

**CENTRALE  
LYON**

ÉCOLE CENTRALE LYON

UE MOD  
SYSTÈMES DE BASES DE DONNÉES  
RAPPORT

---

## Gestion des rallyes sportifs

---

*Élèves :*  
Clément CHENU  
Hugo EBERSCHWEILER

*Enseignant :*  
Mohsen ARDABILIAN

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Conception du modèle de données</b>	<b>2</b>
2.1	Diagramme entité-association . . . . .	2
2.2	Définition des types d'entité . . . . .	4
2.3	Définition des associations . . . . .	6
2.4	Contraintes d'intégrité . . . . .	7
<b>3</b>	<b>Implantation et alimentation de la base de données</b>	<b>8</b>
3.1	Schéma relationnel et création de la base . . . . .	8
3.2	Alimentation de la base de données . . . . .	11
3.2.1	Communication avec la base de données . . . . .	11
3.2.2	Génération des données . . . . .	12
<b>4</b>	<b>Conclusion</b>	<b>13</b>

# 1 Introduction

Ce document présente le rapport du projet du MOD Systèmes de gestion de bases de données. Ce projet a pour objectif de concevoir, d'implanter et d'alimenter une base de données, ainsi que de faire une interface afin de communiquer avec cette base.

Le sujet que nous avons choisi traite des rallyes sportifs. Il s'agit donc de faire une base de données contenant l'ensemble des informations nécessaires à la gestion de ces rallyes. Ce rapport s'intéresse à l'analyse et à la conception de cette base de données, à son implantation ainsi que son alimentation.

La conception a été réalisée avec le logiciel DB-Main. La base de données est ensuite implémentée en PostgreSQL sur la plateforme cloud Neon. Nous la remplissons ensuite en utilisant des scripts Python, avec des données majoritairement générées aléatoirement.

# 2 Conception du modèle de données

Lors de la construction d'une base de données, il est intéressant de découper la tâche en plusieurs étapes. Après avoir lu et compris l'ensemble des besoins utilisateurs (décris dans le sujet PDF du projet), nous avons établi un schéma conceptuel et avons construit un diagramme entité-association.

## 2.1 Diagramme entité-association

Dans cette phase, nous avons réalisé une modélisation entité-association du système étudié avec le logiciel DB-Main. L'objectif d'un tel diagramme est d'avoir une meilleure compréhension du problème et de réaliser une conception progressive. On représente ainsi chacune des entités avec leurs attributs et les associations existantes entre les entités.

Nous avons décidé de réaliser la base de données en anglais. Les scripts seront aussi réalisés en anglais par souci de cohérence. Néanmoins, la base de données étant destinée à un public francophone, les données (ainsi que l'interface) sont en français. Il a de plus été décidé de garder les noms des types d'entité au singulier.

Voici le diagramme entité-association que nous avons construit :

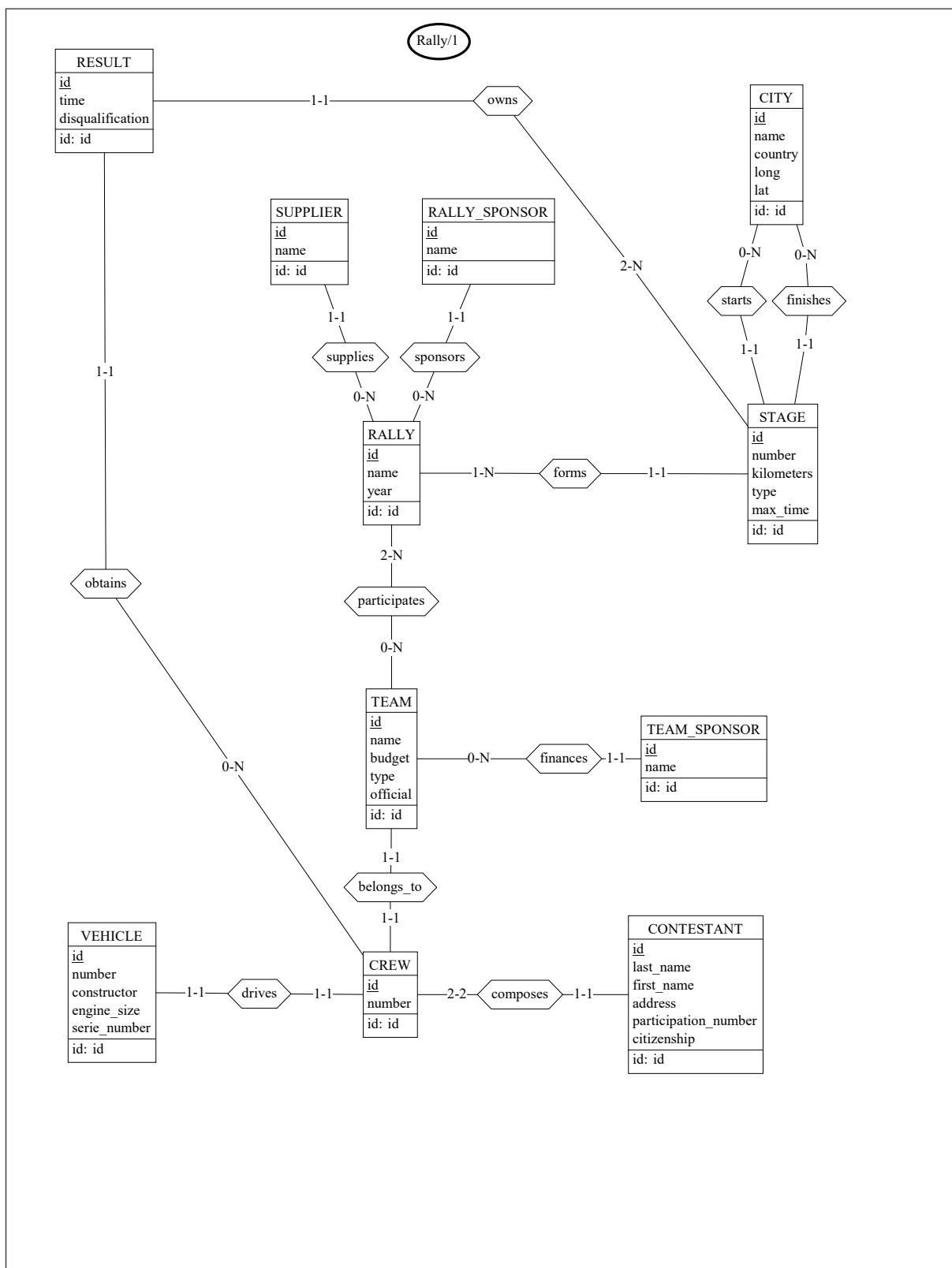


FIGURE 1 – Diagramme Entité-Association

## 2.2 Définition des types d'entité

Les types d'entité sont déduits du sujet, certains choix ou certaines interprétations ont pu être faits et seront explicités. Nous obtenons ainsi onze types d'entité. Il a été décidé de constamment attribuer une clef primaire *id* de façon à éviter les clefs primaires composées, nécessaires dans la majorité des cas, et ainsi simplifier les jointures. L'ensemble des types d'entité vont être présentés dans l'ordre alphabétique.

- L'entité *CITY* représente une ville associée au déroulement d'une étape, notamment comme ville de départ ou d'arrivée.
  - *id* : Clef primaire de l'entité. Cela ne peut pas être le nom car deux villes peuvent avoir le même nom ;
  - *name* : Nom de la ville en français ;
  - *country* : Nom du pays en français ;
  - *long* : Longitude de la ville en degrés ;
  - *lat* : Latitude de la ville en degrés.
- L'entité *CONTESTANT* représente un concurrent de rallyes.
  - *id* : Clef primaire de l'entité ;
  - *last\_name* : Nom de famille du concurrent ;
  - *first\_name* : Prénom du concurrent ;
  - *address* : Adresse postale du concurrent ;
  - *participation\_number* : Nombre de participations à des rallyes du concurrent ;
  - *citizenship* : Citoyenneté du concurrent. Donnée en français, au masculin.
- L'entité *CREW* représente un équipage constitué de deux concurrents.
  - *id* : Clef primaire de l'entité ;
  - *number* : Numéro de l'équipage donné par l'organisation. Nous avons considéré qu'un tel numéro pouvait être donné une nouvelle fois à un autre équipage.
- L'entité *RALLY* représente une édition d'un rallye.
  - *id* : Clef primaire de l'entité ;
  - *name* : Nom du rallye ;
  - *year* : Année de cette édition.
- L'entité *RALLY\_SPONSOR* représente un sponsor d'une édition d'un rallye.
  - *id* : Clef primaire de l'entité ;
  - *name* : Nom du sponsor. Ce nom n'est pas nécessairement unique.
- L'entité *RESULT* représente un résultat d'un équipage pour une étape donnée.
  - *id* : Clef primaire de l'entité ;
  - *time* : Temps obtenu par l'équipe en secondes. Il peut valoir 0 si l'équipage n'a pas fini l'étape ;

- *disqualification* : Indique si l'équipage a été disqualifié lors de l'étape ou non. Cela peut être dû au fait de ne pas finir, ou de ne pas finir dans les temps pour les étapes spéciales.
- L'entité *STAGE* représente une étape d'une édition d'un rallye.
- *id* : Clef primaire de l'entité ;
  - *number* : Numéro de l'étape. Si l'étape est un prologue, le numéro vaut alors 0 ;
  - *kilometers* : Nombre de kilomètres dans l'étape ;
  - *type* : Indique s'il s'agit d'une spéciale (avec un temps à respecter) ou d'une étape de liaison (sans temps limite) ;
  - *max\_time* : Dans le cas d'une spéciale, temps maximal à respecter en secondes. Sinon, vaut 0.
- L'entité *SUPPLIER* représente un fournisseur d'une édition d'un rallye.
- *id* : Clef primaire de l'entité ;
  - *name* : Nom du fournisseur. Ce nom n'est pas nécessairement unique.
- L'entité *TEAM* représente une équipe de rallye.
- *id* : Clef primaire de l'entité ;
  - *name* : Nom de l'équipe. Ce nom n'est pas nécessairement unique ;
  - *budget* : Budget de l'équipe en euros ;
  - *type* : Catégorie de véhicule dans laquelle l'équipe participe (automobile, moto, camion) ;
  - *official* : Indique si l'équipe est officielle (représente une seule marque) ou si c'est une association (plusieurs marques).
- L'entité *TEAM\_SPONSOR* représente un sponsor d'une équipe.
- *id* : Clef primaire de l'entité ;
  - *name* : Nom du sponsor. Ce nom n'est pas nécessairement unique.
- L'entité *VEHICLE* représente un véhicule conduit par un équipage.
- *id* : Clef primaire de l'entité ;
  - *number* : Numéro du véhicule donné par l'organisation. Nous avons considéré qu'un tel numéro pouvait être donné une nouvelle fois à un autre véhicule ;
  - *constructor* : Marque du constructeur du véhicule ;
  - *engine\_size* : Cylindrée du moteur en cm<sup>3</sup> ;
  - *serie\_number* : Numéro de série du véhicule.

D'autres choix de conceptions intéressants auraient pu être retenus. Notamment un type d'entité décrivant une compétition en général et non pas directement une édition précise. Nous aurions aussi pu faire un type d'entité pour les pays. Ces deux types n'ont pas

été retenus par souci de simplicité, la base de données n'étant de toute façon pas conséquente. Les entités *SUPPLIER*, *RALLY\_SPONSOR* et *TEAM\_SPONSOR* auraient pu hériter d'un même type d'entité décrivant les partenaires. Enfin, nous aurions pu séparer les étapes spéciales et de liaisons en deux sous-types d'entité, mais cela aurait complexifié la base pour un seul attribut qui n'est pas commun aux deux.

### 2.3 Définition des associations

Nous allons maintenant détailler les deux associations présentes dans le schéma conceptuel. Elles sont à nouveau présentées dans l'ordre alphabétique.

- L'association *belongs\_to* relie *CREW* à *TEAM*. Elle représente le fait qu'un équipage appartient à une équipe.
- L'association *composes* relie *CONTESTANT* à *CREW*. Elle montre qu'il faut deux concurrents pour composer un équipage.
- L'association *drives* relie *CREW* à *VEHICLE*. Elle indique qu'un équipage conduit une voiture.
- L'association *finances* relie *TEAM\_SPONSOR* à *TEAM*. Elle représente le financement d'une équipe par des sponsors.
- L'association *finishes* relie *CITY* à *STAGE*. Elle indique qu'une ville est la ville d'arrivée d'une étape.
- L'association *forms* relie *STAGE* à *RALLY*. Elle montre qu'une édition d'un rallye est formée d'une ou plusieurs étapes.
- L'association *obtains* relie *CREW* à *RESULT*. Elle indique qu'un équipage obtient des résultats. Nous autorisons les équipes à n'avoir aucun résultat, cela peut être utile si un rallye n'a pas encore eu lieu mais que les équipages sont déjà connus.
- L'association *owns* relie *STAGE* à *RESULT*. Elle montre qu'une étape a plusieurs résultats.
- L'association *participates* relie *TEAM* à *RALLY*. Elle représente le fait qu'une équipe peut participer à des rallyes. Il s'agit d'une relation N-N, une table intermédiaire devra donc être créée pour représenter cette association.
- L'association *sponsors* relie *RALLY\_SPONSOR* à *RALLY*. Elle indique les sponsors d'une édition de rallye.
- L'association *starts* relie *CITY* à *STAGE*. Elle indique qu'une ville est la ville de départ d'une étape.

- L'association *supplies* relie *SUPPLIER* à *RALLY*. Elle donne le fait qu'un rallye peut avoir un ou plusieurs fournisseurs.

Certaines conceptions auraient pu être différentes. Notamment, les associations *starts* et *finishes*, reliant respectivement les villes de départ et d'arrivée aux étapes, auraient pu être regroupées en une seule association avec un attribut indiquant s'il s'agit du départ ou de l'arrivée. La solution actuelle a été choisie car elle nous paraît plus lisible.

## 2.4 Contraintes d'intégrité

Nous définissons ensuite des contraintes d'intégrité destinées à vérifier que chaque modification est conforme à ce que nous attendons. Certaines contraintes d'intégrité sont standards, d'autres sont personnalisées pour répondre précisément à notre cahier des charges.

Tout d'abord, nous avons les contraintes d'unicité. Ce type de contrainte permet d'obliger chaque entité d'un type d'avoir une valeur différente pour un attribut donné. Une clef primaire entraîne directement une contrainte d'unicité. Dans notre modélisation, l'attribut *id* de chaque type d'entité est la clef primaire et donc possède une contrainte d'unicité. Nous n'avons pas défini d'autres contraintes d'unicité, même pour des numéros uniques données par les organisateurs. En effet, nous avons considéré que lors de deux rallyes différents, un tel numéro pourrait potentiellement être redonné.

Nous avons ensuite les contraintes référentielles. Une telle contrainte garantit qu'une valeur située dans une table correspond bien à une autre valeur d'une colonne et d'une table donnée. Les clefs étrangères entraînent directement une contrainte référentielle. Ainsi, toutes nos clefs étrangères, que l'on nommera du type *id\_*, créent des contraintes référentielles.

Le dernier type de contrainte standard est la colonne obligatoire. Elle oblige chaque ligne à avoir une valeur dans une colonne donnée. Une clef primaire entraîne directement cette contrainte. Par défaut, DB-Main met cette contrainte sur l'ensemble des colonnes, nous les avons laissées. Ce comportement n'est pas gênant car l'ensemble des colonnes doivent effectivement avoir des données. Néanmoins, nous aurions pu laisser davantage de flexibilité notamment si l'utilisateur veut rentrer des données en avance, par exemple un équipage (*CREW*) pour lequel le numéro attribué par l'organisateur (*number*) n'a pas encore été donné.

Enfin, nous ajoutons des contraintes de domaine, qui servent à limiter les valeurs attribuables à une colonne. Ainsi, la colonne *TEAM.type* peut seulement prendre les valeurs *car*, *motorbike* et *truck*. La colonne *STAGE.type* peut seulement valoir *linking* ou *special*. Les colonnes *CITY.long* et *CITY.lat* doivent être comprises respectivement entre -180 et 180 et entre -90 et 90. Les colonnes *STAGE.number*, *STAGE.kilometers*, *STAGE.max\_time*, *RESULT.time*, *RALLY.year*, *TEAM.budget*, *VEHICLE.engine\_size* et *CONTESTANT.participation\_number* doivent être positives ou nulles.

## 3 Implantation et alimentation de la base de données

### 3.1 Schéma relationnel et création de la base

L'étape suivante consiste à la création du schéma relationnel qui permet de décrire formellement la structure de la base de données. Elle donne ainsi chaque table avec les attributs, les clefs primaires et les clefs étrangères. À cette étape, nous mettons les noms des tables en minuscule. En effet, PostgreSQL a un comportement différent avec les majuscules et les minuscules, ce changement permet donc d'éviter les problèmes. DB-Main fournit un bouton qui nous permet d'obtenir directement le schéma suivant :

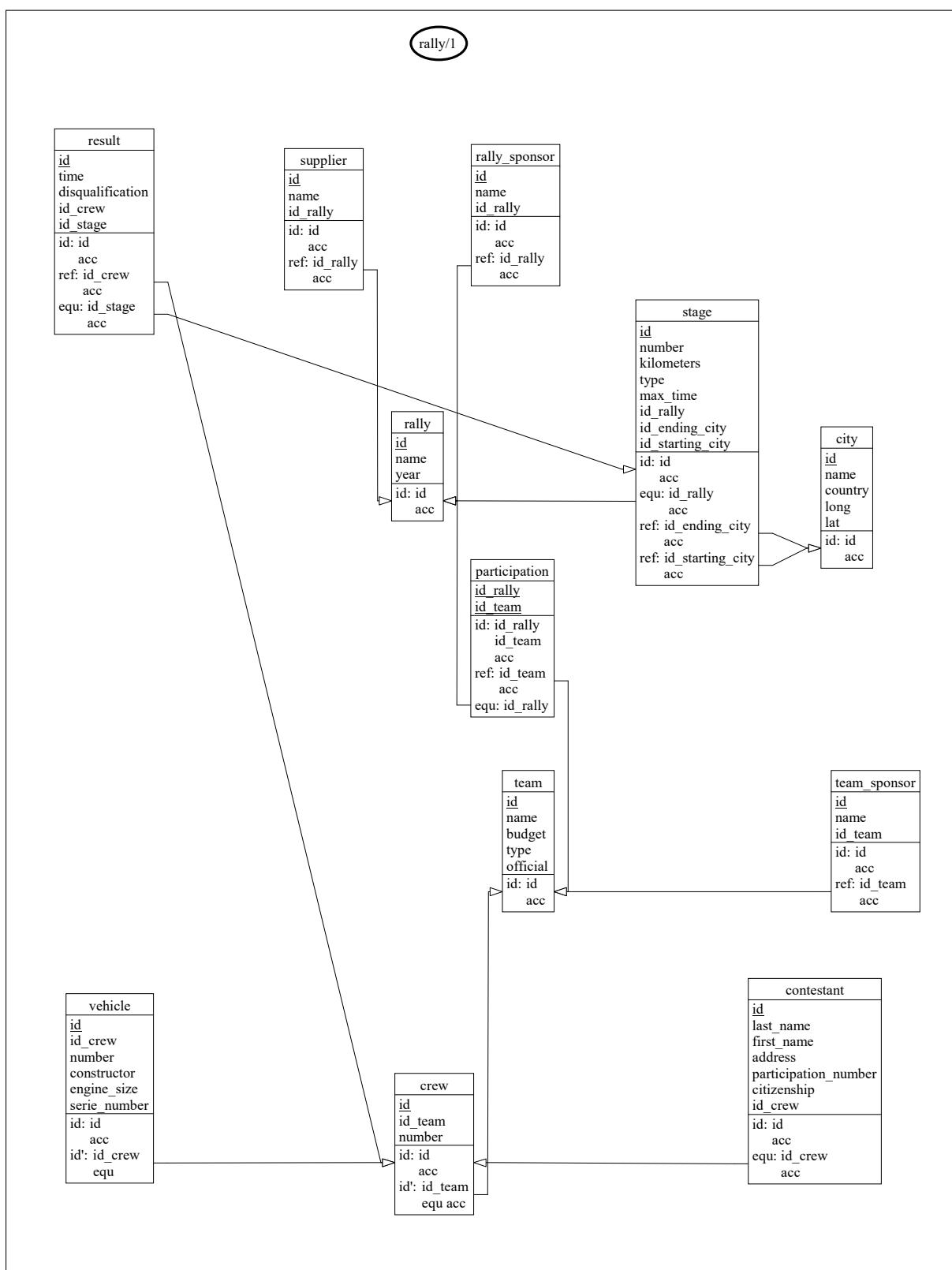


FIGURE 2 – Schéma relationnel

Le logiciel DB-Main permet aussi de traduire ce schéma en fichier DDL de façon à créer une base de données en PostgreSQL. Nous utilisons donc cette fonctionnalité afin d'obtenir un premier fichier. Certaines modifications doivent être apportées au fichier,

notamment pour les types. Ainsi, *varchar(1)* doit être modifié en *text*, *float(1)* en *real*, *numeric(1)* en *integer*. Nous ajoutons à ce fichier les contraintes de domaine ainsi que des vues ayant pour objectif de faciliter notre interface. Nous allons détailler quelques exemples de requête de création du fichier DDL, l'ensemble de ces requêtes sont disponibles dans le fichier [database\\_creation.ddl](#) disponible sur Github.

Afin de créer une table, nous utilisons la commande *CREATE TABLE*, puis nous indiquons chaque nom de colonne avec le type. Pour l'identifiant, nous utilisons le type *serial* qui permet d'avoir un entier qui s'incrémente automatiquement à chaque ligne. Nous pouvons rendre les colonnes obligatoires grâce à *NOT NULL* puis créer une contrainte de clef primaire. Selon la cardinalité des relations, on peut avoir unicité de la clef étrangère, dans ce cas là on peut rajouter la contrainte d'unicité grâce au mot-clef *UNIQUE*. Voici un exemple pour la création de la table *crew*.

```
CREATE TABLE crew (
    id serial NOT NULL,
    id_team integer NOT NULL,
    number integer NOT NULL,
    CONSTRAINT ID_CREW_ID PRIMARY KEY (id),
    CONSTRAINT FKappartient_ID UNIQUE (id_team));
```

Nous ajoutons ensuite les différentes contraintes référentielles. Pour ce faire, nous modifions une table grâce à la commande *ALTER TABLE* puis *ADD CONSTRAINT*, nous ajoutons une contrainte qui correspond à une clef étrangère avec une référence à une autre table. Cela crée alors la référence entre la colonne de la table modifiée et la clef primaire de l'autre table. Voici un exemple pour la clef étrangère *id\_team* présente dans la table *crew*.

```
ALTER TABLE crew ADD CONSTRAINT FKappartient_FK
    FOREIGN KEY (id_team)
    REFERENCES team;
```

Nous créons ensuite des index. Les index sont très importants pour les grosses bases de données, ils permettent de trouver rapidement des lignes, sans avoir besoin de scanner toute la table. Notre base de données étant de taille raisonnable, ils ne sont pas indispensables, mais nous en ajoutons tout de même sur les clefs étrangères. Voici un exemple avec la clef étrangère *id\_crew* de la table *contestant*.

```
CREATE INDEX FKcompose_IND
    ON contestant (id_crew);
```

Nous ajoutons maintenant nos contraintes de domaine. Cela se fait une nouvelle fois avec *ALTER TABLE* et *ADD CONSTRAINT*, mais en ajoutant un *CHECK* suivi de la condition. Voici trois exemples, le premier montre comment limiter à certaines valeurs précises, le deuxième comment faire un intervalle et le troisième comment avec des valeurs positives ou nulles.

```
ALTER TABLE stage
    ADD CONSTRAINT stage_type_check
```

```

CHECK (type IN ('linking', 'special'));

ALTER TABLE city
ADD CONSTRAINT city_lat_long_check
CHECK (
    lat BETWEEN -90 AND 90
    AND long BETWEEN -180 AND 180
);

ALTER TABLE stage
ADD CONSTRAINT stage_number_check
CHECK (number >= 0);

```

Enfin, nous ajoutons les vues qui sont des requêtes sauvegardées sous un nom. Cela nous permet ensuite dans notre interface de sélectionner directement les colonnes sans avoir besoin de réécrire la commande complète avec les jointures. Cela se fait avec la commande *CREATE VIEW*. L'exemple ci-dessous crée une vue *team\_info* qui nous permet d'obtenir toutes les informations relatives à une équipe. Ainsi, il s'agit d'une jointure entre les tables *team*, *crew* et *vehicle* et cela récupère ce dont nous avons besoin pour notre interface.

```

CREATE VIEW team_info AS
SELECT team.name, team.budget, team.type, crew.id AS id_crew,
vehicle.constructor, vehicle.engine_size, vehicle.serie_number, team.id AS id_team
FROM team
JOIN crew ON crew.id_team = team.id
JOIN vehicle ON vehicle.id_crew = crew.id;

```

## 3.2 Alimentation de la base de données

### 3.2.1 Communication avec la base de données

Pour la communication avec la base de données, nous utilisons la bibliothèque Python *psycopg*, qui est destinée à l'interaction avec des bases PostgreSQL. De façon à simplifier les commandes, nous créons une classe *PostgreSQL* qui utilise *psycopg* pour envoyer des requêtes SQL. Cette classe se situe dans le script [db\\_communication.py](#). La classe contient une méthode générale *execute* ainsi que des méthodes pour chacune des requêtes SQL d'accès aux données (CRUD : *create*, *read*, *update*, *delete*).

La méthode *execute* prend en entrée une requête SQL et potentiellement des paramètres et l'exécute directement. Toutes les autres méthodes font directement appel à celle là. Elle permet aussi de gérer les erreurs. Ainsi, en cas de déconnexion avec le serveur, elle gère la reconnexion. Elle effectue aussi un *rollback* en cas d'erreurs, ce qui permet de ne pas bloquer la connexion.

La méthode *write* permet de créer de nouvelles données. Elle prend en entrée le nom de la table où il faut ajouter les données, ainsi que les données sous forme d'une liste de dictionnaires dont les clefs sont les noms des colonnes. Elle crée donc une requête SQL de type *INSERT*.

La méthode *read* permet la lecture des données via une commande *SELECT*. Elle prend en entrée le nom d'une table de données, une ou plusieurs colonnes et éventuellement un dictionnaire contenant des conditions. La fonction va lire les données de la table et retourner les colonnes demandées en suivant la ou les conditions spécifiées. Les conditions permettent de traduire la contrainte *WHERE* en SQL. De plus, la fonction comprend 2 autres arguments en entrée *number\_values* et *return\_type*. Ils permettent respectivement de restreindre le nombre de valeurs en sorties et de choisir le type de données que va retourner la fonction (liste, liste de dictionnaires, dictionnaire).

La méthode *update* permet de modifier des données via une requête SQL *UPDATE*. Elle prend en entrée le nom de la table, les nouvelles données sous forme d'un dictionnaire ainsi que les conditions pour que les données soient modifiées. Si aucune condition n'est spécifiée, toutes lignes sont modifiées.

La méthode *delete\_rows* permet de supprimer des lignes via la commande SQL *DELETE*. Il est nécessaire de fournir le nom de la table ainsi qu'un dictionnaire avec les conditions de suppression. Ce dictionnaire est obligatoire pour éviter les mauvaises manipulations, pour supprimer tout le contenu d'une table, il faut utiliser la méthode *delete\_all*.

### 3.2.2 Génération des données

La génération des données et le remplissage de la base s'effectue via le script Python [fill\\_db.py](#), des fonctions distinctes permettant de remplir les tables. Ce script se connecte à la base de données grâce à notre classe *PostgreSQL*, avec les droits d'écriture (contrairement à la connexion pour l'interface qui a seulement les droits de lecture). Nous remplissons ensuite la base avec des données aléatoires pour la majorité, pour les rallyes Paris-Dakar ayant eu lieu entre 1995 et 2014. L'édition de 2008 ayant été annulée, elle n'est pas présente.

Après le remplissage de la table *rally* grâce à une liste, nous remplissons la table *stage*. Cette table est basée sur des informations réelles. Ainsi, nous avons écrit dans le fichier [stages.csv](#) les différentes étapes avec leur ville de départ et leur ville d'arrivée. La bibliothèque Python *geopy* nous permet de récupérer le pays, la latitude et la longitude de chacune des villes. Certaines corrections sont faites dans le script en cas de ville ayant le même nom dans un autre pays.

Pour les autres tables, les données sont générées aléatoirement. Pour ce faire, nous utilisons la bibliothèque Python *faker* qui permet de générer des noms, des adresses, de noms d'entreprises ou même des numéros de séries. Le module par défaut *random* permet lui de générer les nombres ou de choisir les types d'étapes et d'équipes. Cela permet d'avoir de nombreuses données rapidement, néanmoins, les noms des équipes ne sont pas crédibles et le taux de disqualification que nous avons appliqué était en fait beaucoup trop élevé.

## 4 Conclusion

À travers ce rapport, nous avons pu mettre en avant plusieurs phases de la construction d'une base de données en passant par la conception, l'implémentation et l'alimentation de la base. Nous avons pu concevoir le modèle de données à partir du logiciel DB-Main, puis, nous avons implémenté la base de données via PostgreSQL. Enfin, nous avons pu remplir la base par des scripts Python. Nous avons donc bien suivi la méthodologie phare de la construction de la base de données en suivant les étapes de compréhension des besoins utilisateurs, d'analyse conceptuelle, de conception logique, de conception physique et de création du code.