joue un rôle important en facilitant la communication entre la base de données et l'utilisateur, permettant ainsi la gestion efficace de la mise à jour de la base sans nécessiter que le client envoie des requêtes SQL brutes. Réalisé en JavaScript et Node.js, ce serveur s'appuie sur l'utilisation de frameworks pour simplifier sa gestion.
La structure du serveur se décompose en plusieurs éléments :

Fichier db.js : Ce fichier établit la connexion avec la base de données PostgreSQL, assurant une liaison stable et sécurisée entre le serveur et la base de données.

Fichier server.js : Responsable du démarrage du serveur, ce fichier permet également d'écouter sur un port les messages envoyés par le client, facilitant ainsi la communication bidirectionnelle entre le frontend et le backend.

Dossier controller : Contenant l'ensemble des controllers dédiés à chaque table de la base de données, ce dossier joue un rôle essentiel dans la gestion des opérations liées à chaque table pour les mises à jour de la base de données.

Dossier queries : Centralisant l'ensemble des requêtes SQL regroupées par table, ce dossier est sollicité lors des opérations de communication avec la base de données. Il contribue à maintenir une structure organisée et modulaire du code.

Dossier route : Ce dossier contient les fichiers qui associent les EndPoints de l'API aux fonctions du contrôleur. Il assure la correspondance entre les requêtes HTTP reçues et les actions spécifiques

à entreprendre pour chaque table, établissant ainsi le lien essentiel entre le frontend et le backend de l'application.

La mise en place de ce serveur structuré et modulaire en JavaScript permet d'optimiser la gestion des requêtes, d'assurer la sécurité des communications avec la base de données, et de simplifier le processus de mise à jour de la base sans nécessiter des requêtes SQL complexes de la part du client.

5.1.3 Interface Client

L'interface utilisateur du projet a été développée à l'aide du framework Angular, et elle présente deux types d'interfaces distinctes. Angular a été choisi en raison de ses nombreuses fonctionnalités, de sa structure modulaire, de sa facilité d'utilisation.

Cette interface permet aux utilisateurs de visualiser de nombreuses requêtes de consultations. Elle peut afficher des statistiques, des classements, ou toute autre information pertinente concernant la base de données. Angular facilite la création de composants réutilisables pour présenter ces données de manière claire et conviviale.

Ces interfaces sont dédiées à l'affichage du contenu des tables principales du projet : les étudiants, les voyages et les voitures. Angular facilite la création de composants spécifiques à chaque table, permettant d'ajouter et de rechercher des éléments de manière intuitive.

5.1.4 Description des interfaces

L'application est composée de cinq pages :

Pages de consultations : Ces pages permettent de consulter les informations sur les conducteurs, les passagers, la liste des véhicules disponibles pour un jour donné pour une ville donnée, Les trajets pouvant desservir une ville donnée dans un intervalle de temps.

Page de statistique: Cette page affiche les moyenne des passagers sur l'ensemble des trajets effectués, lamoyenne des distances parcourues en covoiturage par jour, le classement des meilleurs conducteurs d'aprés les avis, le classement des villes selon le nombre de trajets qui les dessert.

Page d'ajout: Cette page permet d'ajouter des elements dans une table.

5.2 Notice d'Utilisation

Déployer la base de données :

```
sudo -u postgres psql
postgres=# CREATE DATABASE projets7e4;
postgres=# \c projets7e4
projets7e4=# \i sql/create.sql
projets7e4=# \i sql/insert.sql
```

Lancer le serveur back-end :

```
cd server
npm install
npm start
```

```
(vous pouvez lancer le serveur back-end en arrière-plan pour ne pas avoir à lancer de nouveau termina
```

Lancer le front-end :

```
(se mettre à la racine du projet dans un nouveau terminal)
cd front
npm install
npm start
```

Regarder le site :

Ouvrez votre navigateur et entrez l'URL donnée lors du lancement du front-end. Normalement, cette URL est : `http://localhost:4200/`

Vous pouvez aussi accéder au back-end via les routes prévues dans `queries/routes/` à cette URL : `http://localhost:1234/`.
Par exemple : `http://localhost:1234/api/etudiant/name/john`. Cette URL recherche les étudiants dont le nom ou prénom contient "john".

À noter :

Vu notre implémentation du serveur en Node.js, nous ne nous servons pas des fichiers `select.sql` et `update.sql`. Les requêtes utilisées par le site sont répertoriées dans le dossier `server/queries/`.

Pour que vous puissiez y avoir accès, nous avons recopié les requêtes dans le fichier `select.sql`. Comme convenu, nous n'avons pas déployé le site sur bordeaux-inp car cela semblait irréalisable en Node.js.