SAE S104
(BASE DE DONNÉES)

Sommaire

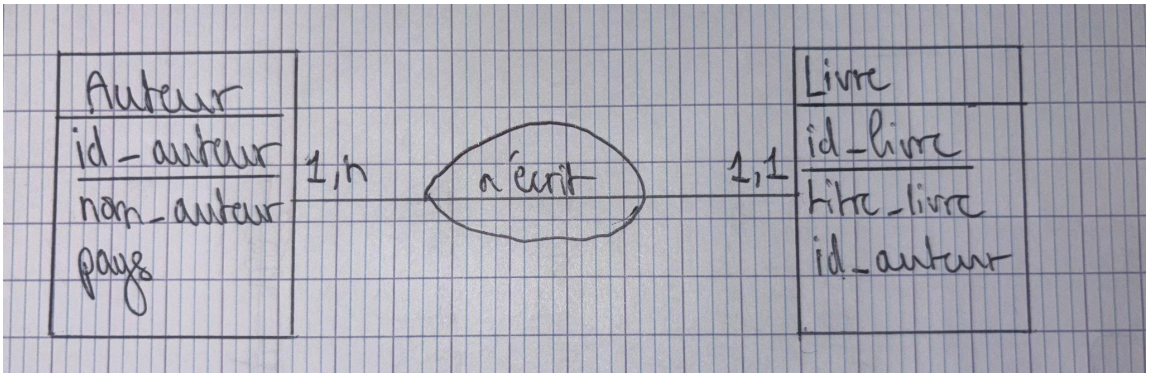
1. Script manuel de création de la base de données

```
DROP TABLE IF EXISTS climate_disaster, country, sub_region,
disaster, region;


CREATE TABLE region (
      region_code INTEGER PRIMARY KEY,
      name VARCHAR NOT NULL
);

CREATE TABLE disaster (
      disaster_code INTEGER PRIMARY KEY,
      disaster VARCHAR NOT NULL
);

CREATE TABLE sub_region (
      sub_region_code INTEGER PRIMARY KEY,
      name VARCHAR NOT NULL,
      region_code INTEGER REFERENCES region
);

 CREATE TABLE country (
      country_code INTEGER PRIMARY KEY,
      name VARCHAR NOT NULL,
      ISO2 CHAR(2),
      ISO3 CHAR(3),
      sub_region_code INTEGER REFERENCES sub_region
);

CREATE TABLE climate_disaster (
      country_code INTEGER REFERENCES country,
      disaster_code INTEGER REFERENCES disaster,
      year INTEGER NOT NULL,
      number INTEGER NOT NULL,
      PRIMARY KEY (country_code, disaster_code, year)
);
```

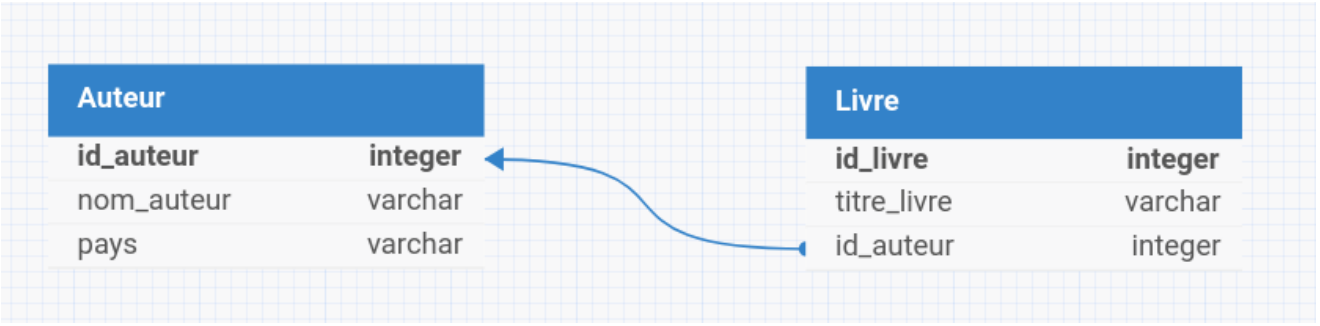
2. Modélisation et script de création avec “AGL”

A. Comparaison Cours/AGL d’une association fonctionnelle

Cours (inventée en prenant exemple sur le cours) :



AGL :



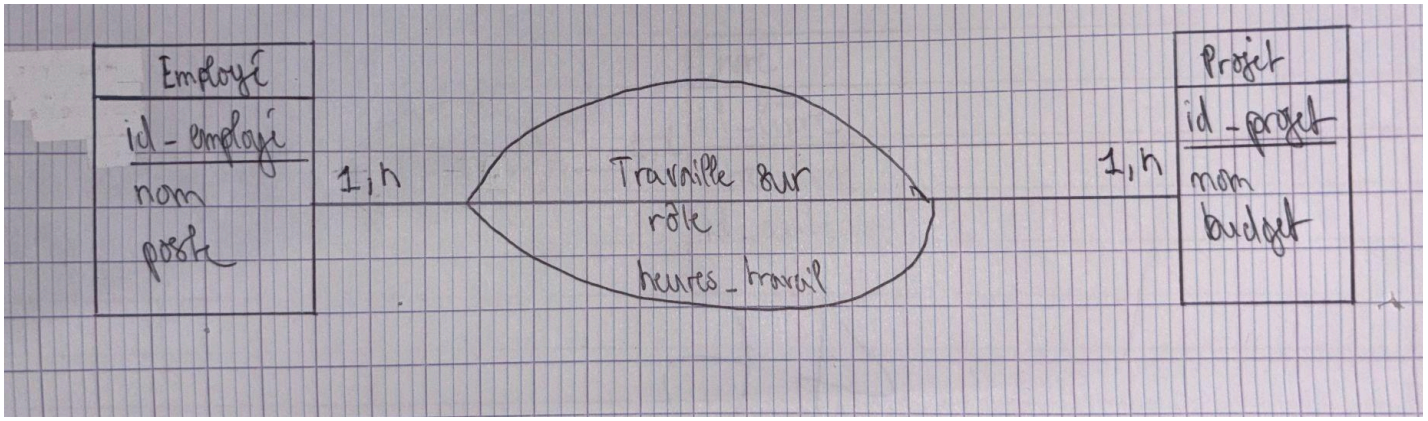
Explication :

Dans le modèle conceptuel vu en cours, on voit la présence ici de deux type-entités : **Auteur** et **Livre**. Entre ces deux type-entités, on a une relation “**a écrit**”. C’est un type-association de type fonctionnel (1,n). Ces types-entités ont chacun 3 attributs : Une clé primaire, un nom pour l’auteur et un titre pour le livre et le pays de l’auteur. **Livre** se souvient de **Auteur**. En effet, un livre est écrit par un seul et même auteur (les cardinalités de la patte du type-association vers **Livre** sont (1,1)), de ce fait, il doit “se souvenir de son créateur”. “id_auteur” devient donc une clé étrangère dans **Livre**.

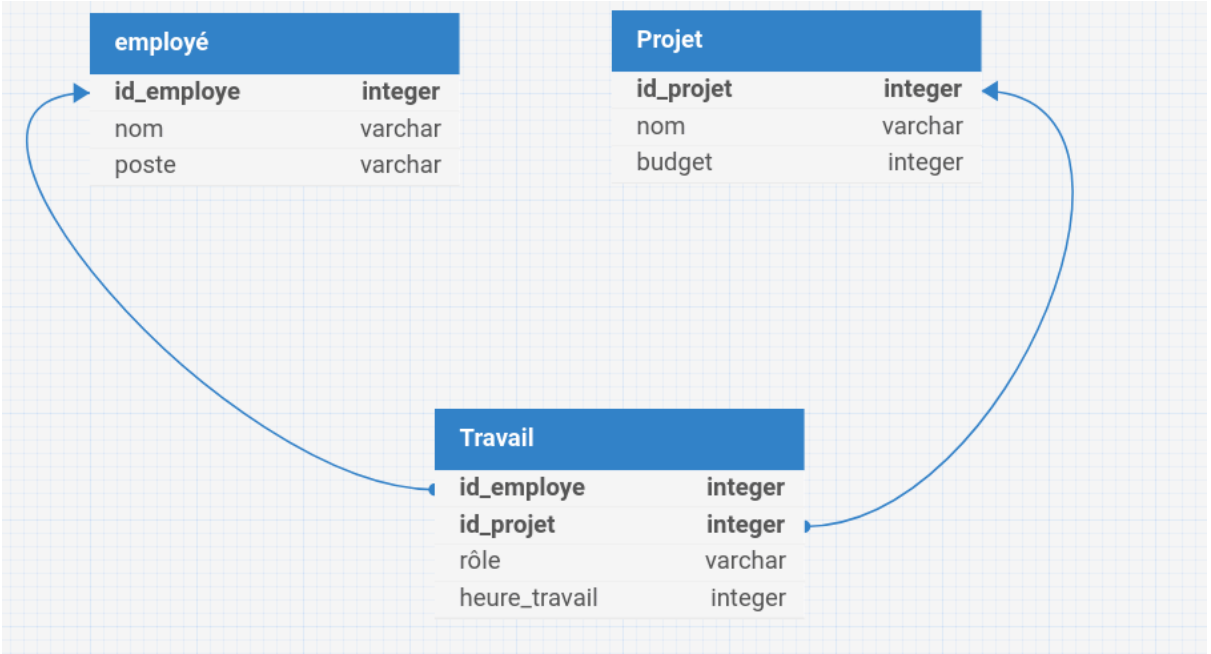
Pour l'AGL, c'est différent. On a cette fois ci des tables, mais on garde les mêmes attributs, et on précise même leur type (entier, chaîne de caractères…). Le type-association disparaît et on se retrouve avec des flèches. On voit une flèche partir de l'attribut "id_auteur" de **Livre** pour arriver vers l'attribut "id_auteur" de **Auteur**. Cela signifie que "id_auteur" de **Livre** est une clé étrangère faisant référence à "id_auteur" de **Auteur**. Les cardinalités n'étant plus présentes et ne pouvant plus communiquer (plus ou moins implicitement) de la provenance d'une clé étrangère, ces flèches s'en chargent. On conserve donc une relation (assez précise) entre les deux tables.

B. Comparaison cours/AGL d'une association maillée

Cours (inventée en prenant exemple sur le cours) :



AGL :



Explication :

Dans le modèle conceptuel vu en cours, on observe ici trois type-entités : **Employé**, **Projet**, et un type-association **Travaille_sur**. Entre ces deux type-entités principales (**Employé** et **Projet**), la relation **Travaille_sur** est une entité associative qui représente une association maillée. Cette association permet de stocker des informations supplémentaires, comme le rôle de l'employé dans un projet et le nombre d'heures travaillées.

Les trois type-entités possèdent leurs propres attributs :

- **Employé** a trois attributs : une clé primaire (id_employe), un nom (nom_employe), et un poste (poste).
- **Projet** a trois attributs également : une clé primaire (id_projet), un nom (nom_projet), et un budget (budget).
- **Travaille_sur**, en tant qu'entité associative, a une clé primaire composée (id_employe, id_projet), ainsi que deux autres attributs : role et heures_travail.

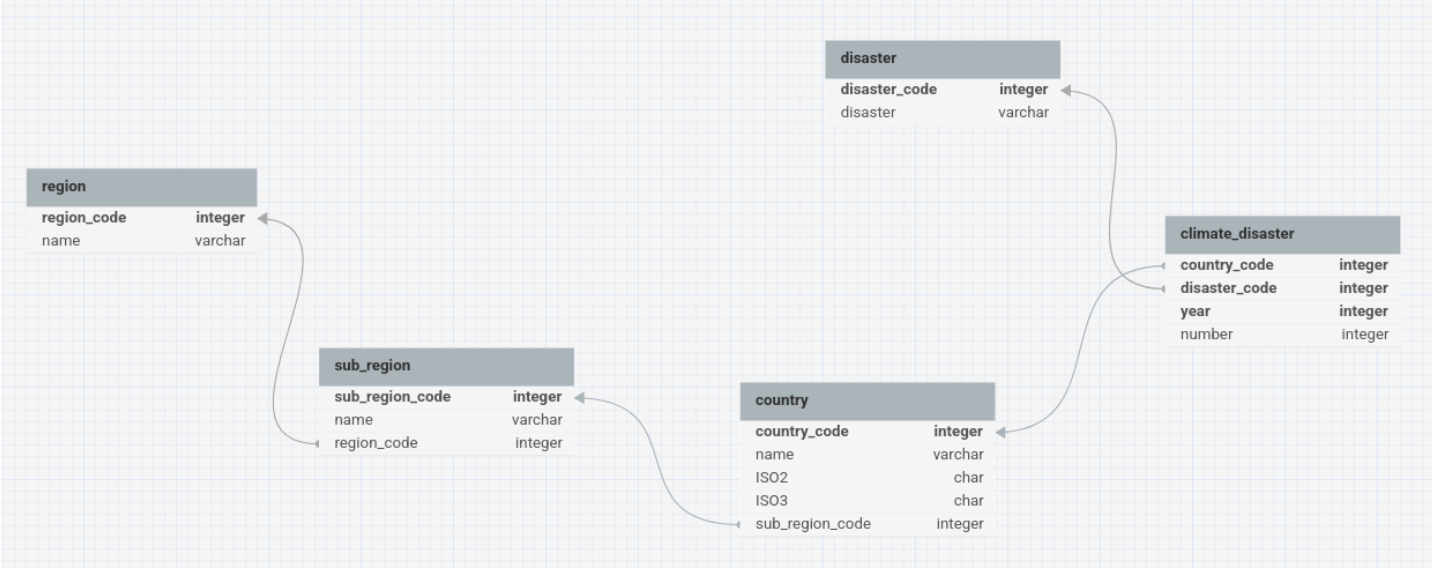
Les cardinalités dans le modèle montrent qu'un Employé peut participer à plusieurs Projets (1,n) et qu'un Projet peut impliquer plusieurs Employés (1,n). Cela signifie qu'un Employé et un Projet peuvent être liés plusieurs fois, avec des détails différents dans l'entité **Travaille_sur**. Dans cette structure, **Travaille_sur** se souvient à la fois de **Employé** et de **Projet**, car elle référence leurs clés primaires (id_employe et id_projet) pour créer une relation maillée enrichie.

Pour l'AGL, le modèle change : on passe de type-entités et d'une entité associative à des tables, où chaque attribut est explicitement typé (entier, chaîne de caractères, etc.). L'entité associative **Travaille_sur** devient une table à part entière, et les relations entre les tables sont représentées par des flèches dans l'AGL. Dans ce modèle, **Employé** et **Projet** conservent leurs clés primaires respectives (id_employe et id_projet). La table **Travaille_sur** inclut les deux clés primaires comme clés étrangères (qui forment à elles-deux la clé primaire de **Travaille_sur**), tout en stockant des attributs supplémentaires :

- role : par exemple, "Chef de projet", "Développeur".
- heures_travail : le nombre d'heures que l'employé a travaillé sur le projet.

Les flèches dans l'AGL indiquent que “id_employe” dans **Travaille_sur** est une clé étrangère pointant vers “id_employe” de la table **Employé**, et que “id_projet” dans **Travaille_sur** est une clé étrangère pointant vers “id_projet” de la table **Projet**. Ces flèches remplacent les cardinalités du MCD en indiquant la provenance des clés étrangères et en conservant une structure précise des relations entre les tables.

C. Modèle physique de données réalisé avec l'AGL



D. Script SQL de création des tables généré automatiquement par l'AGL

```
CREATE TABLE "region" (
      "region_code" bigint NOT NULL,
      "name" varchar(255) NOT NULL,
      PRIMARY KEY ("region_code")
);

CREATE TABLE "disaster" (
      "disaster_code" bigint NOT NULL,
      "disaster" varchar(255) NOT NULL,
      PRIMARY KEY ("disaster_code")
);

CREATE TABLE "sub_region" (
      "sub_region_code" bigint NOT NULL,
      "name" varchar(255) NOT NULL,
      "region_code" bigint NOT NULL,
      PRIMARY KEY ("sub_region_code")
);
```

```
CREATE TABLE "country" (
      "country_code" bigint NOT NULL,
      "name" varchar(255) NOT NULL,
      "ISO2" varchar(2),
      "ISO3" varchar(3),
      "sub_region_code" bigint NOT NULL,
      PRIMARY KEY ("country_code")
);

CREATE TABLE "climate_disaster" (
      "country_code" bigint NOT NULL,
      "disaster_code" bigint NOT NULL,
      "year" bigint NOT NULL,
      "number" bigint NOT NULL,
      PRIMARY KEY ("country_code", "disaster_code", "year")
);



ALTER TABLE "sub_region" ADD CONSTRAINT "sub_region_fk2" FOREIGN KEY
("region_code") REFERENCES "region"("region_code");
ALTER TABLE "country" ADD CONSTRAINT "country_fk4" FOREIGN KEY
("sub_region_code") REFERENCES "sub_region"("sub_region_code");
ALTER TABLE "climate_disaster" ADD CONSTRAINT "climate_disaster_fk0"
FOREIGN KEY ("country_code") REFERENCES "country"("country_code");

ALTER TABLE "climate_disaster" ADD CONSTRAINT "climate_disaster_fk1"
FOREIGN KEY ("disaster_code") REFERENCES
"disaster"("disaster_code");
```

E. Différences entre le script manuel et celui généré par l’AGL

Dans le script SQL écrit manuellement, on observe une structure simplifiée et intuitive : chaque table est créée avec des colonnes directement définies, incluant les types de données et les relations avec d'autres tables. Les types sont adaptés au contexte : **INTEGER** pour des identifiants ou des clés, et **CHAR** ou **VARCHAR** pour des valeurs textuelles. La gestion des clés étrangères est faite directement dans la définition des colonnes, ce qui améliore la lisibilité et la compréhension des relations entre les tables. De plus, la commande `DROP TABLE IF EXISTS` permet de s'assurer qu'aucune table existante ne gênera l'exécution du script.

En revanche, dans le script généré par l'AGL, on remarque une approche plus formelle et standardisée. Les noms des tables et colonnes sont entourés de guillemets doubles (par exemple, "region" ou "region_code"), probablement pour garantir la compatibilité avec des noms potentiellement sensibles (noms contenant des majuscules, espaces ou mots-clés). Les types de données utilisés sont plus génériques : **bigint** remplace **INTEGER**, et **varchar** est systématiquement utilisé pour les chaînes de caractères, même lorsque que l'on connaît déjà la longueur de la chaîne, comme pour ISO2 ou ISO3. Les relations entre les tables ne sont pas spécifiées dans la définition des colonnes, mais ajoutées plus tard via des contraintes explicites avec `ALTER TABLE`, en générant automatiquement des noms de contraintes comme "climate_disaster_fk0". Cette méthode rend le script plus flexible mais allonge sa structure.

Une autre différence majeure réside dans la gestion de l'existence des tables : le script généré par l'AGL utilise `CREATE TABLE IF NOT EXISTS`, ce qui empêche de recréer des tables si elles existent déjà. Cette approche ne supprime cependant pas les anciennes versions des tables, contrairement au `DROP TABLE IF EXISTS` du script manuel, qui garantit un environnement propre avant la création.

En résumé, le script manuel est concis, directement compréhensible, et optimisé pour des besoins courants. Le script généré par l'AGL suit une approche formelle et générique, adaptée à des cas où des normes strictes ou une compatibilité étendue sont nécessaires, mais il peut introduire des redondances et une complexité supplémentaire.

3. Peuplement des tables

J'ai décidé d'utiliser le fichier plat présenté dans la figure 1, qui est plus simple pour le peuplement des tables. Voici le début de mon script :

```
CREATE TABLE "donnees" (
      "country" VARCHAR,
      "iso2" CHAR(2),
      "iso3" CHAR(3),
      "region_code" INTEGER,
      "region" VARCHAR,
      "sub_region_code" INTEGER,
      "sub_region" VARCHAR,
      "disaster" VARCHAR,
      "year" INTEGER,
      "number" INTEGER
);
```

```
\copy "donnees" FROM
'/home/edohdagnon/Downloads/Climate_related_disasters_frequency.csv'
WITH CSV HEADER DELIMITER ',';
```

J'ai décidé de prendre exemple sur l'AGL en encadrant les noms des colonnes par des quotes, donc pour la question de protection des noms "sensibles". Le début du script représente donc la création d'une table temporaire avec toutes les colonnes du fichier CSV fourni et leurs types respectifs. Puis je me suis servi de la commande `\copy` pour copier les données du fichier plat dans ma table temporaire.

En regardant bien le fichier CSV et en comparant au modèle relationnel des données de la SAE, j'ai remarqué que la colonne "disaster_code" de la table **disaster** n'existait pas. Idem pour "country_code" de **country**. Il a été conseillé d'utiliser le type SERIAL pour produire des id automatiquement, alors je vais modifier la partie de la création des tables **disaster** et **country** concernée dans le script de création des table. Cela donne le nouveau script :

```
DROP TABLE IF EXISTS climate_disaster, country, sub_region,
disaster, region;

CREATE TABLE "region" (
      "region_code" bigint NOT NULL,
      "name" varchar(255) NOT NULL,
      PRIMARY KEY ("region_code")
);

CREATE TABLE "disaster" (
      "disaster_code" SERIAL,
      "disaster" varchar(255) NOT NULL,
      PRIMARY KEY ("disaster_code")
);

CREATE TABLE "sub_region" (
      "sub_region_code" bigint NOT NULL,
      "name" varchar(255) NOT NULL,
      "region_code" bigint NOT NULL,
      PRIMARY KEY ("sub_region_code")
);

CREATE TABLE "country" (
      "country_code" SERIAL,
      "name" varchar(255) NOT NULL,
      "ISO2" varchar(2),
      "ISO3" varchar(3),
      "sub_region_code" bigint NOT NULL,
```

```
      PRIMARY KEY ("country_code")
);

CREATE TABLE "climate_disaster" (
      "country_code" bigint NOT NULL,
      "disaster_code" bigint NOT NULL,
      "year" bigint NOT NULL,
      "number" bigint NOT NULL,
      PRIMARY KEY ("country_code", "disaster_code", "year")
);



ALTER TABLE "sub_region" ADD CONSTRAINT "sub_region_fk2" FOREIGN KEY
("region_code") REFERENCES "region"("region_code");
ALTER TABLE "country" ADD CONSTRAINT "country_fk4" FOREIGN KEY
("sub_region_code") REFERENCES "sub_region"("sub_region_code");
ALTER TABLE "climate_disaster" ADD CONSTRAINT "climate_disaster_fk0"
FOREIGN KEY ("country_code") REFERENCES "country"("country_code");

ALTER TABLE "climate_disaster" ADD CONSTRAINT "climate_disaster_fk1"
FOREIGN KEY ("disaster_code") REFERENCES
"disaster"("disaster_code");
```

Maintenant que cela est fait, je peux continuer la création du script de peuplement. Bien sûr, on essaiera de créer un script de peuplement en respectant les relations entre les tables.

Voici la suite :

```
INSERT INTO region (region_code, name)
SELECT DISTINCT region_code, region
FROM donnees
WHERE region_code IS NOT NULL AND region IS NOT NULL;

INSERT INTO sub_region (sub_region_code, name, region_code)
SELECT DISTINCT sub_region_code, sub_region, region_code
FROM donnees
WHERE sub_region_code IS NOT NULL AND sub_region IS NOT NULL AND
region_code IS NOT NULL;

INSERT INTO disaster (disaster)
SELECT DISTINCT disaster
FROM donnees
```

```
WHERE disaster IS NOT NULL;
```

Pour remplir les tables **region**, **sub_region** et **disaster**, j'ai a chaque fois utilisé la même forme :

- Je définis la table, plus précisément les colonnes de la table dans laquelle je veux insérer les données
- Je fais une requête SELECT DISTINCT en choisissant les colonnes de ma table temporaire dans lesquelles je vais prélever les données
- Je spécifie que je veux des lignes avec des valeurs connues a chaque fois (NOT NULL) dans le champ WHERE

La suite du script est un peu plus technique :

```
INSERT INTO country (name, "ISO2", "ISO3", sub_region_code)
SELECT DISTINCT d.country, d."ISO2", d."ISO3", sr.sub_region_code
FROM donnees d INNER JOIN sub_region sr ON d.sub_region = sr.name
WHERE d.country IS NOT NULL;

INSERT INTO climate_disaster (country_code, disaster_code, year,
number)
SELECT
      c.country_code,
      di.disaster_code,
      d.year,
      d.number
FROM donnees d INNER JOIN country c ON d.country = c.name INNER JOIN
disaster di ON d.disaster = di.disaster;



DROP TABLE donnees;
```

Pour remplir les tables country et climate_disaster, j'ai utilisé une logique similaire à celle employée pour les autres tables, mais avec quelques adaptations :

- Insertion dans la table country :

- Je définis la table, en précisant les colonnes dans lesquelles je vais insérer les données : name, "ISO2", "ISO3" et sub_region_code.

- Je fais une requête SELECT DISTINCT pour éviter les doublons, en prélevant les données de la table temporaire donnees.
- J'utilise une jointure interne (INNER JOIN) avec la table sub_region pour récupérer le sub_region_code correspondant au nom de la sous-région.
- Enfin, j'ajoute une condition WHERE pour m'assurer de ne pas insérer de pays ayant des noms nuls.

- Insertion dans la table climate_disaster :
- Je définis la table climate_disaster, en précisant les colonnes dans lesquelles je vais insérer les données : country_code, disaster_code, year et number.
- Je fais une requête SELECT pour récupérer les données nécessaires depuis la table temporaire donnees.
- J'utilise des jointures internes avec les tables country et disaster pour obtenir les codes correspondants (country_code et disaster_code). Ces jointures permettent de transformer les noms de pays et de catastrophes en identifiants uniques déjà présents dans les tables.

Une fois toutes les données insérées dans les tables finales, je supprime la table temporaire avec la commande DROP TABLE. Voici donc le script final de peuplement :

```
DROP TABLE IF EXISTS "donnees";



CREATE TABLE "donnees" (
      "country" VARCHAR,
      "ISO2" CHAR(2),
      "ISO3" CHAR(3),
      "region_code" INTEGER,
      "region" VARCHAR,
      "sub_region_code" INTEGER,
      "sub_region" VARCHAR,
      "disaster" VARCHAR,
      "year" INTEGER,
      "number" INTEGER
);

\copy "donnees" FROM
'/home/edohdagnon/s1/s104/Climate_related_disasters_frequency.csv'
WITH CSV HEADER DELIMITER ',';

INSERT INTO region (region_code, name)
SELECT DISTINCT region_code, region
FROM donnees
WHERE region_code IS NOT NULL AND region IS NOT NULL;
```

```
INSERT INTO sub_region (sub_region_code, name, region_code)
SELECT DISTINCT sub_region_code, sub_region, region_code
FROM donnees
WHERE sub_region_code IS NOT NULL AND sub_region IS NOT NULL AND
region_code IS NOT NULL;

INSERT INTO disaster (disaster)
SELECT DISTINCT disaster
FROM donnees
WHERE disaster IS NOT NULL;

INSERT INTO country (name, "ISO2", "ISO3", sub_region_code)
SELECT DISTINCT d.country, d."ISO2", d."ISO3", sr.sub_region_code
FROM donnees d INNER JOIN sub_region sr ON d.sub_region = sr.name
WHERE d.country IS NOT NULL;

INSERT INTO climate_disaster (country_code, disaster_code, year,
number)
SELECT
      c.country_code,
      di.disaster_code,
      d.year,
      d.number
FROM donnees d INNER JOIN country c ON d.country = c.name INNER JOIN
disaster di ON d.disaster = di.disaster;


DROP TABLE donnees;
```