UNIVERSITÉ SORBONNE PARIS NORD

Base de Données et langage SQL

Situation d'Apprentissage et d'Évaluation (SAE) S104 Création d'une base de données

CLIMATE-RELATED DISASTERS FREQUENCY

BOURG
Solène

BUT 1 Info
Groupe CÉRYNIE

Table des matières :

2.1 Script manuel de création de la base de données

```
DROP TABLE CLIMATE_DISASTER;
DROP TABLE DISASTER;
DROP TABLE COUNTRY;
DROP TABLE SUB_REGION;
DROP TABLE REGION;

--CREATE DES TABLES DE LA BDD--

CREATE TABLE REGION (region_code INT PRIMARY KEY, name VARCHAR NOT NULL);

CREATE TABLE SUB_REGION (sub_region_code INT PRIMARY KEY, name VARCHAR NOT NULL, region_code INT
REFERENCES REGION (region_code) ON DELETE SET NULL);

CREATE TABLE COUNTRY (country_code INT PRIMARY KEY, name VARCHAR NOT NULL, ISO2 CHAR(2), ISO3
CHAR(3), sub_region_code INT REFERENCES SUB_REGION (sub_region_code) ON DELETE SET NULL);

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

CREATE TABLE CLIMATE_DISASTER (country_code INT REFERENCES COUNTRY (country_code) ON DELETE
CASCADE, disaster_code INT REFERENCES DISASTER (disaster_code) ON DELETE CASCADE, year INT, number
INT);

ALTER TABLE CLIMATE_DISASTER  ADD CONSTRAINT CLIMATE_DISASTER_pkey PRIMARY KEY
(country_code,disaster_code,year);
```

Voici ce que renvoie le terminal après utilisation de la commande **\i** pour lancer le script la première fois :

```
sae=> \i /home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql
psql:/home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql:1: ERROR:  table "climate_disaster" does not exist
psql:/home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql:2: ERROR:  table "disaster" does not exist
psql:/home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql:3: ERROR:  table "country" does not exist
psql:/home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql:4: ERROR:  table "sub_region" does not exist
psql:/home/solene/Bureau/BUT_1/BDD/SAE/bdd.sql:5: ERROR:  table "region" does not exist
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
sae=>
```

Évidemment, on constate que les premières instructions ‘DROP TABLE’ rencontrent une erreur puisque les tables n'existent pas. Cependant, le reste du script ne rencontre aucun problème.

On peut visionner les tables créées avec la commande **\d** :

```
sae=> \d
                  List of relations
 Schema |            Name             |   Type   | Owner
--------+-----------------------------+----------+--------
 public | climate_disaster            | table    | solene
 public | country                     | table    | solene
 public | disaster                    | table    | solene
 public | disaster_disaster_code_seq  | sequence | solene
 public | region                      | table    | solene
 public | sub_region                  | table    | solene
(6 rows)

sae=>
```

On compte les 5 tables provenant des ‘CREATE TABLE’ plus une, ‘disaster_disaster_code-seq’, dû au type ‘SERIAL’ de la clé primaire ‘disaster_code’ de la table ‘DISASTER’ qui crée un attribut unique automatiquement à chaque nouvel enregistrement.

On le constate en utilisant la commande **\d disaster_disaster_code_seq** :

```
sae=> \d disaster_disaster_code_seq
                Sequence "public.disaster_disaster_code_seq"
  Type   | Start | Minimum |  Maximum   | Increment | Cycles? | Cache
---------+-------+---------+------------+-----------+---------+-------
 integer |     1 |       1 | 2147483647 |         1 | no      |     1
Owned by: public.disaster.disaster_code
```

Le type ‘SERIAL’ peut supporter plus de 2.10^9 enregistrements ce qui est largement suffisant dans ce sujet puisqu’il n’y a que 6 lignes dans la table ‘DISASTER’.

2.2 Modélisation et script de création “avec AGL”

Pour réaliser cette SAE, je me suis servie de pgModeler. Pour comprendre le fonctionnement de cet AGL, je me suis informée auprès de tutoriels vidéos disponibles sur internet. Ainsi, j’ai pu comparer le modèle physique de pgModeler avec le modèle conceptuel présenté dans la ressource R104 Introduction BD.

Présentation des deux modèles

Le modèle conceptuel de données présente plusieurs éléments essentiels. Tout d’abord, les types-associations lient plusieurs type-entités entre eux pour associer les entités de l’un et de l’autre type-entité. Un type-entité est représenté sous forme d’un tableau avec le nom de la table, souvent un majuscule, puis les attributs.

Les clés primaires sont signalées en soulignant l’/les attribut(s) concernés(s).

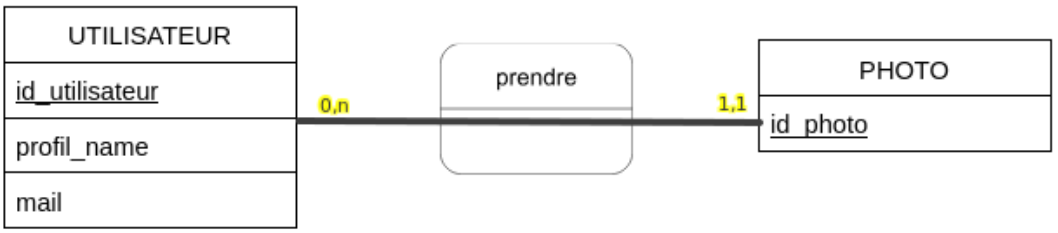
Pour savoir combien de fois une entité peut être, au moins et au plus, représentée dans une association, ce modèle se sert de cardinalité (‘0,1’ ‘0,n’ ‘1,1’ ‘1,n’).

Le modèle physique de données, bien que visuellement proche du modèle conceptuel, présente beaucoup de différences. On remarque tout de suite que la forme d’un tableau est ici aussi utilisée avec le nom de la table puis les attributs. Mais la première différence visible est l’absence d’un type-association entre les type-entités. Les associations se font grâce à une clé étrangère signalée en tant que contrainte sur un attribut. Ce lien est représenté par une ligne avec différentes terminaisons pour exprimer les cardinalités.

De plus, on peut noter qu'il y a beaucoup plus d'informations relatives aux attributs. La clé primaire est d'une couleur différente des autres attributs en précisant le type de donnée que l'on peut y entrer (ici, 'integer') et les lettres 'pk' (primary key) pour signaler la contrainte de l'attribut en tant que clé primaire.

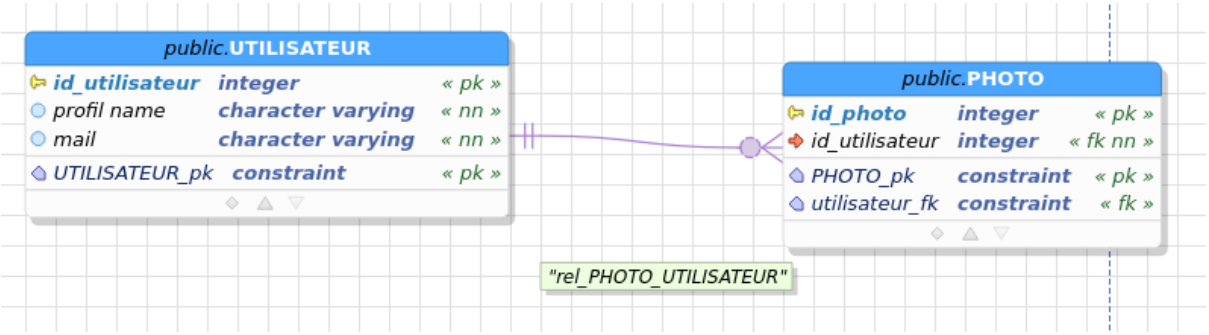
2.2.1 Association fonctionnelle

modèle conceptuel de données



Dans cet exemple, on a deux type-entités, 'UTILISATEUR' et 'PHOTO' ainsi qu'un type-association, 'prendre'.
On retrouve les cardinalités '0,n' et '1,1' signifiant respectivement qu'un 'UTILISATEUR' peut prendre aucune ou plusieurs photos tandis qu'une 'PHOTO' est forcément prise par une seule personne. Ainsi, la cardinalité '0,n' est placée du côté 'UTILISATEUR' et la '1,1' du côté 'PHOTO'.

modèle physique de données



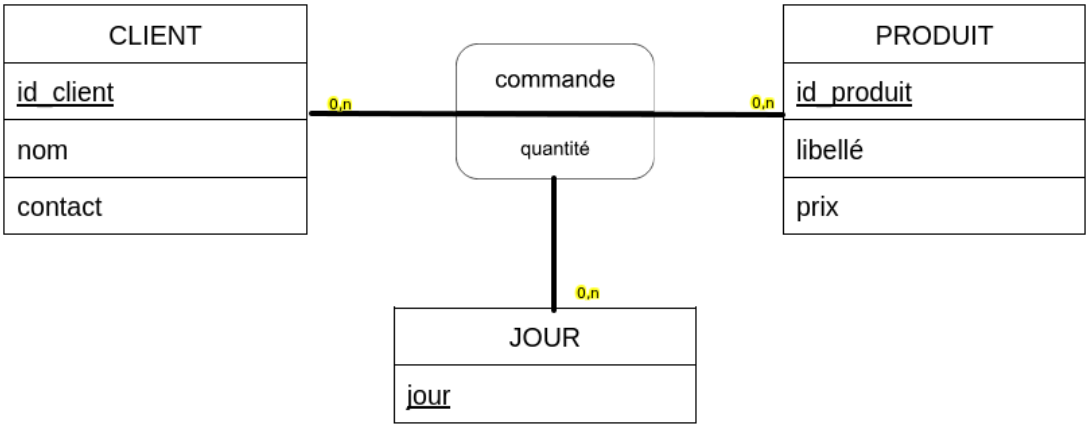
Dans ce modèle, du côté de la table ‘PHOTO’, on repère des ‘pattes d’oie’ avec un cercle. Ceci remplace la cardinalité ‘0,n’ (aucun ou plusieurs) et de l’autre côté deux traits verticaux qui représentent la cardinalité ‘1,1’ (un seul).

On remarque que la représentation des cardinalités est inversée : au lieu d'avoir les “pattes d’oie” du côté ‘UTILISATEUR’, comme dans le modèle conceptuel, elles sont placées en face de ‘PHOTO’.

C’est comme si on avait : ‘UTILISATEUR’ 1,1 —- —- —- 0,n ‘PHOTO’. Mais cela ne change pas la manière de lire la relation. Seule la représentation change.

2.2.2 Association maillée

modèle conceptuel de données



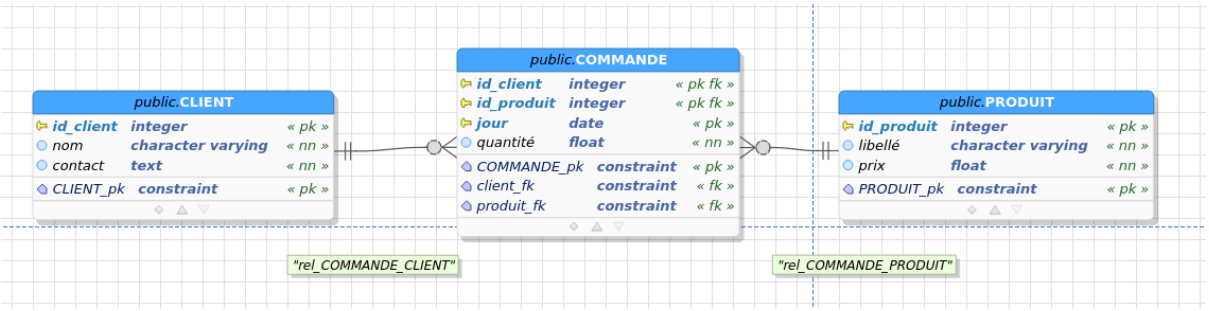
Dans cet exemple, on a trois type-entités, ‘CLIENT’, ‘PRODUIT’ et ‘JOUR’ et un type-association ‘commande’. Cette association maillée est modélisée avec la cardinalité ‘0,n’ sur chaque patte reliant le type-association avec les type-entités.

On remarque que ‘commande’ possède un attribut (ici, ‘quantité’). Ce type-association a trois clés primaires : les mêmes que celles des trois type-entités.

En effet, pour une association, il y aura un client, un produit et un jour et une quantité qui permettent d'instancier plusieurs commandes différentes.

On lit la relation ainsi, un ‘CLIENT’ peut commander plusieurs produits à différents jours. Un ‘PRODUIT’ peut être commandé par plusieurs clients à différents jours. Dans un ‘JOUR’ plusieurs produits peuvent être commandés par différents clients.

modèle physique de données



Ce modèle simplifie le modèle conceptuel en supprimant la table ‘JOUR’.

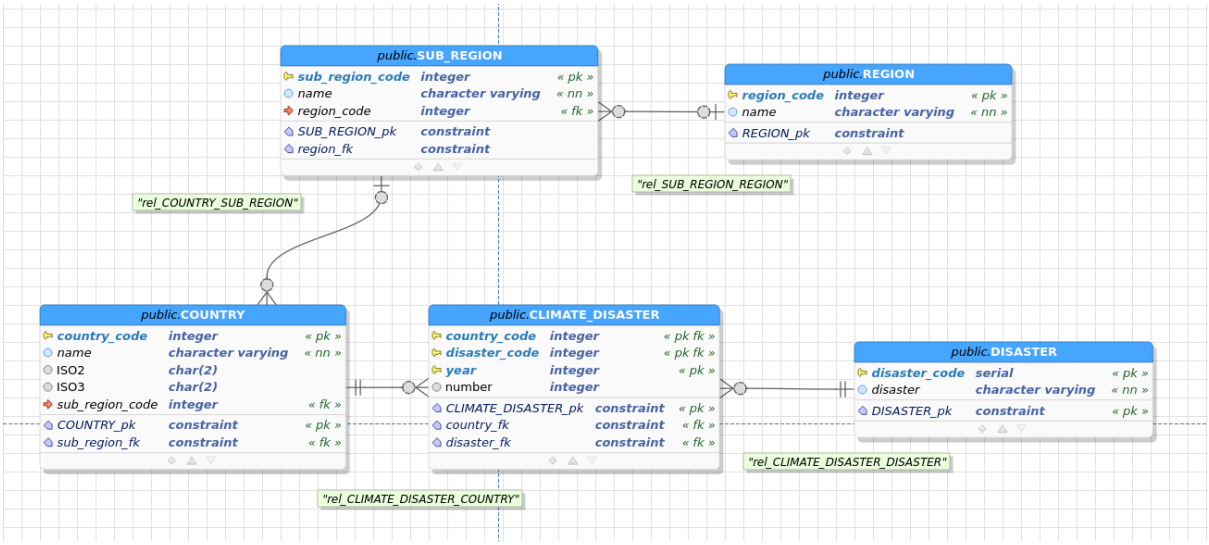
Ici, on voit clairement les clés primaires de la table ‘COMMANDE’ reliées aux clé primaires des tables ‘CLIENT’ et ‘PRODUIT’ : ce sont des clés étrangères. On peut le voir avec les lettres ‘fk’ (foreign key) à côté des clés primaires.

Remarquons que la clé primaire ‘jour’ ne fait pas référence à une table ‘JOUR’, c’est une simple clé primaire.

De la même manière que l’association fonctionnelle précédente, les terminaisons des liens entre les tables sont inversées. Du côté ‘CLIENT’ et ‘PRODUIT’, on voit les deux traits verticaux qui représentent la cardinalité ‘1,1’ alors que dans l’autre modèle c’est la cardinalité ‘0,n’ à côté de ‘CLIENT’.

On lit la relation ainsi : un ‘CLIENT’ peut commander plusieurs fois à différents moments, une ‘COMMANDE’ est forcément adressée à un client, un ‘PRODUIT’ peut être commandé plusieurs fois à différents moments et une ‘COMMANDE’ comporte forcément un produit.

2.2.3 Modèle physique de données réalisé avec l'AGL



2.2.4 Script SQL de création de tables généré automatiquement par l'AGL

```
-- Database generated with pgModeler (PostgreSQL Database Modeler).
-- pgModeler version: 0.9.4
-- PostgreSQL version: 13.0
-- Project Site: pgmodeler.io
-- Model Author: ---

-- Database creation must be performed outside a multi lined SQL file.
-- These commands were put in this file only as a convenience.
--
-- object: sae | type: DATABASE --
DROP DATABASE IF EXISTS sae;
CREATE DATABASE sae;
-- ddl-end --


-- object: public."COUNTRY" | type: TABLE --
DROP TABLE IF EXISTS public."COUNTRY" CASCADE;
CREATE TABLE public."COUNTRY" (
	country_code integer NOT NULL,
	name character varying NOT NULL,
	"ISO2" char(2),
	"ISO3" char(2),
```

```
	sub_region_code integer,
	CONSTRAINT "COUNTRY_pk" PRIMARY KEY (country_code)
);
-- ddl-end --
ALTER TABLE public."COUNTRY" OWNER TO postgres;
-- ddl-end --

-- object: public."DISASTER" | type: TABLE --
DROP TABLE IF EXISTS public."DISASTER" CASCADE;
CREATE TABLE public."DISASTER" (
	disaster_code serial NOT NULL,
	disaster character varying NOT NULL,
	CONSTRAINT "DISASTER_pk" 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" (
	country_code integer NOT NULL,
	disaster_code integer NOT NULL,
	year integer NOT NULL,
	number integer,
	CONSTRAINT "CLIMATE_DISASTER_pk" PRIMARY KEY (country_code,disaster_code,year)
);
-- ddl-end --
ALTER TABLE public."CLIMATE_DISASTER" 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 character varying NOT NULL,
	region_code integer,
	CONSTRAINT "SUB_REGION_pk" PRIMARY KEY (sub_region_code)
);
-- ddl-end --
ALTER TABLE public."SUB_REGION" OWNER TO postgres;
-- ddl-end --

-- object: public."REGION" | type: TABLE --
DROP TABLE IF EXISTS public."REGION" CASCADE;
CREATE TABLE public."REGION" (
	region_code integer NOT NULL,
	name character varying NOT NULL,
	CONSTRAINT "REGION_pk" PRIMARY KEY (region_code)
);
-- ddl-end --
ALTER TABLE public."REGION" OWNER TO postgres;
-- ddl-end --

-- object: sub_region_fk | type: CONSTRAINT --
ALTER TABLE public."COUNTRY" DROP CONSTRAINT IF EXISTS sub_region_fk CASCADE;
ALTER TABLE public."COUNTRY" ADD CONSTRAINT sub_region_fk FOREIGN KEY (sub_region_code)
REFERENCES public."SUB_REGION" (sub_region_code) MATCH SIMPLE
ON DELETE SET NULL ON UPDATE NO ACTION;
-- ddl-end --

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

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

```
-- object: region_fk | type: CONSTRAINT --
ALTER TABLE public."SUB_REGION" DROP CONSTRAINT IF EXISTS region_fk CASCADE;
ALTER TABLE public."SUB_REGION" ADD CONSTRAINT region_fk FOREIGN KEY (region_code)
REFERENCES public."REGION" (region_code) MATCH SIMPLE
ON DELETE SET NULL ON UPDATE NO ACTION;
-- ddl-end --
```

Dans ce script généré par pgModeler, on remarque plusieurs éléments non utilisés dans le script manuel. D'abord, pouvoir 'DROP' une table uniquement si elle existe et ainsi ignorer la commande dans le cas contraire et non pas lancer la commande même si la table n'existe déjà pas. En effet, cela engendre des erreurs disant que la table n'existe pas et qu'on ne peut donc pas la supprimer.
Également, le fait de pouvoir 'DROP' une altération de table est quelque chose que je ne savais pas possible ni utile.

De plus, la manière de préciser les clés primaires est différente. Alors que dans mon script manuel j'ai simplement utilisé 'PRIMARY KEY' sur mon attribut, ici c'est une contrainte de table qui est utilisée pour désigner la clé primaire de la table.

Les clés étrangères ne sont pas mentionnées directement dans la création de la table sur chaque attribut concerné mais en tant qu'altération de table. Là où, dans le script manuel, les trois clés primaires de la table 'CLIMATE_DISASTER' sont précisées en tant qu'altération de table, le script de pgModeler les insère en tant que contrainte de table.

L'ordre de création de table et d'altérations de table n'a pas d'impact dans ce script. En effet, elles apparaissent dans le même ordre qu'elles ont été créées manuellement sur pgModeler. Ainsi, le 'DROP' se fait en 'CASCADE' et supprime donc les tables associées également sans se soucier de l'ordre.

Voici ce que renvoie le terminal après le lancement du script de l'AGL avec la commande **\i** :

```
sae=> \d
Did not find any relations.
sae=> \i Bureau/BUT_1/BDD/SAE/database-sae.sql
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:11: ERROR:  cannot drop the currently open database
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:12: ERROR:  database "sae" already exists
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:17: NOTICE:  table "COUNTRY" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:27: ERROR:  must be member of role "postgres"
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:31: NOTICE:  table "DISASTER" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:38: ERROR:  must be member of role "postgres"
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:42: NOTICE:  table "CLIMATE_DISASTER" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:51: ERROR:  must be member of role "postgres"
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:55: NOTICE:  table "SUB_REGION" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:63: ERROR:  must be member of role "postgres"
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:67: NOTICE:  table "REGION" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:74: ERROR:  must be member of role "postgres"
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:78: NOTICE:  constraint "sub_region_fk" of relation "COUNTRY" does not exist, skipping
ALTER TABLE
ALTER TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:85: NOTICE:  constraint "country_fk" of relation "CLIMATE_DISASTER" does not exist, skipping
ALTER TABLE
ALTER TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:92: NOTICE:  constraint "disaster_fk" of relation "CLIMATE_DISASTER" does not exist, skipping
ALTER TABLE
ALTER TABLE
psql:Bureau/BUT_1/BDD/SAE/database-sae.sql:99: NOTICE:  constraint "region_fk" of relation "SUB_REGION" does not exist, skipping
ALTER TABLE
ALTER TABLE
```

Lors du test du script, toutes les tables sont créées sans soucis. On note les quelques erreurs sur le ‘DROP DATABASE’ qui ne peut ‘DROP’ la database ‘sae’ dans laquelle on se situe pour lancer le script. Également, pour l’instruction ‘ALTER TABLE public."[....]" OWNER TO postgres’ une erreur apparaît au niveau du rôle “postgres”. L’utilité de cette instruction répétée pour chaque table est difficile à cerner. Cela porte probablement sur la colonne “Owner” que l’on peut voir avec la commande **\d**, faisant apparaître le nom du propriétaire, ici ‘solene’, pour chaque table.
Cette instruction prévoit sûrement de changer cet élément pour “postgres”, mais rien n’est moins sûr. On peut tout de même noter que cela n’empêche pas la création des tables.

Visionnons les tables créées avec la commande **\d** :

```
sae=> \d
                  List of relations
 Schema |            Name             |   Type   | Owner
--------+-----------------------------+----------+--------
 public | CLIMATE_DISASTER            | table    | solene
 public | COUNTRY                     | table    | solene
 public | DISASTER                    | table    | solene
 public | DISASTER_disaster_code_seq  | sequence | solene
 public | REGION                      | table    | solene
 public | SUB_REGION                  | table    | solene
(6 rows)
```

2.3 Peuplement des tables

```
--TABLES TEMPORAIRES ACCUEILLANT LES DONNÉES DES FiCHIERS .CSV --

CREATE TABLE tmp (name VARCHAR, alpha_2 VARCHAR, alpha_3 VARCHAR, country_code INT, iso_3166_2
VARCHAR,region varchar, sub_region VARCHAR, intermediate_region VARCHAR, region_code INT, sub_region_code INT,
intermediate_region_code INT);

\copy tmp FROM doc/github.csv DELIMITER ',' CSV HEADER

CREATE TABLE tmp2 (country VARCHAR, iso2 CHAR(2), iso3 CHAR(3), region_code INT, region VARCHAR,
sub_region_code INT, sub_region VARCHAR, disaster VARCHAR, year INT, number INT);

\copy tmp2 FROM doc/Climate.csv DELIMITER ',' CSV HEADER

--INSERT DES DONNÉES DANS LA BDD--

INSERT INTO REGION (name,region_code) SELECT DISTINCT region,region_code FROM tmp WHERE region_code IS
NOT NULL;

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

INSERT INTO COUNTRY (name,country_code,sub_region_code) SELECT DISTINCT
name,country_code,sub_region_code FROM tmp WHERE sub_region_code IS NOT NULL;
UPDATE COUNTRY SET ISO2=tmp2.iso2, ISO3=tmp2.iso3 FROM tmp2 WHERE COUNTRY.name=tmp2.country;

INSERT INTO DISASTER (disaster) SELECT DISTINCT disaster FROM tmp2 WHERE tmp2.disaster IS NOT NULL;

INSERT INTO climate_disaster (country_code, disaster_code, year,number) SELECT country_code, disaster_code,year,
number FROM country, disaster, tmp2 WHERE tmp2.country=country.name AND tmp2.disaster=disaster.disaster AND
tmp2.year IS NOT NULL AND tmp2.number IS NOT NULL GROUP BY (country_code, disaster_code, year, number);

--SUPPRESSION DES TABLES TEMPORAIRES--

DROP TABLE tmp;
DROP TABLE tmp2;
```

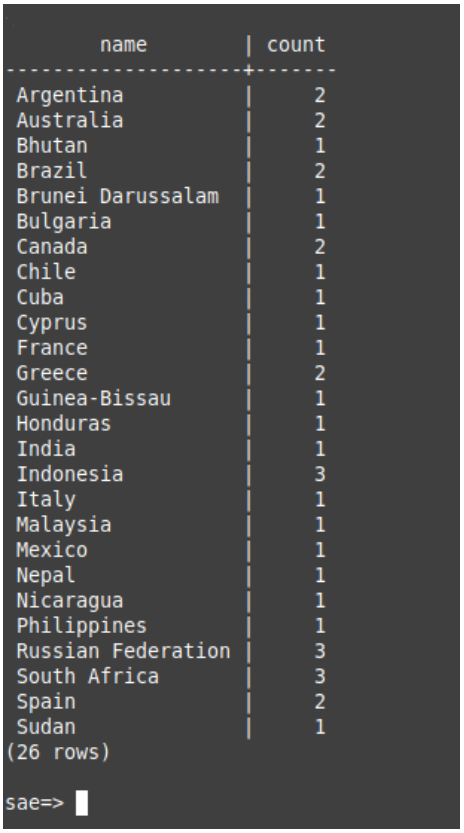
Ce script permet de peupler la base de données à partir des fichiers .csv mis à disposition. Il fonctionne avec le script manuel. *(Voir partie 2.1)*

Voici ce que renvoie le terminal après utilisation de la commande **\i** pour lancer le script :

```
sae=> \i Bureau/BUT_1/BDD/SAE/script.sql
CREATE TABLE
COPY 249
CREATE TABLE
COPY 6448
INSERT 0 5
INSERT 0 17
INSERT 0 247
UPDATE 142
INSERT 0 6
INSERT 0 4518
DROP TABLE
DROP TABLE
sae=>
```

On peut tester quelques requêtes SQL :

```
SELECT country.name, COUNT(country.country_code) FROM country
JOIN climate_disaster
ON country.country_code = climate_disaster.country_code
JOIN disaster
ON climate_disaster.disaster_code = disaster.disaster_code
WHERE disaster.disaster='Wildfire'
AND climate_disaster.year>='1998' AND year<='2000'
GROUP BY country.name;
```



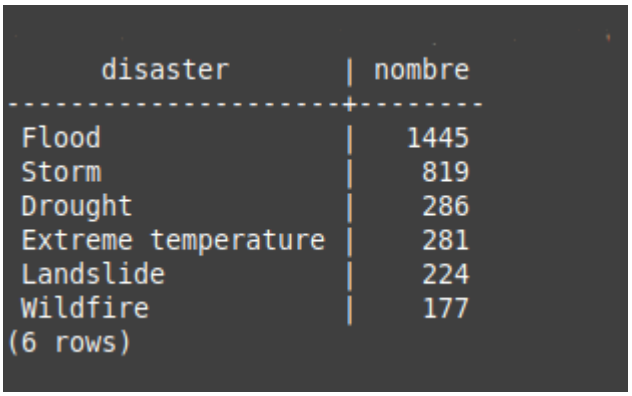
name	count
Argentina	2
Australia	2
Bhutan	1
Brazil	2
Brunei Darussalam	1
Bulgaria	1
Canada	2
Chile	1
Cuba	1
Cyprus	1
France	1
Greece	2
Guinea-Bissau	1
Honduras	1
India	1
Indonesia	3
Italy	1
Malaysia	1
Mexico	1
Nepal	1
Nicaragua	1
Philippines	1
Russian Federation	3
South Africa	3
Spain	2
Sudan	1

(26 rows)

sae=>

>> Renvoie la liste des pays ayant connu des feux de forêt entre 1998 et 2000 ainsi que le nombre de fois où ils ont été touchés.

```
SELECT disaster, COUNT(disaster) AS nombre FROM disaster JOIN climate_disaster ON disaster.disaster_code = climate_disaster.disaster_code WHERE year>='1998' AND year<='2024' GROUP BY disaster ORDER BY nombre DESC;
```



disaster	nombre
Flood	1445
Storm	819
Drought	286
Extreme temperature	281
Landslide	224
Wildfire	177

(6 rows)

>> Renvoie combien de fois chaque désastre a touché un pays entre 1998 et 2024 rangé par ordre décroissant.