

# Climate-Related Disasters Frequency

## **Sommaire :**

### Création des tables

[Modèle conceptuel des données \(MCD\)](#)

[Modèle physique des données \(MPD\)](#)

[Choix de l'AGL](#)

[Modèle réalisé avec l'AGL](#)

[Lecture du modèle de l'AGL](#)

[Script SQL généré par l'AGL](#)

### Différence entre les scripts produits manuellement et automatiquement

### Illustrations comparatives

[Association fonctionnelle](#)

[Association maillée](#)

### Peuplement des tables

[Modification du script de création de table](#)

[Première partie du peuplement \(fichier fig. 3\)](#)

[Création des tables temporaires](#)

[Insertion des données dans les premières tables](#)

[Deuxième partie du peuplement \(fichier fig. 2\)](#)

[Création d'une table temporaire](#)

[Insertion des données dans les tables restantes](#)

### Fichier SQL complet du script de peuplement

### Source

## **Création des tables**

### **Modèle conceptuel des données (MCD)**

```
CREATE TABLE region (  
  region_code INTEGER PRIMARY KEY,  
  name VARCHAR  
);  
  
CREATE TABLE sub_region (  
  sub_region_code INTEGER PRIMARY KEY,  
  name VARCHAR,  
  region_code INTEGER REFERENCES region (region_code)  
);  
  
CREATE TABLE country (  
  country_code INTEGER PRIMARY KEY,  
  name VARCHAR,  
  ISO2 VARCHAR,  
  ISO3 VARCHAR,  
  sub_region_code INTEGER REFERENCES sub_region (sub_region_code)  
);
```

```

CREATE TABLE disaster (
  disaster_code INTEGER PRIMARY KEY,
  disaster VARCHAR
);

CREATE TABLE climate_disaster (
  country_code INTEGER REFERENCES country (country_code),
  disaster_code INTEGER REFERENCES disaster (disaster_code),
  year INTEGER,
  -- j'ai utilisé le type INTEGER plutôt que DATE, car le format -- de
  DATE est AAAA-MM-JJ
  number INTEGER,
  PRIMARY KEY (country_code, disaster_code, year)
);

```

Je n'ai pas mis **NOT NULL** aux différents attributs de chaque table, notamment pour des cas où certaines valeurs pourraient être manquantes. Je n'ai pas jugé nécessaire de contraindre un pays à avoir une région par exemple.

Je n'utilise pas l'auto incrémentation (**SERIAL**) pour les identifiants (**PRIMARY KEY**) des différentes tables pour l'instant. Je déciderai de son utilisation lors de l'utilisation des fichiers pour le peuplement de mes tables.

Ci-dessous la liste des tables après avoir tapé mon script dans le terminal :

List of relations	
Name	Type
climate_disaster	table
country	table
disaster	table
region	table
sub_region	table

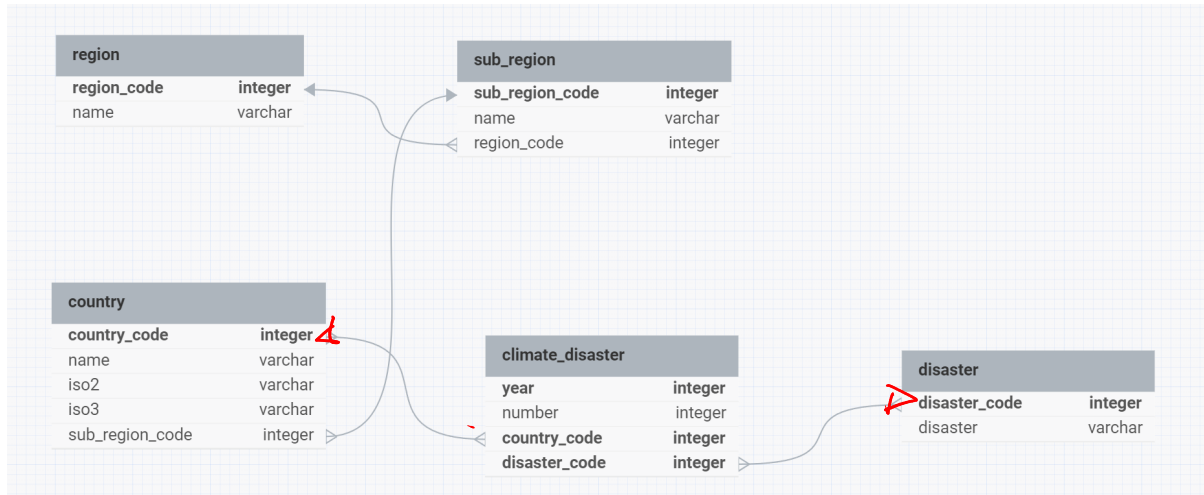
## Modèle physique des données (MPD)

### Choix de l'AGL

J'ai opté pour l'AGL **DBDesigner** pour plusieurs raisons :

- **Esthétique.** Interface minimaliste et bien organisée, on ne se perd pas visuellement.
- **Intuitive** et facile à prendre en main, on comprend assez facilement où cliquer. J'ai également réussi à trouver un tutoriel récent.
- **Gratuit** avec une limite de nombre de tables, mais pas besoin de beaucoup de tables dans mon cas.
- **Site internet**, [dbdesigner.net](http://dbdesigner.net), donc pas besoin d'installation.

## Modèle réalisé avec l'AGL



## Lecture du modèle de l'AGL

### Relations entre les entités :

- region ↔ sub\_region** :
  - Une région peut contenir 0 ou plusieurs sous-régions (0, n). Sur l'AGL elle est représentée avec 3 branches du côté des sous-régions (**sub\_region**).
  - Une sous-région appartient toujours à une seule et unique région (1, 1). Sur l'AGL elle est représentée avec une flèche pleine du côté des régions (**region**).
- sub\_region ↔ country** :
  - Une sous-région peut contenir plusieurs pays (0, n). Sur l'AGL elle est représentée avec 3 branches du côté des pays (**country**).
  - Un pays appartient toujours à une seule et unique sous-région (1, 1). Sur l'AGL elle est représentée avec une flèche pleine du côté des sous-régions (**sub\_region**).
- country ↔ climate\_disaster** :
  - Un pays peut être affecté par plusieurs catastrophes climatiques (0, n).
  - Une catastrophe climatique est toujours associée à un seul pays pour une année donnée. Cependant le code du pays (**country\_code**) peut apparaître plusieurs fois dans la table **climate\_disaster**, s'il a un désastre différent la même année, ou bien le même désastre mais à une année différente. Donc c'est une relation (0, n).
- disaster ↔ climate\_disaster** :
  - Un désastre peut affecter plusieurs pays (0, n).
  - Un type de catastrophe peut apparaître plusieurs fois dans **climate\_disaster**, pour des pays ou des années différentes. Donc c'est également une relation (0, n).

*même relation (bien)*

## Script SQL généré par l'AGL

```

CREATE TABLE IF NOT EXISTS "region" (
  "region_code" integer NOT NULL,
  "name" varchar(255) NOT NULL,

```

```

        PRIMARY KEY ("region_code")
    );

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

CREATE TABLE IF NOT EXISTS "country" (
    "country_code" integer NOT NULL,
    "name" varchar(255) NOT NULL,
    "iso2" varchar(255) NOT NULL,
    "iso3" varchar(255) NOT NULL,
    "sub_region_code" bigint NOT NULL,
    PRIMARY KEY ("country_code")
);

CREATE TABLE IF NOT EXISTS "disaster" (
    "disaster_code" integer NOT NULL,
    "disaster" varchar(255) NOT NULL,
    PRIMARY KEY ("disaster_code")
);

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

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_fk2"
        FOREIGN KEY ("country_code") REFERENCES "country"("country_code");

ALTER TABLE "climate_disaster"
    ADD CONSTRAINT "climate_disaster_fk3"
        FOREIGN KEY ("disaster_code") REFERENCES "disaster"("disaster_code");

```

## Différence entre les scripts produits manuellement et automatiquement

Les différences concernent principalement la gestion des clés étrangères et le type des attributs, nous allons les énumérer :

Script manuel	Script automatique
Pas de condition <code>IF NOT EXISTS</code> . Jugé non nécessaire.	Condition <code>IF NOT EXISTS</code> à chaque création de table, pour ne pas produire d'erreur au cas où.
Pas de limites de taille pour <code>VARCHAR</code> .	Limite de 255 caractères pour les <code>VARCHAR</code> .
Utilisation du type <code>INTEGER</code> pour les attributs correspondant au code de certaines colonnes.	Utilisation de <code>bigint</code> dans le cas où il y aurait des données à gros volumes.
Aucun <code>NOT NULL</code> .	A mis des <code>NOT NULL</code> à tous les attributs.
Intégration des clés étrangères directement lors de la création des tables, avec <code>REFERENCES region (region_code)</code> par exemple.	Crée des clés étrangères via des <code>ALTER TABLE</code> après la création des tables, en les modifiant. Avec des noms explicites pour les contraintes comme <code>sub_region_fk2</code> .

Ce tableau met en évidence que le script manuel est plus minimaliste et lisible, tandis que le script automatique est bien plus précis, avec certaines conditions, contraintes et des types différents. Le script automatique est beaucoup plus adapté à des bases de données complexes.

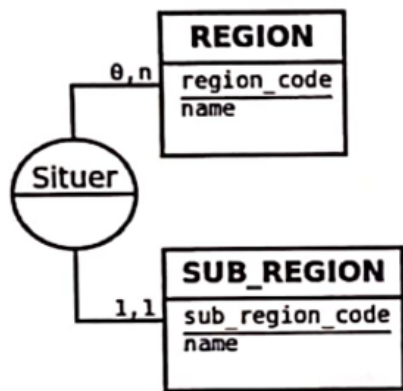
Cependant dans notre cas, notre base de données n'étant pas complexe, on veut un script qui soit le plus dépouillé possible. Pour cela on va procéder à certaines modifications dans le script de l'AGL :

- On peut se permettre de retirer les `IF NOT EXISTS` (évite les erreurs si la table existe) car le script est destiné à la création d'une nouvelle base de données. Donc ils ne sont pas nécessaires vu que les tables n'existent pas.
- On retire également la spécification de taille pour les `varchar`, la taille spécifiée n'a pas d'impact sur l'utilisation de la mémoire, donc ce sera souvent inutile.
- On va remplacer les `bigint` par des `integer`. Les `bigint` sont destinés à de très grand nombre (des milliards), ce qui ne sera pas notre cas ici.
- Les contraintes (ex. `ADD CONSTRAINT "sub_region_fk2"`) dans `ALTER TABLE` peuvent être retirées sans impacter notre script, pour que ce soit plus lisible.
- On peut également retirer les `NOT NULL` car certaines colonnes peuvent ne pas être remplies, comme par exemple la région ou sous-région, ou encore l'ISO si le pays n'en a pas. Je préfère retirer tous les `NOT NULL` pour que ce soit plus lisible.

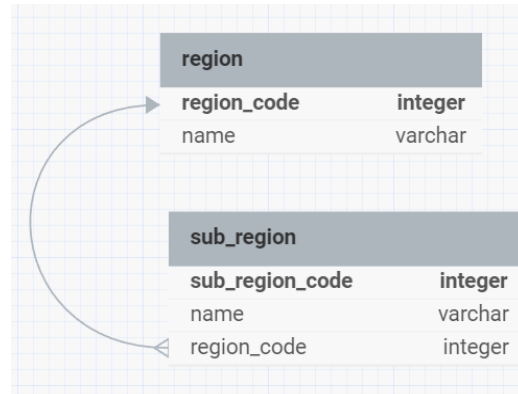
## Illustrations comparatives

### Association fonctionnelle

Quel AGL?



MCD association fonctionnelle

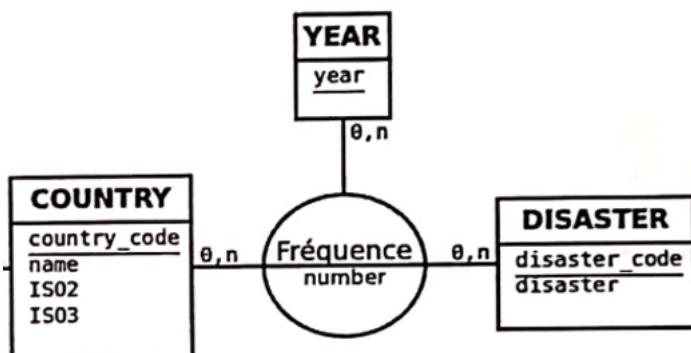


MPD association fonctionnelle

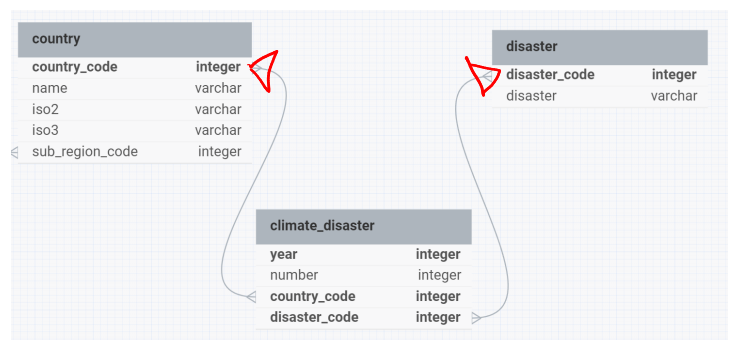
La différence entre nos 2 modélisations se joue surtout sur les différents éléments constitutifs chez l'un et chez l'autre.

Modèle conceptuel des données (MCD)	Modèle physique des données (MPD)
Possède des entités, <i>region</i> et <i>sub_region</i> .	Possède des tables, <i>region</i> et <i>sub_region</i> .
Les clés étrangères sont implicites grâce aux cardinalités.	Les clés étrangères sont visibles afin de pouvoir faire les relations.
Possède une association "situer" reliant <i>region</i> et <i>sub_region</i> .	Possède une relation matérialisée par la clé étrangère <i>region_code</i> dans <i>sub_region</i> , qui fait référence à <i>region_code</i> de <i>region</i> .

## Association maillée



MCD association maillé



MPD association maillé

Modèle conceptuel des données (MCD)	Modèle physique des données (MPD)
Possède des entités, <code>country</code> , <code>disaster</code> et <code>year</code> .	Possède des tables, <code>country</code> , <code>disaster</code> . Mais <code>year</code> est devenu l'attribut d'une table de relation <code>climate_disaster</code> .
Les clés étrangères sont implicites grâce aux cardinalités.	Les clés étrangères sont visibles afin de pouvoir faire les relations.
Possède une association "fréquence" avec son attribut <code>number</code> reliant <code>country</code> , <code>disaster</code> et <code>year</code> .	Possède une table de relation <code>climate_disaster</code> possédant en attribut la table <code>year</code> et l'attribut <code>number</code> faisant les relations entre les tables <code>country</code> et <code>disaster</code> . Les relations matérialisée par les clés étrangères sont <code>country_code</code> de <code>climate_disaster</code> qui fait référence à <code>country_code</code> dans <code>country</code> et <code>disaster_code</code> de <code>climate_disaster</code> qui fait référence à <code>disaster_code</code> de <code>disaster</code> .

## Peuplement des tables

### Modification du script de création de table

Pour le peuplement des tables, j'ai décidé de prendre le second jeu de données.

Tout d'abord je vais modifier mon script de création de table, la modification se fera sur la création de la table `disaster`, en effet aucun des 2 fichiers ne contient un id pour l'attribut `disaster`, je vais donc devoir reprendre ma création de table et remplacer le type `INTEGER` par `SERIAL` pour `disaster_code`, afin de permettre l'auto incrémentation de cette colonne lors du peuplement de la table `disaster`.

```
CREATE TABLE disaster (
  disaster_code SERIAL PRIMARY KEY,
  disaster VARCHAR
);
```

### Première partie du peuplement (fichier fig. 3)

#### Création des tables temporaires

Ensuite nous allons créer une table temporaire (se détruit automatiquement après fermeture du terminal) afin d'y mettre les informations nécessaires. Dans cette table, nous allons mettre les attributs présentes dans les colonnes du fichier CSV de la figure 3 (fichier qu'on va appeler "all.csv").

```
CREATE TEMP TABLE temp_all (
  country VARCHAR,
  iso2 VARCHAR,
  iso3 VARCHAR,
  country_code INTEGER,
```

```

isotemp VARCHAR,
region VARCHAR,
sub_region VARCHAR,
intermediate VARCHAR,
region_code INTEGER,
sub_region_code INTEGER,
intermediate_code INTEGER
);

```

Il faut que le nombre d'attributs de cette table corresponde au nombre de colonnes de notre fichier. C'est bien le cas ici. On peut d'ailleurs remarquer qu'il y aura des colonnes qu'on n'utilisera pas lors de l'insertion comme par exemple `isotemp`, ou encore `intermediate`, qui sont là seulement pour respecter le nombre de colonnes du fichier.

Ensuite nous devons importer les données du fichier CSV ("all.csv") dans notre table temporaire. Cependant dans ce fichier certains pays n'ont pas de région et de sous-région, c'est notamment le cas de l'Antarctique.

En essayant d'importer le fichier dans ma table temporaire, il m'explique ne pas accepter les chaînes vides. J'ai donc cherché des moyens de transformer mes chaînes vides en NULL. Cependant la version de mon PostgreSQL n'acceptait pas les différentes requêtes que je lui proposais, comme les `IF NULL`, etc. J'ai donc procédé autrement, j'ai créé une deuxième table temporaire comportant les mêmes attributs que la première, à une différence près, elles sont toutes de type `VARCHAR`.

```

CREATE TEMP TABLE temp_varchar (
country VARCHAR,
iso2 VARCHAR,
iso3 VARCHAR,
country_code VARCHAR,
isotemp VARCHAR,
region VARCHAR,
sub_region VARCHAR,
intermediate VARCHAR,
region_code VARCHAR,
sub_region_code VARCHAR,
intermediate_code VARCHAR
);

```

La création de cette table m'a permis d'importer le fichier "all.csv" sans erreurs, car il acceptait les chaînes vides sans problèmes. J'ai ensuite utilisé la commande `\copy` avec l'option `CSV HEADER`, qui indique que la première ligne du fichier n'est pas à mettre dans notre table, car ce sont les noms des colonnes du fichier.

```

\copy temp_varchar FROM 'all.csv' CSV HEADER;

```

Après cela on insère dans notre première table temporaire "temp\_all", celle ayant les attributs de types `INTEGER`, toutes les données de notre seconde table "temp\_varchar" mais avec une condition `NULLIF` et une conversion via `::INTEGER` (trouvés dans les sites Internet mis dans la source). `NULLIF(arg1, arg2)` va comparer les 2 arguments, s'il sont égaux (donc si notre donnée est égal à une chaîne vide) alors elle devient `NULL`, dans le cas contraire on garde notre donnée (arg1). Pour les données à convertir, de `VARCHAR` à `INTEGER`, on utilise `::INTEGER`.



```

INSERT INTO temp_all
SELECT
    NULLIF(country, ''),
    NULLIF(iso2, ''),
    NULLIF(iso3, ''),
    NULLIF(country_code, '')::INTEGER,
    NULLIF(isotemp, ''),
    NULLIF(region, ''),
    NULLIF(sub_region, ''),
    NULLIF(intermediate, ''),
    NULLIF(region_code, '')::INTEGER,
    NULLIF(sub_region_code, '')::INTEGER,
    NULLIF(intermediate_code, '')::INTEGER
FROM temp_varchar;

```

## Insertion des données dans les premières tables

Les tables que nous allons pouvoir remplir grâce à ce premier fichier CSV (fig. 3) seront les tables **region**, **sub\_region** et **country** car elle contient les colonnes qui nous intéresse. On va donc procéder aux insertions.

```

INSERT INTO region (region_code, name)
-- à insérer dans les colonnes de notre table region
SELECT DISTINCT region_code, region
-- valeurs insérées sans doublons
FROM temp_all
WHERE region_code IS NOT NULL -- évite les lignes NULL
AND region IS NOT NULL;

-- rien ne change par rapport à region
INSERT INTO sub_region (sub_region_code, name, region_code)
SELECT DISTINCT sub_region_code, sub_region, region_code
FROM temp_all
WHERE sub_region_code IS NOT NULL
AND sub_region IS NOT NULL
AND region_code IS NOT NULL;

INSERT INTO country (country_code, name, iso2, iso3, sub_region_code)
SELECT DISTINCT
    country_code,
    country AS name,
-- indique que la colonne country de la table temporaire
-- doit être insérée dans la colonne name de notre table country
    iso2,
    iso3,
    sub_region_code
FROM temp_all;

```

## Deuxième partie du peuplement (fichier fig. 2)

### Création d'une table temporaire

Maintenant pour remplir nos tables restantes, **disaster** et **climate\_disaster**, on va utiliser le fichier de fig. 2 qu'on va nommer "Indicator.csv".

Comme pour la première partie de peuplement, on va également créer une table temporaire pour y mettre les données du deuxième fichier. Encore une fois il faut le même nombre d'attributs dans la table temporaire que dans les colonnes du fichier, donc il faudra un attribut pour chaque année.

```
CREATE TEMP TABLE temp_indicator (  
  ObjectId INTEGER,  
  country VARCHAR,  
  iso2 VARCHAR,  
  iso3 VARCHAR,  
  indicator VARCHAR,  
  unit VARCHAR,  
  year_1980 INTEGER,  
  year_1981 INTEGER,  
  year_1982 INTEGER,  
  year_1983 INTEGER,  
  year_1984 INTEGER,  
  year_1985 INTEGER,  
  year_1986 INTEGER,  
  year_1987 INTEGER,  
  year_1988 INTEGER,  
  year_1989 INTEGER,  
  year_1990 INTEGER,  
  year_1991 INTEGER,  
  year_1992 INTEGER,  
  year_1993 INTEGER,  
  year_1994 INTEGER,  
  year_1995 INTEGER,  
  year_1996 INTEGER,  
  year_1997 INTEGER,  
  year_1998 INTEGER,  
  year_1999 INTEGER,  
  year_2000 INTEGER,  
  year_2001 INTEGER,  
  year_2002 INTEGER,  
  year_2003 INTEGER,  
  year_2004 INTEGER,  
  year_2005 INTEGER,  
  year_2006 INTEGER,  
  year_2007 INTEGER,  
  year_2008 INTEGER,  
  year_2009 INTEGER,  
  year_2010 INTEGER,  
  year_2011 INTEGER,  
  year_2012 INTEGER,  
  year_2013 INTEGER,  
  year_2014 INTEGER,  
  year_2015 INTEGER,  
  year_2016 INTEGER,  
  year_2017 INTEGER,  
  year_2018 INTEGER,  
  year_2019 INTEGER,  
  year_2020 INTEGER,  
  year_2021 INTEGER,  
  year_2022 INTEGER,  
  year_2023 INTEGER,  
  source VARCHAR  
);
```

On importe ensuite les données du fichier dans notre nouvelle table temporaire.

```
\copy temp_indicator FROM 'Indicator.csv' CSV HEADER;
```

## Insertion des données dans les tables restantes

Pour rappel, dans la table **disaster**, on a changé le type **INTEGER** par **SERIAL** pour l'attribut **disaster\_code** pour permettre l'auto incrémentation. Il se fera automatiquement donc pas besoin de le mettre dans mon insertion. Etant donné que dans la colonne **Indicator** du fichier, il y a toujours la même phrase qui se répète puis énonce le désastre, j'ai utilisé **SUBSTRING()** et les expressions régulières afin de ne garder que le désastre (et enlever 'TOTAL' par la même occasion).

**SUBSTRING(text FROM motif)** va utiliser l'expression régulière (motif) que je lui donne pour extraire une partie du texte contenu dans la colonne **Indicator** de la table. L'expression régulière est la suivante **'.\*: (.\*)'**.

- **.\*:** → n'importe quel texte jusqu'au deux-points ':'
- **' '** → un espace, car il y en a un entre les deux-points et le mot qui nous intéresse (le désastre).
- **(.\*)** → garde ce qu'il y a entre parenthèses, donc tout ce qu'il reste après la phrase retirée.

```
INSERT INTO disaster (disaster)
-- pas besoin de disaster_code, disaster suffit
SELECT DISTINCT
    SUBSTRING(indicator FROM '.*: (.*)')
FROM temp_indicator
WHERE indicator IS NOT NULL
    AND SUBSTRING(indicator FROM '.*: (.*)') != 'TOTAL';
-- si la partie extraite n'est pas 'TOTAL'
-- alors on l'insère
```

Pour finir, il ne nous reste plus que la table **climate\_disaster** à peupler. Cette fois on devra extraire les informations de différentes tables, **country\_code** de la table **country**, **disaster\_code** de la table **disaster**, **year** et **number** de la table temporaire. Le moyen que j'ai trouvé est d'extraire les informations pour chaque année. Je vais donner l'exemple avec 1980.

```
INSERT INTO climate_disaster (country_code, disaster_code, year, number)
SELECT
    c.country_code,
    d.disaster_code,
    1980 AS year,
    t.year_1980 AS number
FROM temp_indicator t
JOIN country c USING(iso3)
-- on fait le lien grâce au 'iso3' contenu dans les 2 tables
JOIN disaster d ON d.disaster = SUBSTRING(t.indicator FROM '.*: (.*)')
-- pour comparer la colonne de la table tempo avec un disaster
WHERE t.year_1980 IS NOT NULL;
-- ne sont pris en compte que les valeurs non NULL
```

Évidemment c'est long de taper le même bloc 43 fois, et si on était dans une étude plus grande et qu'on ajoutait plus d'années, taper à la main n'aurait pas été envisageable.

J'ai donc écrit une fonction python avec une boucle afin de m'éviter d'écrire un bloc pour chaque année :

```
def annee(debut, fin):
    while debut <= fin:
```

Très  
bonne  
idée

```


        print("INSERT INTO climate_disaster (country_code, disaster_code,
year, number)")
        print("SELECT")
        print("  c.country_code,")
        print("  d.disaster_code,")
        print("  ", debut, "AS year,")
        print("  t.year_ " + str(debut) + " AS number")
        #str(debut) pour éviter les espaces entre les string
        print("FROM temp_indicator t")
        print("JOIN country c USING(iso3)")
        print("JOIN disaster d ON d.disaster = SUBSTRING(t.indicator FROM
'.*: (.*)')")
        print("WHERE t.year_ " + str(debut) + " IS NOT NULL;")
        print()
        debut += 1

annee(1981, 2023)

```

Je colle ensuite le résultat de l'exécution de cette fonction dans le terminal pour obtenir toutes les données utiles à la table **climate\_disaster**. Il y avait évidemment des fonctions PostgreSQL qui m'aurait évité de taper un bloc d'instructions par année, mais j'ai préféré utiliser ce que je savais faire et comprenais.

## Fichier SQL complet du script de peuplement

 `peuplement.sql`

## Source

**NULLIF** : <https://neon.tech/postgresql/postgresql-tutorial/postgresql-nullif>

**Conversion** : <https://neon.tech/postgresql/postgresql-tutorial/postgresql-cast>

**SUBSTRING()** : <https://neon.tech/postgresql/postgresql-string-functions/postgresql-substring>