

Nom: Constantine  
Prénom: Janshica  
Groupe: Erymanthe

# SAE S104 Crédit d'une base de données

Climate-Related Disaster Frequency



## Sommaire:

1. Introduction
2. Script manuel de création de la base de données
3. Modélisation et script de création “avec AGL”
4. Peuplement des tables
5. Conclusion

### 1-Introduction:

Dans le cadre de cette SAE, ce rapport présente les étapes clés de la création d'une base de données, ainsi que les outils utilisés pour y parvenir. Nous avons manipulé divers outils, tels que PostgreSQL et AGL, pour mettre en pratique les concepts fondamentaux des bases de données.

Le rapport sera divisé en trois parties : la première abordera le processus de création manuelle des tables à l'aide de scripts SQL. La deuxième portera sur la modélisation et la génération automatique de scripts grâce à AGL. Enfin, la troisième aborde le peuplement des tables avec des données provenant d'un fichier CSV.

L'objectif de cette SAE est de mettre en pratique les connaissances théoriques acquises sur la création et la gestion des bases de données

### 2- Script manuel de création de la base de données:

Cette étape consiste à réaliser manuellement un script de création de la base de données à partir du schéma relationnel présenté dans la figure 4. Ce script respecte les contraintes définies dans le modèle relationnel, telles que les clés primaires, les clés étrangères et les autres règles d'intégrité nécessaires pour garantir la cohérence des données.

Pour mieux comprendre ce script, commençons par rappeler le schéma relationnel de la figure 4 :

- **region** (region\_code, name)
- **sub\_region** (sub\_region\_code, name, region\_code) où region\_code est une clé étrangère qui fait référence au schéma de relation **region**.
- **country** (country\_code, name, ISO2, ISO3, sub\_region\_code) où sub\_region\_code est une clé étrangère qui fait référence au schéma de relation **sub\_region**.
- **disaster** (disaster\_code, disaster)
- **climate\_disaster** (country\_code, disaster\_code, year, number) où country\_code et disaster\_code sont des clés étrangères qui font respectivement référence aux schémas de relation **country** et **disaster**.

D'après ce schéma, nous devons créer cinq tables : `region`, `sub_region`, `country`, `disaster`, et `climate_disaster`. Cependant, dans notre script SQL, il est essentiel de respecter un ordre précis lors de la création des tables, en commençant par `region` et en terminant par `climate_disaster`, afin de respecter les contraintes des clés étrangères. Par exemple, la table `sub_region` ne peut être créée qu'après la table `region`, car elle contient une clé étrangère qui fait référence à cette dernière. En outre, pour que notre script soit bien structuré et facile à lire, il est important d'y inclure des commentaires explicatifs, d'utiliser des indentations correctes, et de veiller à une présentation soignée.

Voyons maintenant de plus près comment créer chaque table tout en respectant ces conditions.

#### →La table region:

Cette table contient une clé primaire: `region_code` ainsi que un attribut `name`.

```
-- Création de la table région--  
CREATE TABLE region (  
    region_code INTEGER PRIMARY KEY, -- Code unique pour identifier chaque  
    continent--  
    name VARCHAR NOT NULL;)-- Nom du continent, ne peut pas être vide--
```

#### Description/ essentiel pour bien comprendre :

L'instruction `CREATE TABLE` est utilisée pour créer une table.

Pour définir une clé primaire, on attribue à l'attribut concerné le type `INTEGER`, suivi de la contrainte `PRIMARY KEY`.

Le type `INTEGER` correspond à des entiers signés codés sur 4 octets.

Quant au type `VARCHAR`, il représente une chaîne de caractères. Cela convient parfaitement pour les noms des régions, qui doivent être des chaînes de caractères.

Dans ce script, j'ai ajouté la contrainte `NOT NULL` à l'attribut `name` afin de garantir que chaque région possède obligatoirement un nom. En effet, une région ne peut pas exister sans nom.

À noter que les `commentaires` dans le code SQL sont indiqués avec deux tirets (--) et permettent de documenter les différentes parties du script.

#### →La table, sub\_region:

Cette table contient une clé primaire : `sub_region_code`, un attribut `name`, et une clé étrangère `region_code` qui fait référence à la table `region`.

```
-- Création de la table sous-région--  
CREATE TABLE sub_region (  
    sub_region_code INTEGER PRIMARY KEY, -- Code unique pour identifier chaque  
    sous région--  
    name VARCHAR NOT NULL,-- Nom de la sous région, ne peut pas être vide--  
    region_code INTEGER REFERENCES region (region_code) -- Référence au code  
    de la région dans la table 'region'--  
);
```

### Description/ essentiel pour bien comprendre :

Pour créer une clé étrangère, on commence par attribuer le type INTEGER à l'attribut concerné, suivi de la contrainte REFERENCES <nom de la table à référencer> (<attribut de la table référencée>).

### →La table country:

Elle est composée d'une clé primaire, de trois attributs : name, ISO2, et ISO3, ainsi que d'un attribut clé étrangère qui fait référence à la table sub\_region.

```
-- Création de la table pays
CREATE TABLE country (
    country_code INTEGER PRIMARY KEY,-- Code unique pour identifier chaque
    pays--  

    name VARCHAR NOT NULL, -- Nom du pays, ne peut pas être vide--  

    ISO2 CHAR(2) NOT NULL, -- Code ISO2 (2 caractères) doit être obligatoire--  

    ISO3 CHAR(3) NOT NULL, -- Code ISO3 (3 caractères) doit être  

    obligatoire.--  

    sub_region_code INTEGER REFERENCES sub_region (sub_region_code) -- clé  

    étrangère--  

);
```

### Essentiel à retenir :

Le type **CHAR** correspond à une chaîne de caractères de longueur fixe, généralement inférieure ou égale à 255 caractères. Les attributs ISO2 et ISO3 sont de type **CHAR** car ils contiennent respectivement 2 et 3 caractères

### →La table disaster :

Elle est composée d'une clé primaire : **disaster\_code**, ainsi que d'un attribut **disaster**.

```
-- Création de la table désastre
CREATE TABLE disaster (
    disaster_code INTEGER PRIMARY KEY,-- Code unique pour identifier chaque
    pays--  

    disaster VARCHAR NOT NULL -- Nom du désastre, ne peut pas être vide--  

);
```

### →La table climate\_disaster :

Elle a une clé primaire composite composée de trois attributs : **country\_code**, **disaster\_code**, et **year**. Les attributs **country\_code** et **disaster\_code** sont des clés étrangères qui font respectivement référence aux tables **country** et **disaster**

```
-- Création de la table désastre climatique
```

```

CREATE TABLE climate_disaster (
    country_code INTEGER REFERENCES country (country_code),-- Code du pays,
    clé étrangère faisant référence à la table 'country'--
    disaster_code INTEGER REFERENCES disaster (disaster_code), -- Code du
désastre, clé étrangère faisant référence à la table 'disaster'--
    year INTEGER,-- Année du désastre--
    number INTEGER ,-- Nombre d'événements désastreux en cette année--
    PRIMARY KEY(country_code, disaster_code, year) -- Définition de la clé
 primaire composée--
);

```

### Essentiel à retenir :

Pour créer une clé composite, on utilise la contrainte PRIMARY KEY suivie des éléments qui composent la clé, entre parenthèses.

Maintenant, exécutons ce script dans notre terminal pour vérifier s'il fonctionne correctement. Pour ma part, j'utilise le terminal sous Linux ainsi qu'un SGBDR PostgreSQL.

```

constantine@constantine-25:~$ psql -U constantine -W
Password:
psql (16.6 (Ubuntu 16.6-0ubuntu0.24.04.1))
Type "help" for help.

constantine=> SELECT *FROM region;
 region_code | name
-----+-----
(0 rows)

constantine=> SELECT *FROM sub_region;
 sub_region_code | name | region_code
-----+-----+-----
(0 rows)

constantine=> SELECT *FROM country;
 country_code | name | iso2 | iso3 | sub_region_code
-----+-----+-----+-----+
(0 rows)

constantine=> SELECT *FROM disaster;
 disaster_code | disaster
-----+-----
(0 rows)

constantine=> SELECT *FROM climate_disaster;
 country_code | disaster_code | year | number
-----+-----+-----+-----
(0 rows)

constantine=>

```

```

CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE

```

Les cinq tables ont été créées avec succès. La requête

```

SELECT name
FROM <nom de la table>;

```

nous a permis d'afficher le contenu de chaque table, ce qui correspond bien à la figure 4.

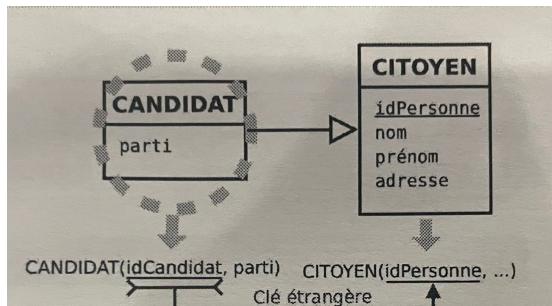
### 3- Modélisation et script de création “avec AGL”

Cette partie consiste à manipuler correctement un logiciel AGL. Sur ce logiciel, il est essentiel de savoir créer différents types de schémas, allant des plus simples, comme les associations fonctionnelles, aux plus complexes, tels que les associations maillées. Pour cette SAE, j'ai choisi d'utiliser pgModeler pour plusieurs raisons.

Tout d'abord, lors de mes recherches, je me suis rendu compte qu'il était simple à manipuler et qu'il est compatible avec Linux, le système que j'utilise. De plus, pgModeler est conçu spécifiquement pour PostgreSQL, ce qui permet de générer des scripts SQL très précis pour ce SGBD.

La première étape consiste à reproduire un schéma d'association fonctionnelle, tiré du cours, dans pgModeler. Une fois le schéma créé, il sera nécessaire de le comparer avec celui fourni dans le cours afin d'identifier les éventuelles différences.

Pour ma part je choisi ce schéma d'association fonctionnelle vue en cours:



Pour créer ce modèle dans pgModeler, il faut d'abord définir deux tables correspondant aux entités Citoyen et Candidat. Ensuite, dans chaque table, on crée les colonnes correspondant aux attributs du schéma.

### Table Citoyen

- idPersonne : type `integer`
- nom : type `varchar`
- prenom : type `varchar`
- adresse : type `varchar`

### Table Candidat

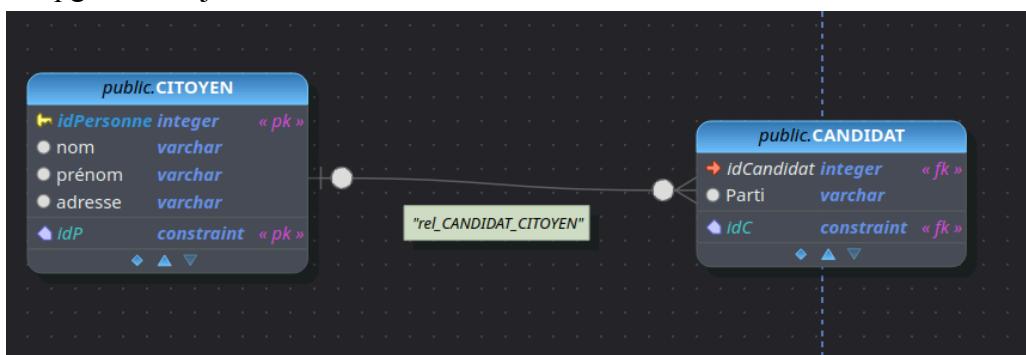
- idCandidat : type `integer`
- parti : type `varchar`

Dans pgModeler, il faut préciser les types de données pour chaque colonne, comme dans un script SQL classique. Une fois les colonnes créées, il faut ajouter les contraintes nécessaires :

1. Clé primaire :
  - Pour la table Candidat, définir une contrainte `PRIMARY KEY` sur la colonne `idCandidat`.
  - Pour la table Citoyen, définir une contrainte `PRIMARY KEY` sur la colonne `idPersonne`.
2. Clé étrangère :
  - Ajouter une contrainte `FOREIGN KEY` dans la table Candidat sur la colonne `idPersonne`, qui fera référence à la colonne `idPersonne` de la table Citoyen.

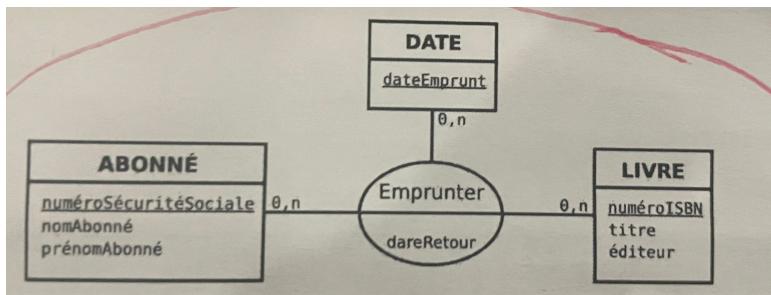
La clé étrangère permettra d'établir une relation entre les deux tables et d'afficher cette relation visuellement dans pgModeler.

Sur pgModeler j'obtiens cette modélisation:



Nous allons maintenant passer au schéma association maillé.

Voici le schéma association tiré du cours que je vais représenter dans pgModeler.



Pour modéliser ce type de schéma, il faut créer quatre tables : Date, Livre, Abonné, et Emprunter.

Chaque table doit contenir ses colonnes et contraintes spécifiques. Cependant, une

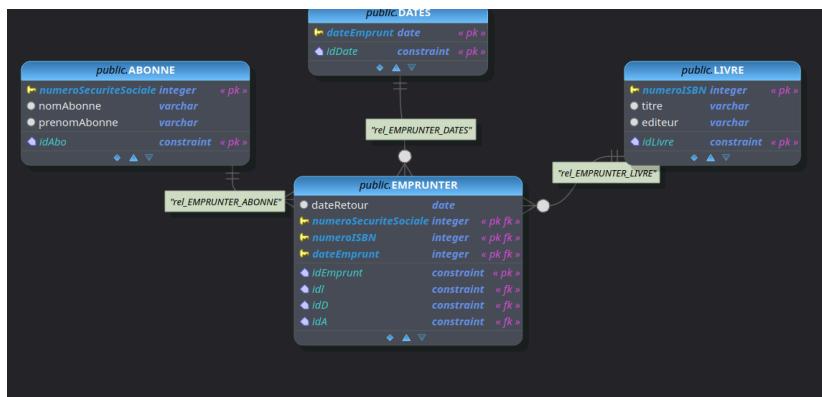
particularité existe dans le cas d'un schéma maillé : la table Emprunter, qui relie les trois autres tables (Date, Livre, et Abonné), doit comporter une clé composite.

Cette clé composite est composée des colonnes suivantes :

- **dateEmprunt**, qui est une clé étrangère référencée depuis la table Date.
- **numeroISBN**, qui est une clé étrangère référencée depuis la table Livre.
- **numeroSecuriteSociale**, qui est une clé étrangère référencée depuis la table Abonné.

Ainsi, la table Emprunter joue un rôle central en établissant la relation entre les trois autres tables, tout en garantissant l'intégrité référentielle grâce aux clés étrangères et à la clé composite.

Voici le rendu sur pg modeler:



Lorsqu'on compare un modèle AGL avec un modèle Entité-Association, plusieurs différences majeures peuvent être observées.

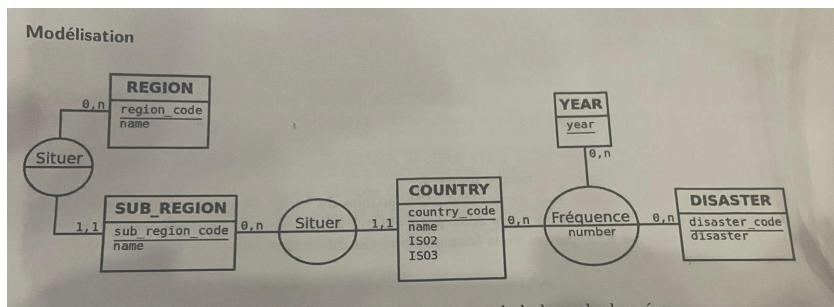
Tout d'abord, dans un modèle AGL, les entités du modèle Entité-Association sont transformées en tables. Les attributs de ces entités deviennent des colonnes dans les tables correspondantes. De plus, le modèle AGL précise pour chaque colonne le type de données ce qui n'est pas le cas dans le modèle Entité-Association, où ces informations ne sont pas explicitement définies.

Ensuite dans un modèle AGL, les clés primaires et les clés étrangères sont clairement identifiées et mises en évidence dans la représentation. À l'inverse, dans un modèle Entité-Association, les clés étrangères ne sont pas explicitement représentées.

Puis, dans un modèle Entité-Association, les relations sont présentées sous forme de liens entre des entités, souvent accompagnées de cardinalités pour indiquer la nature de ces relations. Cependant, ces relations restent abstraites. Dans un modèle AGL, les relations sont concrétisées par des clés étrangères, qui matérialisent les liens entre les tables de manière explicite et technique.

Enfin, le modèle AGL, plus détaillé, est directement exploitable pour créer et gérer une base de données. Il précise les types de données, les contraintes et la structure relationnelle. En revanche, le modèle Entité-Association reste conceptuel, utile pour la conception, mais nécessite une traduction avant une utilisation technique.

La prochaine étape consiste à créer le schéma de la figure 4:



Pour modéliser ce schéma dans pgModeler, nous suivons une approche similaire à celle décrite précédemment. Tout d'abord, nous divisons le modèle en deux schémas d'association fonctionnelle et un schéma maillé :

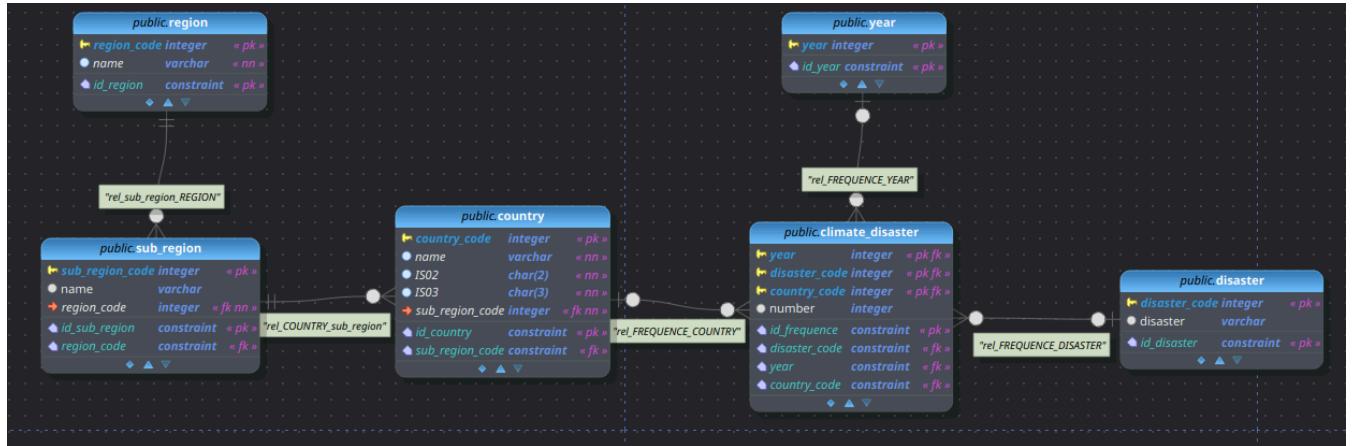
1. Schémas d'association fonctionnelle :
  - region et sub\_Région.
  - country et sub\_région.
2. Schéma maillé :
  - Les tables Country, Year, et Disaster sont reliées par une table d'association que je renomme climate\_disaster pour garantir une meilleure cohérence dans la suite de mon travail.

Pour les schémas d'association fonctionnelle, la démarche reste similaire à celle expliquée précédemment. Cependant, ici, une particularité existe au niveau des cardinalités.

Entre `region` et `sub_region`, ainsi que `Country` et `sub_region`, les cardinalités sont respectivement 1,1 et 0,n. Cela implique que dans pgModeler, les colonnes correspondant aux clés étrangères présentes dans les tables `Sub_Region` et `Country` doivent être définies comme NOT NULL pour respecter les contraintes de cardinalité.

Enfin, pour le schéma maillé, nous utilisons les mêmes principes décrits plus tôt, en définissant une clé composite et des relations entre les tables impliquées.

Voici le modèle physique de données réalisé avec pgmodeler:



Notre prochaine étape consiste à générer automatiquement le script SQL pour ce modèle grâce à pgModeler. C'est assez simple, il suffit tout simplement d'exporter le script SQL.

Voici le script produit par pgmodeler:

```
-- object: new_database | type: DATABASE --
-- DROP DATABASE IF EXISTS new_database;
CREATE DATABASE new_database;
-- ddl-end --
-- object: public.region | type: TABLE --
-- DROP TABLE IF EXISTS public.region CASCADE;
CREATE TABLE public.region (
    region_code integer NOT NULL,
    name varchar NOT NULL,
    CONSTRAINT id_region PRIMARY KEY (region_code)
);
-- ddl-end --
ALTER TABLE public.region OWNER TO postgres;
-- ddl-end --

-- object: public.sub_region | type: TABLE --
-- DROP TABLE IF EXISTS public.sub_region CASCADE;
CREATE TABLE public.sub_region (
    sub_region_code integer NOT NULL,
    name varchar,
    region_code integer NOT NULL,
    CONSTRAINT id_sub_region PRIMARY KEY (sub_region_code)
);
```

```

ALTER TABLE public.region OWNER TO postgres;
-- ddl-end --

-- object: public.sub_region | type: TABLE --
-- DROP TABLE IF EXISTS public.sub_region CASCADE;
CREATE TABLE public.sub_region (
    sub_region_code integer NOT NULL,
    name varchar,
    region_code integer NOT NULL,
    CONSTRAINT id_sub_region PRIMARY KEY (sub_region_code)
);
-- ddl-end --
ALTER TABLE public.sub_region OWNER TO postgres;
-- ddl-end --

-- object: public.country | type: TABLE --
-- DROP TABLE IF EXISTS public.country CASCADE;
CREATE TABLE public.country (
    country_code integer NOT NULL,
    name varchar NOT NULL,
    "IS02" char(2) NOT NULL,
    "IS03" char(3) NOT NULL,
    sub_region_code integer NOT NULL,
    CONSTRAINT id_country PRIMARY KEY (country_code)
);
-- ddl-end --
ALTER TABLE public.country OWNER TO postgres;
-- ddl-end --

-- object: public.year | type: TABLE --
-- DROP TABLE IF EXISTS public.year CASCADE;
CREATE TABLE public.year (
    year integer NOT NULL,
    CONSTRAINT id_year PRIMARY KEY (year)
);
-- ddl-end --
ALTER TABLE public.year OWNER TO postgres;
-- ddl-end --

-- object: public.disaster | type: TABLE --
-- DROP TABLE IF EXISTS public.disaster CASCADE;
CREATE TABLE public.disaster (
    disaster_code integer NOT NULL,
    disaster varchar,
    CONSTRAINT id_disaster PRIMARY KEY (disaster_code)
);
-- ddl-end --
ALTER TABLE public.disaster OWNER TO postgres;
-- ddl-end --
-- object: public.climate_disaster | type: TABLE --
-- DROP TABLE IF EXISTS public.climate_disaster CASCADE;
CREATE TABLE public.climate_disaster (
    year integer NOT NULL,

```

```

disaster_code integer NOT NULL,
country_code integer NOT NULL,
number integer,
CONSTRAINT id_frequence PRIMARY KEY (year,disaster_code,country_code)
);
-- ddl-end --
ALTER TABLE public.climate_disaster OWNER TO postgres;
-- ddl-end --

-- object: region_code | type: CONSTRAINT --
-- ALTER TABLE public.sub_region DROP CONSTRAINT IF EXISTS region_code
CASCADE;
ALTER TABLE public.sub_region ADD CONSTRAINT region_code FOREIGN KEY
(region_code)
REFERENCES public.region (region_code) MATCH SIMPLE
ON DELETE RESTRICT ON UPDATE CASCADE;
-- ddl-end --

-- object: sub_region_code | type: CONSTRAINT --
-- ALTER TABLE public.country DROP CONSTRAINT IF EXISTS sub_region_code
CASCADE;
ALTER TABLE public.country ADD CONSTRAINT sub_region_code FOREIGN KEY
(sub_region_code)
REFERENCES public.sub_region (sub_region_code) MATCH SIMPLE
ON DELETE RESTRICT ON UPDATE CASCADE;
-- ddl-end --

-- object: disaster_code | type: CONSTRAINT --
-- ALTER TABLE public.climate_disaster DROP CONSTRAINT IF EXISTS
disaster_code CASCADE;
ALTER TABLE public.climate_disaster ADD CONSTRAINT disaster_code FOREIGN KEY
(disaster_code)
REFERENCES public.disaster (disaster_code) MATCH SIMPLE
ON DELETE NO ACTION ON UPDATE NO ACTION;
-- ddl-end --

-- object: year | type: CONSTRAINT --
-- ALTER TABLE public.climate_disaster DROP CONSTRAINT IF EXISTS year
CASCADE;
ALTER TABLE public.climate_disaster ADD CONSTRAINT year FOREIGN KEY (year)
REFERENCES public.year (year) MATCH SIMPLE
ON DELETE NO ACTION ON UPDATE NO ACTION;
-- ddl-end --

-- object: country_code | type: CONSTRAINT --
-- ALTER TABLE public.climate_disaster DROP CONSTRAINT IF EXISTS country_code
CASCADE;
ALTER TABLE public.climate_disaster ADD CONSTRAINT country_code FOREIGN KEY
(country_code)
REFERENCES public.country (country_code) MATCH SIMPLE
ON DELETE NO ACTION ON UPDATE NO ACTION;
-- ddl-end -

```

On observe plusieurs différences entre les scripts SQL manuels, comme celui que j'ai créé, et les scripts automatiques générés par un outil comme pgModeler. Tout d'abord, Les scripts manuels contiennent des commentaires détaillés pour chaque ligne, ce qui facilite sa compréhension et son ajustement. En revanche, les scripts générés par AGL incluent des commentaires qui soulignent les bonnes pratiques, comme l'utilisation de `DROP TABLE` avant de créer une nouvelle table.

De plus, les scripts générés spécifient des détails importants, tels que les actions `ON DELETE` et `ON UPDATE` pour les contraintes de clé étrangère.

Les contraintes sont données séparément avec la commande `ALTER TABLE`, rendant leur gestion plus claire. Enfin, les scripts automatiques précisent également le propriétaire des tables avec une commande `ALTER TABLE ... OWNER`.

En outre, bien que les scripts générés par des outils comme pgModeler offrent des fonctionnalités avancées pour gérer les relations et les contraintes de manière automatisée, le script SQL manuel présente l'avantage de sa flexibilité. Il est plus simple à modifier selon les besoins spécifiques d'un projet, car il n'est pas limité par les fonctionnalités préconfigurées de l'outil.

### 3) Peuplement des tables

Pour cette partie, j'ai choisi de manipuler uniquement un seul fichier, à savoir la figure 1. L'objectif de cette partie est de réaliser un script SQL permettant la projection des données présentes dans ce fichier afin de peupler les tables de notre base de données.

Tout d'abord on va copier ce fichier plat dans une table temporaire. On va créer une table temporaire qui aura autant de colonne que le fichier csv.

voici le script pour cette table:

```
CREATE TABLE temporaire (
    country varchar,
    is02 char(2),
    is03 char(3),
    region_code integer,
    region varchar,
    sub_region_code integer,
    sub_region varchar,
    disaster varchar,
    year integer,
    number integer);
```

Maintenant, après avoir extrait le fichier ZIP et donné les permissions, nous allons le copier dans notre table temporaire grâce à la commande suivante :

```
\copy temporaire (country, is02, is03, region_code, region, sub_region_code,
sub_region, disaster, year, number) FROM 'chemin du fichier' CSV HEADER;
```

→Le mot-clé HEADER indique que la première ligne du fichier CSV contient les noms des colonnes, et PostgreSQL ignorera cette ligne lors de l'importation.

```
constantine=> \copy temporaire (country, is02, is03, region_code, region, sub_region_code, sub_region, disaster, year, number) FROM 'sae_extracted/Climate_related_disasters_frequency.csv' CSV HEADER;
COPY 6448
```

Après avoir exécuté la commande, nous obtenons 6448 éléments copiés dans la table temporaire, ce qui correspond au nombre de lignes du fichier.

Ensuite, pour éviter d'éventuels bugs, nous allons vérifier que la table temporaire ne contient pas de valeurs nulles, notamment dans les colonnes qui ne doivent pas être **NOT NULL**.

Nous effectuons cette vérification grâce à la requête suivante :

```
code | country | is02 | is03 | region_code | region | s
-----+-----+-----+-----+-----+
Saint Barthélémy | | BLM | 19 | Americas |
419 | Latin America and the Caribbean | Storm | 2017 | 1
Saint Martin (French Part) | | MAF | 19 | Americas |
419 | Latin America and the Caribbean | Storm | 2017 | 1
[2 rows)
```

```
SELECT *
FROM temporaire
WHERE <nomcolonne> IS NULL;
```

Lors des tests sur psql, nous constatons que seule la colonne `is02` contient des valeurs `NULL`. Nous allons donc exécuter une requête pour remplacer ces valeurs nulles par des valeurs par défaut :

```
UPDATE temporaire
SET is02 = 'XX'
WHERE is02 IS NULL;
```

Il se révèle qu'ensuite, dans notre table temporaire, les colonnes `country_code` et `disaster_code` ne figurent pas. Il est donc nécessaire de générer ces codes automatiquement. Cela est très simple : il suffit de reprendre notre script SQL de création de table et de modifier le type de ces deux colonnes. Nous allons remplacer `INTEGER` par `SERIAL` pour qu'elles s'auto-incrémentent.

La prochaine étape consiste à peupler nos tables. Il est important de respecter l'ordre de création des tables pour éviter des problèmes liés aux clés étrangères. Cela signifie que pour commencer, nous allons remplir la table `region`.

Cette table contient deux colonnes : `region_code` et `name`. Ces colonnes seront remplies respectivement par les colonnes `region_code` et `region` de la table temporaire.

Voici la commande SQL utilisée :

```
-- Insère les données dans la table region--
INSERT INTO region (region_code, name)
--Sélectionne les colonnes en veillant à éviter les doublons.--
SELECT DISTINCT region_code, region
-- Spécifie la table source d'où proviennent les données.--
FROM temporaire;
```

[Explication des commandes utilisées:](#)

**INSERT INTO** : Cette commande permet d'insérer des données dans une table spécifique de la base de données. Les colonnes ciblées dans la table de destination sont précisées entre parenthèses.

**SELECT DISTINCT** : Cette commande est utilisée pour sélectionner uniquement les lignes uniques à partir des colonnes spécifiées dans une table source.

L'utilisation de **DISTINCT** garantit que les données insérées dans la table cible ne contiennent aucun doublon, ce qui est particulièrement important pour des informations uniques comme des régions, des pays ou des identifiants.

Pour remplir la table `sub_region`, la colonne `sub_region_code` et la colonne `name` seront remplies respectivement avec les valeurs des colonnes `sub_region_code` et `sub_region` de la table temporaire. La colonne `region_code` de la table `sub_region` sera remplie avec le `region_code` de la table `region`, afin de faire référence à la clé étrangère. Pour cela, une jointure sera réalisée entre la table `region` et la table temporaire sur la colonne `region`.

```
-- Insertion des données dans la table sub_region--
INSERT INTO sub_region (sub_region_code, name, region_code)
-- Sélection des colonnes--
SELECT DISTINCT temporaire.sub_region_code, -- Code de la sous-région--
temporaire.sub_region, -- Nom de la sous-région--
region.region_code -- Code de la région associé-
-- Spécifie la table source d'où proviennent les données.--
FROM temporaire
-- Jointure entre la table temporaire et la table region sur la colonne
'region' pour récupérer le 'region_code'--
JOIN region ON temporaire.region = region.name;
```

Explication commandes:

**JOIN** : combine des données provenant de plusieurs tables selon une condition.

Pour remplir la table `country`, nous procédons de la même manière que pour la table précédente. Les colonnes de `country` seront remplies avec les données correspondantes de la table temporaire, et la colonne `sub_region_code`.

Voici la commande sql:

```
-- Insertion des données dans la table country--
INSERT INTO country (name, "IS02", "IS03", sub_region_code)
SELECT DISTINCT temporaire.country, -- Sélection du nom du pays--
temporaire.IS02, -- Sélection du code IS02 (avec valeurs par
défaut si NULL)--
temporaire.IS03, -- Sélection du code IS03--
sub_region.sub_region_code -- code de la sous-région-
-- Spécifie la table source d'où proviennent les données.--
FROM temporaire
JOIN sub_region ON temporaire.sub_region = sub_region.name; -- Jointure sur
le nom de la sous-région--
```

Pour remplir la table disaster on procède pareil que la table region:

```
-- Insertion des données dans la table disaster--
INSERT INTO disaster(disaster)
--Sélectionne les colonnes en veillant à éviter les doublons.--
SELECT DISTINCT temporaire.disaster
-- Spécifie la table source d'où proviennent les données.--
FROM temporaire;
```

Peupler la table `country` est assez similaire aux étapes précédentes, mais avec une particularité : il y a une double jointure. La première jointure se fait entre la table temporaire et la table `country` pour récupérer le `country_code`, et la seconde jointure se fait entre la table temporaire et la table `disaster` pour récupérer le `disaster_code`. Bien qu'il puisse y avoir des doublons à cause des jointures, la structure des données et les relations entre les tables garantissent l'absence de doublons dans la table `country`.

Voici le script sql:

```
-- Insertion des données dans la table climate_disaster--
INSERT INTO climate_disaster (year, disaster_code, country_code, number)
-- Sélection des données à insérer--
SELECT temporaire.year, -- Sélection de l'année depuis la table temporaire--
        disaster.disaster_code, -- Sélection du code du désastre depuis la
        table disaster--
        country.country_code, -- Sélection du code du pays depuis la table
        country--
        temporaire.number -- Sélection du nombre d'événements depuis la table
        temporaire--
FROM temporaire
-- Jointure entre la table temporaire et la table country sur le nom du
pays--
JOIN country ON temporaire.country = country.name
-- Jointure entre la table temporaire et la table disaster sur le nom du
désastre--
JOIN disaster ON temporaire.disaster = public.disaster.disaster;
```

### 3- Conclusion :

Pour conclure, cette SAÉ nous a permis de manipuler divers outils essentiels pour la gestion de bases de données, tels que PostgreSQL et un AGL. Nous avons également travaillé avec des scripts SQL, non seulement pour créer des tables et rédiger des requêtes permettant de les peupler, mais aussi pour extraire des informations à partir d'un fichier CSV. Cette expérience nous a offert une compréhension globale du cycle de vie d'une base de données et nous a aidé à renforcer nos compétences pratiques dans ce domaine.