Filière Informatique

Compte rendu projet SGBD : Cartes à collectionner

Coin-coin de l'espace :
Théo Matricon
Tanguy Pemeja
Maxime Rosay

Encadrant :
Sylvain Lombardy

Version du 11 décembre 2019

Table des matières

1 Introduction

L'objectif de ce projet est de créer une base de données avec une interface graphique : de sa modélisation jusqu'à son implémentation.

1.1 Choix du sujet : les cartes à collectionner

Nous avons choisi le sujet concernant des cartes à collectionner. Il consiste à réaliser une base de données dans laquelle des joueurs peuvent rentrer leur collection de cartes mais aussi réaliser quelques actions :

- Consulter les cartes qu'ils possèdent (nombre d'exemplaires, version..)
- Créer/modifier/supprimer des decks
- Enregistrer les parties qu'ils ont joué en précisant le deck utilisé
- Préciser différents aspects sur l'obtention de leurs cartes (prix, état..)

1.2 Outils utilisés

Dans le but de réaliser cette base de données et son interface graphique, nous avons choisi d'utiliser les outils suivants :

- MySQL pour le Système de Gestion de Base de Données, nous l'avons choisi puisqu'il se rapproche le plus de `node.js` pour la réalisation de notre interface graphique afin de croiser nos connaissances des différents cours que nous suivons
- JavaScript pour notre interface graphique, en se basant sur le framework `Node.js` pour les serveurs. On disposera d'une serveur API REST qui permettra la communication avec la base de données et d'un serveur web qui servira les pages webs.

2 Historique des versions

L'idée dans cette partie est de retracer l'évolution de notre modèle conceptuel pour la base de données, de notre première version naïve à celle qui est réellement implémentée.

2.1 Une première version complexe

Après avoir lu attentivement le sujet plusieurs fois, nous avons abouti à une première ébauche de modèle entités-associations pour le sujet :

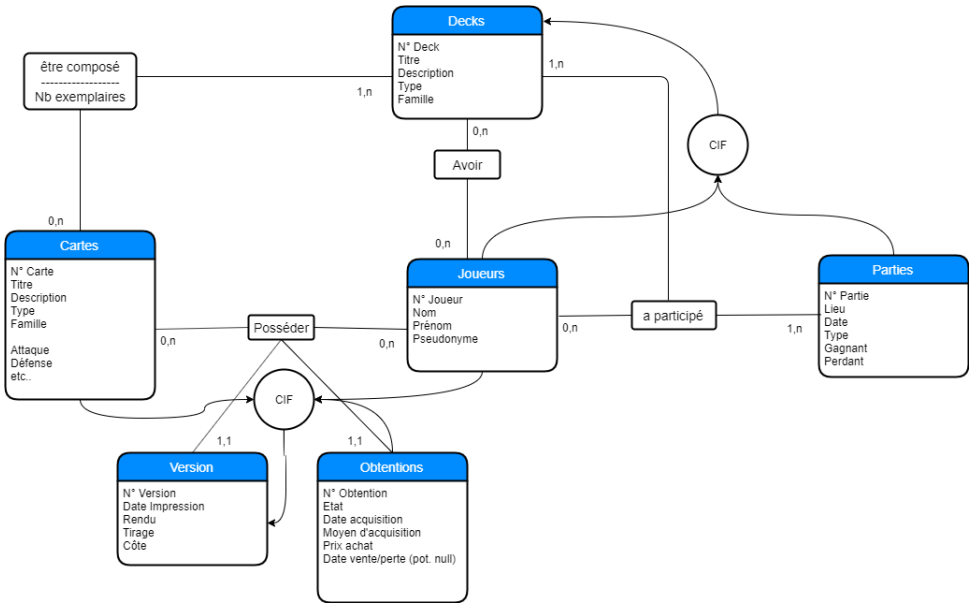

FIGURE 1 – Premier modèle conceptuel

Une de nos premières idées est de séparer Cartes et Versions. En effet, il peut exister plusieurs versions d'une même carte, par exemple une version *brillante* et une version *normale*.

Cependant, nous n'étions pas satisfaits sur l'aspect visuel du modèle, c'est-à-dire la relation quaternaire composée des entités *Cartes, Versions, Obtentions, Joueurs* nous semblait très lourde. De plus, les deux cardinalités (1,1) pour les entités Version et Obtentions n'allaient pas non plus, en effet avec ces cardinalités, ces entités vont disparaître avec la mise en troisième forme normale. Le modèle sur cette partie là est clairement insatisfaisant.

Sur un aspect plus général, un deck est composé d'un certain nombre d'exemplaires d'une carte. Une carte peut apparaître dans un nombre quelconque de decks. Nous avons choisis de lier une Carte à un Deck et de ne pas lier sa version, ce qui est un point purement cosmétique. De plus, pour copier les decks d'autres joueurs, il suffit d'utiliser la même carte, utiliser la même carte et la même version n'est pas nécessaire. Un joueur possède un nombre quelconque de decks. Et un deck n'est pas forcément associé à un joueur, par exemple si le joueur A réalise une partie avec le deck D puis supprime le deck D. On doit toujours être capable de retrouver le contenu du deck D malgré le fait que A ne soit plus lié à D. Cela va nous amener à la CIF $(joueur, partie) \longrightarrow deck$, en effet une partie est associée à deux joueurs et deux decks, on doit donc pouvoir retrouver le deck utilisé par chaque joueur.

2.2 Une seconde version remaniée

Après avoir échangé avec notre encadrant pour le projet, nous sommes donc revenus sur cette relation quaternaire afin de la simplifier en ajoutant deux nouvelles associations. Ainsi, notre modèle comporte seulement des relations

binaires et une seule ternaire avec une CIF. On a gardé la même partie droite du modèle qui nous semble satisfaisante.

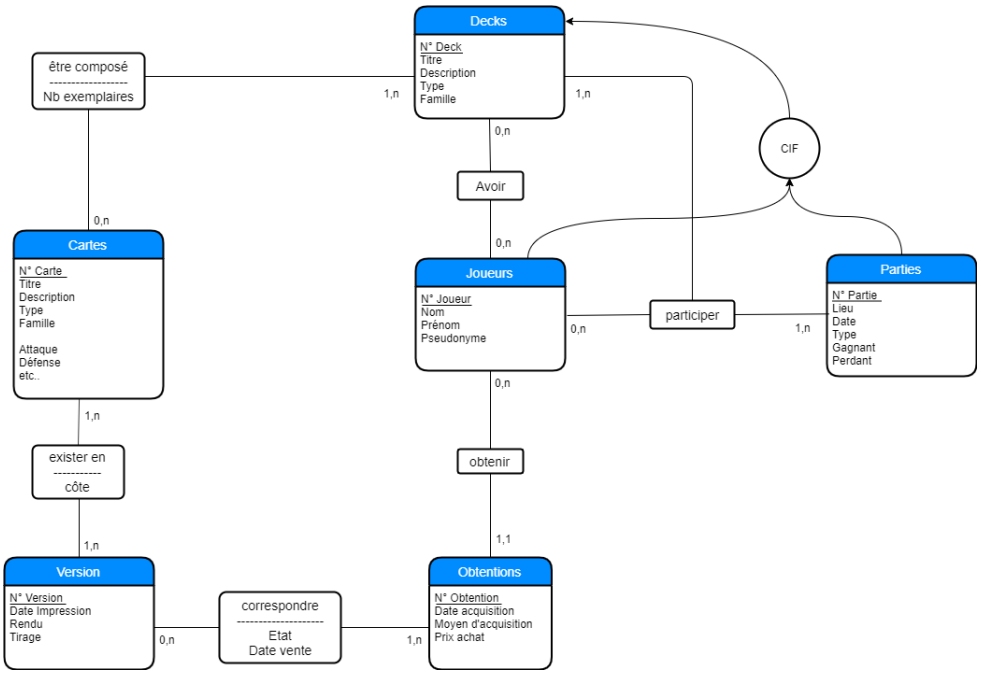

FIGURE 2 – Second modèle conceptuel

L'idée est donc qu'une Obtention soit liée à une Carte et la Version dans laquelle est obtenue. On a donc une CIF $(obtention, carte) \longrightarrow version$. Une carte peut être obtenue un nombre quelconque de fois. Une obtention est liée à un seul joueur.

2.3 Version après le rendu

Le modèle ci-dessous correspond à celui que l'on a implémenté, quelques modifications mineures avec le précédent :

- L'attribut *Date* de l'entité **Parties** a été renommé en *Date Partie*
- Ajout d'un attribut *Nb Exemplaires* dans l'association **exister en** des entités **Cartes** & **Versions**

Néanmoins, au fur et à mesure du développement nous nous sommes rendus compte de plusieurs problèmes. Tout d'abord, la cardinalité 1,1 de Obtentions implique le remaniement de l'entité pour la faire disparaître dans l'association. Ensuite, l'association Version-Cartes nous semble mal développée, en effet, l'idée de séparer version et cartes est de pouvoir stocker quelques versions et de les lier à beaucoup de cartes. Mais à cause de l'association Obtentions-Version, une Version doit être liée à une unique carte seulement sinon nous ne pouvons pas associer une obtention à une carte. Notre modèle conceptuel doit donc être retravaillé.

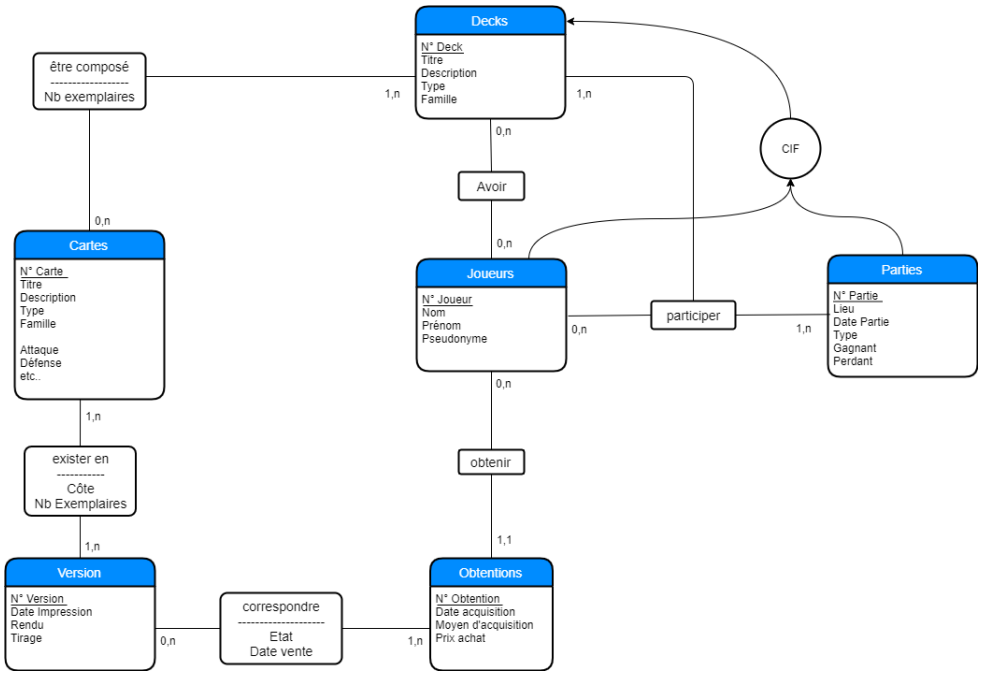

FIGURE 3 – Modèle conceptuel rendu

2.4 Modèle final

Le modèle final est finalement un mélange des versions précédentes. Tout d'abord, une carte peut exister en un certain nombre de versions. Ensuite, un joueur est associé à des obtentions qui correspondent à l'acquisition de cartes. Une obtention est liée à un joueur et peut correspondre à un ensemble de couples (Version,Carte) comme on pourrait avoir dans un *booster*[1]. Un couple (Version, Carte) peut être obtenu un nombre quelconque de fois.

1. paquet de cartes en français

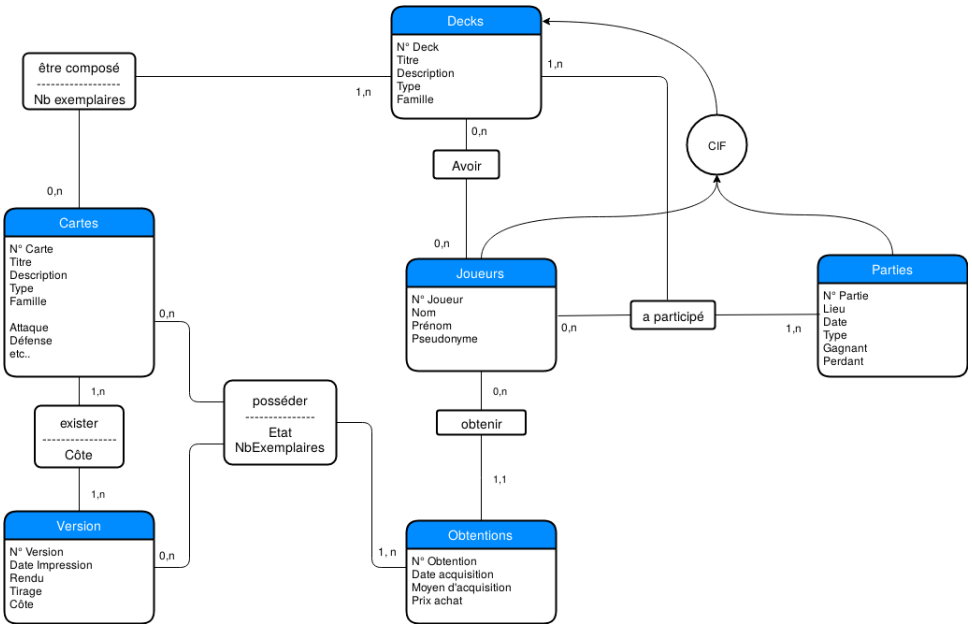

FIGURE 4 – Modèle conceptuel final

2.5 Modèle relationnel

En modèle relationnel maintenant nous obtenons ceci mis sous troisième forme normale :

```
Parties(_NoPartie, Lieu, DatePartie, Type, #NoJoueurGagnant,
        #NoJoueurPerdant)
PartieParticipation(_#NoPartie, _#NoJoueur, #NoDeck)
Joueurs(_NoJoueur, Nom, Prenom, Pseudonyme)
Obtentions(_NoObtention, Date_Acquisition, Moyen_Acquisition,
           Prix_achat, #NoJoueur)
Possessions(_#NoObtention, _#NoCarte, _#NoVersion, _Etat,
            NbExemplaires)
Versions(_NoVersion, DateImpression, Rendu, Tirage)
Cartes(_NoCarte, Titre, Description, Type, Famille, Attaque,
       Defense)
VersionCartes(_#NoCarte, _#NoVersion, Cote)
Decks(_NoDeck, Titre, Description, Type, Famille)
DeckComposition(_#NoDeck, _#NoCarte, NbExemplaires)
AvoirDeck(_#NoJoueur, _#NoDeck)
```

La plupart des entités sont traduites telles quelles et toutes les associations sont aussi transformées en tables. Nous allons expliquer la table Possessions qui correspond à l'association ternaire Cartes-Version-Obtentions. Une possession lie les 3 éléments, au commencement nous ne voulions pas de table Possessions

et nous préférions déplacer les champs dans Obtentions au vu de la cardinalité. Néanmoins, Obtentions peut correspondre à des ouvertures de paquets de cartes. Donc une obtention doit pouvoir être associée à plusieurs couples (carte, version), d'où la création de la table Possessions. La clé primaire de Possessions est le n-uplet (NoObtention, NoCarte, NoVersion, Etat), en effet si nous enlevons l'état de la clé primaire alors il n'est pas possible d'obtenir deux fois le même couple (carte, version) dans la même obtention avec différents états. Cela nous semblait une mauvaise idée, puisque les joueurs peuvent échanger des cartes et éventuellement plusieurs cartes qui ne sont pas dans le même état, cela reviendrait donc à créer des obtentions supplémentaires, ce que l'on évite avec notre choix de clé primaire.

3 Requêtes pertinentes

Nous allons maintenant étudier quelques requêtes qui nous semblent pertinent d'expliquer.

3.1 Liste des cartes par nombre décroissant de joueurs possédant chaque carte

```
SELECT Cartes.*, COUNT(*) as NbOwners FROM
  (Cartes NATURAL JOIN Obtentions)
  GROUP BY NoCarte ORDER BY NbOwners DESC;
```

Pour cela il faut relier les tables des cartes à leur obtention. On n'a pas besoin de faire une jointure avec la table Joueurs car Obtentions possède une cardinalité 1,1 vers celle-ci.

3.2 Liste des joueurs par nombre décroissant de parties gagnées

```
SELECT Joueurs.*, COUNT(*) as NbVictoires FROM
  ((Joueurs NATURAL JOIN PartieParticipation)
    LEFT JOIN Parties ON (Joueurs.NoJoueur = Parties.NoJoueurGagnant AND PartieParticipati
  GROUP BY NoJoueur ORDER BY NbVictoires DESC;
```

Pour cela il faut relier les tables des Joueurs à leur participation à une partie pour pouvoir les joindre aux Parties. Néanmoins on ne s'intéresse qu'aux parties où le joueur est gagnant donc on ne joint les deux tables seulement lorsque l'id du gagnant est l'id du joueur.

3.3 Liste des joueurs par valeur décroissante de leur collection

```
SELECT Joueurs.*, SUM(T2.Valeur) as ValeurJoueur FROM
  (Joueurs NATURAL JOIN
    (SELECT *, Etat * Cote * NbExemplaires / 100 as Valeur FROM
      (VersionCartes NATURAL JOIN Obtentions)
     ) as T2)
  GROUP BY NoJoueur ORDER BY ValeurJoueur DESC;
```

Tout d'abord, découpons la requête en deux grâce à cette phrase : la valeur de la collection d'un joueur correspond à la somme de la valeur des cartes qu'il possède. Une première étape consiste donc à déterminer la valeur de chaque carte. C'est ce que nous faisons avec :

```
SELECT *, Etat * Cote * NbExemplaires / 100 as Valeur FROM
      (VersionCartes NATURAL JOIN Obtentions)
      )
```

On joint les obtentions aux versions ce qui permet d'obtenir l'état, la côte et le nombre d'exemplaires d'une carte.

Une fois la valeur des cartes calculées, il suffit de les joindre aux joueurs et de les sommer, ce qui se fait naturellement car le NoJoueur est inclus dans la table Obtentions.

3.4 Liste des collectionneurs

La liste des collectionneurs est la liste des joueurs n'ayant jamais fait de partie. Nous avons traduit cela avec :

```
SELECT * FROM Joueurs
MINUS
SELECT Joueurs.*, COUNT(*) as NbParties FROM
(Joueurs NATURAL JOIN PartieParticipation) GROUP BY NoJoueur;
```

Nous faisons la différence entre l'ensemble des joueurs et l'ensemble des joueurs qui ont fait au moins une partie.

3.4.1 Requêtes avec MINUS ou EXCEPT

MySQL ne supporte pas les mots `MINUS` et `EXCEPT` nous devons donc, à travers d'autres opérations, reproduire le même comportement. Ainsi nous allons proposer une traduction à :

```
S1 MINUS S2;
```

Pour cela, nous allons utiliser la syntaxe LEFT JOIN :

```
SELECT T1.* FROM
  ((S1) as T1
LEFT JOIN
  (S2) as T2
ON (JOIN_COND)) WHERE T2.S2_COL_NAME IS NULL;
```

Il faut donc fournir les paramètres `JOIN_COND`, la condition de jointure et `S2_COL_NAME` qui est le nom d'une colonne de `S2`. En utilisant `LEFT JOIN` nous gardons l'ensemble des colonnes de `S1`. De plus, toutes les lignes de `S1` qui ne correspondent pas à une ligne de `S2` possèdent alors les colonnes de `S2` avec des `NULL`. Il suffit alors de garder seulement ces lignes avec les `NULL` qui ne sont donc pas présentes dans `S2`. A condition, bien sûr que `la condition` de jointure soit unique c'est à dire qu'une seule paire soit possible.

3.4.2 Requête finale

La requête transformée donne :

```
SELECT T1.* FROM
  ((SELECT * FROM Joueurs) as T1
LEFT JOIN
  (SELECT Joueurs.*, COUNT(*) as NbParties FROM
    (Joueurs NATURAL JOIN PartieParticipation)
    GROUP BY NoJoueur) as T2
ON (T1.NoJoueur = T2.NoJoueur)) WHERE T2.NoJoueur IS NULL;
```

`T1` correspond à l'ensemble des joueurs. `T2` correspond à l'ensemble des joueurs ayant fait une partie. On établit une jointure à gauche sur `T1` avec `T2` de sorte de garder l'ensemble des joueurs. Néanmoins il est stipulé dans la clause du `WHERE` que l'on garde seulement les lignes où la clé primaire de la seconde table est `NULL`, c'est à dire les lignes présentes dans `T1` mais non présentes dans `T2`.

3.5 Trigger sur la délétion d'un joueur

Nous avons fait de nombreux triggers lors de la suppression d'un élément notamment afin de conserver la cohérence, nous n'exposerons ici qu'un seul de ces triggers.

```
CREATE TRIGGER joueur_suppression
BEFORE DELETE
ON Joueurs FOR EACH ROW
BEGIN
  -- On met à jour Parties
  UPDATE Parties SET
      NoJoueurGagnant = 1
  WHERE NoJoueurGagnant = OLD.NoJoueur;
  UPDATE Parties SET
      NoJoueurPerdant = 1
  WHERE NoJoueurPerdant = OLD.NoJoueur;
  -- On met à jour PartieParticipation
  UPDATE PartieParticipation SET
    NoJoueur = 1
  WHERE NoJoueur = OLD.NoJoueur;
  -- On supprime les liens avec les decks
  DELETE FROM AvoirDeck WHERE NoJoueur = OLD.NoJoueur;
  -- On supprime les liens avec obtentions
  DELETE FROM Obtentions WHERE NoJoueur = OLD.NoJoueur;
END
```

On souhaite conserver la cohérence lorsqu'on supprime un joueur, on ne souhaite donc pas voir apparaître une clé étrangère NoJoueur ne correspondant à aucun joueur donc on ajoute un trigger avant la délétion de celui-ci. Lorsqu'on supprime un Joueur on a deux choix, on supprime tout ce à quoi il est lié en cascade, ou on pointe sur un joueur spécial "Joueur Supprimé". Nous faisons un mélange des deux. On décide de supprimer tous les liens avec decks, obtentions et cartes. Néanmoins, on désir garder l'historique des parties, en effet une partie

se déroule à deux, et nous pensons que ce serait dommage pour l'autre joueur de perdre son historique. C'est pourquoi nous mettons à jour les parties et les participations aux parties du joueur supprimé vers le joueur spécial "Joueur supprimé".

3.6 Trigger d'ajout de lien Deck-Joueur

Ce trigger se déclenche avant une insertion sur AvoirDeck et déclenche une exception si le lien n'est pas valide. Le code du trigger est exportée dans une procédure afin de factoriser le code :

```
CREATE TRIGGER new_avoir_deck
BEFORE INSERT
ON AvoirDeck FOR EACH ROW
BEGIN
  DECLARE exist INT;
  CALL joueur_a_pas_carte_decks(New.NoDeck, New.NoJoueur, @exist);
  IF exist > 0 THEN
    signal sqlstate '45000';
  END IF;
END
```

Nous explicitons donc le code de la procédure :

```
CREATE PROCEDURE joueur_a_pas_carte_decks(IN MyNoDeck INT, IN MyNoJoueur INT, OUT MyOutput
BEGIN
  SELECT COUNT(*) INTO MyOutput FROM
    ((SELECT NoCarte, NbExemplaires FROM DeckComposition WHERE NoDeck = MyNoDeck) as T1
      LEFT JOIN
    (SELECT NoCarte, SUM(NbExemplaires) as Possedes FROM
      ((SELECT NoJoueur, NoObtention FROM Obtentions WHERE NoJoueur = MyNoJoueur) as T2
      NATURAL JOIN Possessions)
      GROUP BY NoCarte
    ) as T3
    ON (T1.NoCarte = T3.NoCarte))
    WHERE T3.NoCarte IS NULL OR NbExemplaires > Possedes;
END
```

Concrètement nous retournons dans la variable représentée par `MyOutput` le nombre de cartes dont le joueur n'a pas assez d'exemplaires. En premier lieu, expliquons la requête interne correspondant à `T1` : on fait la liste des cartes présentes dans le deck dont il est question. En second lieu, la requête interne correspondant à `T3`, elle calcule les cartes posséder par le joueur en question, et le nombre d'exemplaires de chacune de ces cartes. Finalement, on fait une jointure externe sur `T1` depuis `T3`. On veut ne garder que les cartes dont le nombre d'exemplaires est insuffisant, c'est pourquoi on ajoute la condition si le nombre d'exemplaires nécessaire est supérieur à celui possédé ou si le joueur n'a pas cette carte qui correspond au cas où `T3.NoCarte` vaut `NULL`.

3.7 Liste exhaustive des triggers

Voici la liste exhaustive des triggers misent en places :

— suppression d'une obtention : les associations possessions liées sont supprimées.
— suppression d'une version : les associations pointent vers la version spéciale "version supprimée"
— suppression d'une carte : les associations Version-Cartes et Deck-Composition sont supprimées, mais les associations dans la table Obtentions pointent vers la carte spéciale "Carte supprimée"
— suppression d'un deck : les associations dans les tables AvoirDeck et DeckComposition sont supprimées et les associations dans la table PartieParticipation pointent vers "Deck Supprimé"
— suppression d'un joueur : les associations dans les tables Parties, PartieParticipation pointent vers le "Joueur supprimé" tandis que le reste des associations sont éliminées.
— suppression d'une partie : toutes les associations dans la table PartieParticipation sont supprimées.
— suppression d'une possession : les associations dans AvoirDeck avec les decks dont le joueur ne possède plus assez d'exemplaires de la carte qui vient d'être éliminée sont supprimées.
— mise à jour d'une possession : les associations dans AvoirDeck avec les decks dont le joueur ne possède plus assez d'exemplaires de la carte qui vient d'être éliminée sont supprimées.
— mise à jour dans DeckComposition : les associations dans AvoirDeck avec les les joueurs qui ne possèdent pas assez d'exemplaires de la carte mise à jour sont éliminées.
— ajout dans DeckComposition : les associations dans AvoirDeck avec les les joueurs qui ne possèdent pas assez d'exemplaires de la carte mise à jour sont éliminées.
— ajout dans Obtentions : une erreur est déclenchée si la la carte n'existe pas dans le version ajoutée.
— une nouvelle ou une mise à jour de l'association dans AvoirDeck : on vérifie que le joueur a toutes les cartes du deck sinon on lève une exception

L'ensemble de ces triggers permet d'imposer l'ensemble des contraintes liées au modèle que nous avons choisis. Néanmoins, il est important de noté que dans les cas où on lève des exceptions, cela revient à interdire l'opération et à laisser la gestion de l'opération du côté applicatif. Le SQL n'est pas tout puissant.

4 Application

Au niveau applicatif, nous étions soucieux de séparer communication avec le SGBD et interface. Cela permet de gérer la communication avec le SGBD de façon indépendante et de fournir une couche d'abstraction par rapport à la a communication à celle-ci. Nous avons donc le 'backend' qui est un serveur Node.js qui communique avec le serveur MySQL, et qui offre une interface d'accès avec une API REST sur le port 3000. Puis nous avons le 'frontend' qui est un serveur web Node.js bâti avec le framework Vue.js qui est l'interface client sur le port 8080. Il communique par l'intermédiaire de l'API REST fourni par le backend.

En d'autres termes, le 'backend' répond à des requêtes HTTP par du JSON en fonction de la route demandée. Par exemple, lorsqu'il reçoit une requête

GET sur /cards/all, le 'backend' lance la requête SQL et une fois qu'il obtient la réponse renvoie le code 200 et dans ce cas la liste de toutes les cartes.

5 Conclusion

Nous avons pu constater que nous avons longtemps tourné autour du problème en revenant presque à notre solution initiale de modèle conceptuel. La transformation en modèle relationnel ne nous a pas posé problème. D'un autre coté la mise en place de l'ensemble des contraintes que l'on veut imposer dans le SGBD c'est trouvé plus complexe. En effet, tout n'est pas possible et certains cas sont reportés à l'application.

Il est donc important d'établir un bon modèle en amont et d'assurer le plus de contraintes afin de distinguer la partie gérée par le SGBD de la partie qui devra être gérée par la partie applicative, ce qui pourrait influencer le design logiciel.