

Rapport - SAé 2.04

MULLIER Mathys - Giorgio UTZERI

BUT Informatique - IUT de Lille

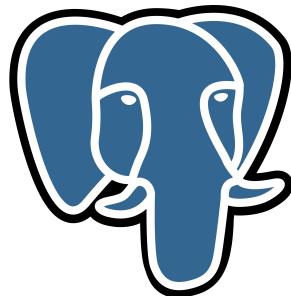


Table des matières

1 - Introduction

2 - Phase de prise en main et de compréhension des données

- 2.1 - Analyse du fichier de données
- 2.2 - Importation des données dans PostgresSQL

3 - Ventilation des données

- 3.1 - Suppression des colonnes calculables
- 3.2 - Création du MCD
- 3.3 - Passage d'un MCD aux tables
- 3.4 - Ecriture d'un script de création des tables
 - 3.1 - Téléchargement du fichier source
 - 3.2 - Création des nouvelles tables
 - 3.3 - Ventilation des données
 - 3.4 - Comparaison des tailles

4 - Requêteage

5 - Annexes

Introduction

Dans ce compte rendu technique, nous allons expliquer les différentes étapes de cette SAé 204, allant de la prise en main d'un fichier de plusieurs milliers de lignes à la création d'une base de données pouvant être interrogée à l'aide de requêtes SQL.

Pour cette SAé, notre objectif est d'exploiter un fichier de données et de les manipuler afin de créer une base de données à l'aide du Système de Gestion de Base de Données (SGBD) PostGreSQL.

Ce fichier contient de manière brute toutes les informations et les statistiques des phases d'admission pour les formations de l'enseignement supérieur sur Parcoursup en 2023. Il représente donc une quantité importante de données. De plus, bien que ces données soient fournies par un site officiel de l'État, elles ne sont pas très bien structurées pour une utilisation en tant que base de données. Nous devons donc, après avoir analysé la structure des différentes données, remodéliser la structure de données sous forme de tables, en évitant toute redondance, avant de ventiler les données dans ces tables.

En résumé, les différentes étapes sont donc : importer les données brutes, concevoir un Modèle Conceptuel de Données (MCD) évitant les redondances du fichier initial, ventiler les données initiales, et enfin interroger le résultat obtenu à l'aide de requêtes SQL.

Phase de prise en main et de compréhension des données

Analyse du fichier de données

Pour ce qui est de la première étape, nous allons prendre en main et comprendre, à travers des questions et à l'aide de commandes Unix, la structure du fichier téléchargé contenant toutes les données brutes.

Toutefois, avant toute chose, il est de bon usage de vérifier que le fichier correspond bien à l'encodage souhaité. Pour cela, on exécute la commande :

```
file -i <chemin>
```

Ici, le système nous renvoie l'information que le fichier est encodé en UTF-8, le système d'encode souhaité, nous pouvons donc passer à la suite.

Question 1 - Combien y-a t-il de lignes ?

Pour connaître le nombre de lignes d'un fichier, une solution est d'utiliser la commande UNIX suivante :

```
cat <chemin> | wc -l
```

On utilise donc cette commande sur le fichier de données et obtient comme résultat 13870. Il y a donc évidemment 13870 lignes dans ce fichier. De ce résultat il est important de considérer qu'il contient une ligne d'en-tête dans notre fichier. On en déduit donc qu'il y a 13869 enregistrements dans ce fichier.

Question 2 - Que représente une ligne ?

Pour savoir ce que représente une ligne, il est intéressant de connaître la natures des différentes colonnes qui la compose. Ainsi, nous avons décidé de s'intéresser à la ligne d'en-tête du fichier ainsi qu'au premier enregistrement afin de connaître le nom d'une colonne et un exemple pour chacune d'entre-elles. Ces lignes sont les deux premières lignes du fichier. En Unix, pour afficher uniquement les deux premières lignes on peut utiliser la commande :

```
head -n 2 <chemin>
```

A la lecture des ces deux lignes, on devine facilement, au vu du contexte de Parcoursup, qu'une ligne correspond à une formation dans un établissement (le BUT Informatique de l'IUT de Lille est une ligne différente que celle du BUT Informatique de l'IUT de Lens).

Question 3 - Combien y-a t-il de colonnes ?

Pour connaître le nombre de colonnes du fichier, une première approche serait d'utiliser la commande unix suivante :

```
cat <chemin> | head -n 1 | tr ';' '\n' | wc -l
```

Cette commande récupère la première ligne du fichier, remplace les points virgules par un retour à la ligne ce qui permet d'avoir un nom de colonne par ligne et compte donc logiquement le nombre de lignes. Après execution de la commande, on obtient comme résultat 118. Il y a donc 118 colonnes dans ce fichier.

Question 4 - Quelle colonne identifie un établissement ?

Pour connaître de manière plus simple, pour tout les questions suivantes et pour toutes les manipulations futures, nous allons créer un fichier **dico.xls** dans lequel nous allons associer une valeur nX (avec X un entier initialisé à 1 et incrementé de 1 à chaque occurrence) à chaque nom de colonne. Ce fichier nous permettra de lire bien plus aisement les noms de colonnes et nous permettra de pouvoir faire la correspondance entre les différentes colonnes lors des requêtes SQL qui auront pour but de ventiler les données après importation.

Pour créer ce fichier, on peut utiliser le script Java mis en **annexe**.

Attention : la version du *dico.xls* que nous vous avons rendu contient d'autres colonnes que celles générées par le script qui lui ne fait que fournir un dico contenant uniquement le nom de la table import et le nom dans le fichier d'origine.

Pour revenir à la question, en utilisant le fichier fraîchement généré, on devine qu'un établissement peut-être identifié de manière explicite par la colonne "Établissement" (n4) qui correspond au nom de l'établissement. Toutefois un établissement est identifié de manière unique par la colonne "Code UAI de l'établissement" (n3).

Question 5 - Quelle colonne identifie une formation ?

Pour cette question, on peut également consulter le fichier dico.xls. Une formation est donc identifiée par la colonne .

Question 6 - Combien de lignes font référence à notre BUT Informatique ?

Pour obtenir le nombre de BUT Informatique, on peut exécuter la commande suivante :

```
cat <chemin> | grep "BUT - Informatique" | wc -l
```

Cette commande récupère le contenu du fichier donné en paramètres en ne gardant uniquement les lignes contenant l'expression régulière "BUT - Informatique". Toutefois, au lieu d'afficher le contenu sur la sortie standard du terminal, cette commande renvoie le nombre de lignes contenues ce qui permet concrètement de compter uniquement les formations ayant l'expression "BUT - Informatique" dans ces lignes.

Après exécution de cette commande sur notre fichier, on obtient un résultat de 49 ce qui correspond au nombre de BUT Informatique en France.

Pour ce qui est de notre BUT Informatique, il a comme spécificité de se dérouler dans la ville de Villeneuve-d'Ascq. On peut donc ajouter plus de précision dans la recherche en exécutant la commande suivante pour notre fichier :

```
cat <chemin> | grep "BUT - Informatique" | grep "Villeneuve-d'Ascq" | wc -l
```

Après exécution de la commande, on obtient comme résultat 1 ce qui vient bien valider le fait que le fichier contient une ligne par formation.

Question 7 - Quelle colonne identifie un département ?

Pour connaître quelle colonne identifie le département de l'établissement, on utilise également le fichier dico.xls. De ce fichier, on extrait les lignes suivantes :

```
n5;Code départemental de l'établissement;null;varchar(3);oui;codeDepartements;Departements  
n6;Département de l'établissement;null;varchar(50);oui;Departements;Departements
```

Ces deux colonnes permettent d'identifier que les colonnes "Code départemental de l'établissement" et "Département de l'établissement" identifient le département de l'établissement.

Question 8 - Comment envisagez vous importer ces données ?

Pour importer les données dans PostGreSQL la meilleure de solution est de créer une table temporaire nommée **import** contenant autant de colonnes que le fichier contenant les données brutes. L'avantage d'une table temporaire est quelle se supprimera, elle et ses données, à la fin de la session PostGreSQL. Chaque colonne aura dans un premier temps un type défini à **text**, toutes les données étant importables en tant que texte (pour générer directement le script sql permettant de créer la table import et ses 118 colonnes, on utilise un programme Java détaillé en annexe de ce fichier). Une fois la table créée il faudra essayer d'importer les données en utilisant les commandes fournis par notre SGBD. Une fois ces données importées, nous devons alors affiner les types de chaque colonne puis à chaque fois essayer de les importer avec

la même commande et ajuster les types provoquant des erreurs. Ces types seront inscrits dans le fichier **dico.xls**.

Question 9 - Quels problèmes identifiez vous dans ces données initiales ?

Les principaux problèmes rencontrés lors de l'importation des données sont les différentes colonnes ayant beaucoup de lignes contenant des valeurs nulles rendant leur typage compliqué ainsi que le nombre d'erreur de typage rencontrés. Toutefois ces problèmes se résolvent avec un peu (parfois beaucoup) de patience.

Importation des données dans PostGresSQL

Première étape : Génération d'un fichier dico.xls

Comme expliqué dans la question 4 de l'exercice précédent, nous avons généré le fichier dico.xls à l'aide d'un programme Java détaillé en annexe. Ce fichier permet de faire la correspondance entre les numéros de colonnes et les noms du fichier initial.

Deuxième étape : Création d'une table import

Comme expliqué dans la question 8 de l'exercice précédent, nous avons généré tout d'abord un script SQL qui créer une table import ayant 118 colonnes avec chacune un type text. Le script ressemblait à ça :

```
CREATE TEMP TABLE import (
    n1 TEXT,
    n2 TEXT,
    n3 TEXT,
    ...
    n118 TEXT
);
```

Il faut donc maintenant exécuter le script puis importer les données avec la commande PostGreSql suivante :

```
\COPY import FROM <chemin> WITH (DELIMITER ';', NULL '"', HEADER)
```

Troisième étape : Amélioration de la table import

Après avoir importé les données sous forme de texte, il faut maintenant affiner au maximum les types afin de pouvoir manipuler les données avec les outils les plus adaptés à chaque type. Après de nombreuses importations et tests non fructueux, nous sommes arrivés au résultat suivant :

```
DROP TABLE IF EXISTS import;

CREATE TEMP TABLE import (
    n1 INT,
    n2 VARCHAR(50),
    n3 VARCHAR(8),
    n4 VARCHAR(200),
    n5 VARCHAR(3),
    n6 VARCHAR(50),
    n7 VARCHAR(50),
    n8 VARCHAR(50),
```

```
n9 VARCHAR(40),  
n10 VARCHAR(300),  
n11 VARCHAR(50),  
n12 VARCHAR(50),  
n13 VARCHAR(500),  
n14 VARCHAR(100),  
n15 VARCHAR(200),  
n16 VARCHAR(200),  
n17 VARCHAR(30),  
n18 INT,  
n19 INT,  
n20 INT,  
n21 INT,  
n22 INT,  
n23 INT,  
n24 INT,  
n25 INT,  
n26 INT,  
n27 INT,  
n28 INT,  
n29 INT,  
n30 INT,  
n31 INT,  
n32 INT,  
n33 INT,  
n34 INT,  
n35 INT,  
n36 INT,  
n37 INT,  
n38 INT,  
n39 INT,  
n40 INT,  
n41 INT,  
n42 INT,  
n43 INT,  
n44 INT,  
n45 INT,  
n46 INT,  
n47 INT,  
n48 INT,  
n49 INT,  
n50 INT,  
n51 NUMERIC,  
n52 NUMERIC,  
n53 NUMERIC,  
n54 INT,  
n55 INT,  
n56 INT,  
n57 INT,  
n58 INT,  
n59 INT,  
n60 INT,  
n61 INT,  
n62 INT,  
n63 INT,  
n64 INT,
```

```
n65 INT,  
n66 NUMERIC(5,2),  
n67 INT,  
n68 INT,  
n69 INT,  
n70 INT,  
n71 INT,  
n72 INT,  
n73 INT,  
n74 NUMERIC(5,2),  
n75 NUMERIC(5,2),  
n76 NUMERIC(5,2),  
n77 NUMERIC(5,2),  
n78 NUMERIC(5,2),  
n79 NUMERIC(5,2),  
n80 NUMERIC(5,2),  
n81 NUMERIC(5,2),  
n82 NUMERIC(5,2),  
n83 NUMERIC(5,2),  
n84 NUMERIC(5,2),  
n85 NUMERIC(5,2),  
n86 NUMERIC(5,2),  
n87 NUMERIC(5,2),  
n88 NUMERIC(5,2),  
n89 NUMERIC(5,2),  
n90 NUMERIC(5,2),  
n91 NUMERIC(5,2),  
n92 NUMERIC(5,2),  
n93 NUMERIC(5,2),  
n94 NUMERIC(5,2),  
n95 NUMERIC(5,1),  
n96 NUMERIC(5,2),  
n97 NUMERIC(5,2),  
n98 NUMERIC(5,2),  
n99 NUMERIC(5,2),  
n100 NUMERIC(5,2),  
n101 NUMERIC,  
n102 VARCHAR(50),  
n103 NUMERIC,  
n104 VARCHAR(50),  
n105 INT,  
n106 VARCHAR(50),  
n107 INT,  
n108 VARCHAR(100),  
n109 VARCHAR(20),  
n110 INT,  
n111 TEXT,  
n112 TEXT,  
n113 NUMERIC(5,2),  
n114 NUMERIC(5,2),  
n115 NUMERIC(5,2),  
n116 NUMERIC(5,2),  
n117 VARCHAR(10),  
n118 VARCHAR(10)  
);
```

Ce code est évidemment retrouvable dans le fichier **parcoursup.sql**.

Quatrième étape : Importation des données

Nous pouvons donc, maintenant que nous avons les types affinés comme il le faut, importer les données. Pour cela, on réutilise la même commande déjà utilisée les des importations de "test" qui est :

```
\COPY import FROM <chemin> WITH (DELIMITER ';', NULL '', HEADER)
```

Cinquième étape : Quelques requêtes

Maintenant que nos données sont importées nous avons la possibilité d'exécuter des requêtes SQL sur nos données. Nous allons donc pour chaque question donner la requête SQL permettant d'obtenir le résultat ainsi que le résultat.

Combien il y a de formations gérés par ParcourSup ?

On utilise la requête suivante :

```
SELECT COUNT(DISTINCT n10) FROM import;
```

Cette requête permet de compter les valeurs distinctes non nulles de la colonne n10 de la table import qui est la colonne permettant d'identifier une filière de formation. On obtient comme résultat 3207. Il y a donc 3207 filières de formation. Toutefois, si la question demande de compter le nombre de formations au total, en distinguant les formations appartenant au même filière de formation (en distinguant par exemple le BUT Informatique de Lille et le BUT Informatique de Lens), on doit exécuter la requête :

```
SELECT COUNT(n10) FROM import;
```

Cette requête permet de lire les lignes ayant une valeur pour la colonne n10 de la table import non nulle. On retrouve donc le même résultat que lors des questions préliminaires soit 13869.

Combien il y a d'établissements gérés par ParcourSup ?

On utilise la requête suivante :

```
SELECT COUNT(DISTINCT n3) FROM import
```

Cette requête permet de compter les valeurs distinctes non nulles de la colonne n3 de la table import qui est la colonne permettant d'identifier un établissement. On obtient comme résultat 3965 ce qui correspond au nombre d'établissements gérés par ParcourSup.

Combien il y a de formations pour l'Université de Lille ?

On utilise la requête suivante :

```
SELECT COUNT(n10) FROM import  
WHERE n4='Université de Lille';
```

Cette requête permet de compter les lignes ou les valeurs de la colonne n10 sont non nulles et pour lesquelles la colonne n4, identifiant le nom de l'établissement, est égale à '**Université de Lille**'.

Ici, il est important de ne pas utiliser de **DISTINCT** car certaines formations comme les licences PASS sont divisées selon leurs options et chacune de ces divisions est considérée comme une formation différente. Toutefois, la colonne n10 ne contient pas d'information permettant d'identifier cette subtilité **DISTINCT**. De plus, une formation totalement similaire ne sera pas deux fois présente dans le même établissement, il n'y aurait donc aucun intérêt à utiliser le mot-clé.

Après exécution, on obtient donc comme résultat 124 ce qui correspond au nombre total de formations pour l'Université de Lille proposées sur ParcourSup.

Combien il y a de formations pour notre IUT ?

On utilise la requête suivante :

```
SELECT COUNT(n10) FROM import  
WHERE n4='Institut universitaire de technologie de Lille - Université de Lille';
```

Après exécution de cette commande, on obtient comme résultat qu'il existe 10 formations différentes liées à notre IUT.

Quel est le code du BUT Informatique de l'Université de Lille ?

On utilise la requête suivante :

```
SELECT n110 FROM import  
WHERE n4='Institut universitaire de technologie de Lille - Université de Lille'  
AND n10='BUT - Informatique';
```

Après exécution de cette commande, on obtient comme résultat que le code du BUT Informatique de l'Université de Lille est **6888**.

Citez 5 colonnes contenant des valeurs nulles

Pour cette question, nous avons relevé tout au long de notre exploitation des données pour les questions précédentes quelques lignes ayant des valeurs nulles. Toutefois, pour bien vérifier le résultat il suffit d'exécuter la requête suivante :

```
SELECT COUNT(*) FROM import WHERE <numColonne> IS NULL;
```

Si le résultat est supérieur à 0, alors la colonne contient des valeurs nulles.

Ainsi on a, pas exclusivement, les colonnes suivantes avec des valeurs nulles :

n118: composante_id_paysage. On compte **13632** valeurs nulles soit **98,291%** des lignes.

n54: Dont effectif des admis en internat. On compte **12886** valeurs nulles soit **92,912%** des lignes.

n111: Concours communs et banque d'épreuves. On compte **13202** valeurs nulles soit **95,191%** des lignes.

n22: Dont effectif des candidats ayant postulé en internat. On compte **12886** valeurs nulles soit **92,912%** des lignes.

n70: Dont effectif des admis issus du même établissement (BTS/CPGE). On compte **7547** soit **54,416%** des lignes.

Ventilation des données

Nous pouvons donc maintenant passer à la ventilation des données. Nous avons tout d'abord décidé de supprimer les données calculables à l'aide d'autres colonnes car elles constituent une grande quantité de données inutiles car recalculables.

Suppression des colonnes calculables

Nous allons prendre l'exemple de la colonne n74 de la table import. Cette colonne est calculable en utilisant les colonnes 51 et 47 avec la requête suivante :

```
SELECT ROUND((n51 / n47) * 100) FROM import  
WHERE n47 <> 0;
```

Toutefois, il faut vérifier si le calcul est bon. Pour cela, on peut exécuter la requête suivante renvoyant le pourcentage d'erreur pour le re-calcul de la colonne 74:

```
SELECT ((COUNT(*)*100.00)/13869.00) FROM import  
WHERE ROUND(n74) <> ROUND((n51 / n47) * 100)  
AND n47 <> 0;
```

Cette requête vérifie pour chaque ligne si le calcul est égal à la valeur de la colonne 74 (seulement si la valeur de la colonne n47 est non nulle pour éviter d'être sujet à une division par 0). Si le calcul ne l'est pas, la fonction d'agrégation **COUNT(*)** est incrémentée de 1. La requête renvoie donc, grâce à cette fonction, le pourcentage d'erreur.

Attention: la requête arrondit les valeurs pour la vérification pour éviter les erreurs uniquement liés au manque de précision du calcul flottant.

Pour cette requête, on obtient un pourcentage d'erreur d'environ 1.50%, ce que nous considérons comme étant acceptable.

Nous pouvons donc maintenant répéter le même processus pour chaque colonne calculable en utilisant, en guise de vérification, une requête de la forme suivante :

```
SELECT ((COUNT(*)*100.00)/13869.00) FROM import  
WHERE ROUND(<colonneOriginale>) <> ROUND(<calcul>)  
AND <conditionsCalcul>;
```

On obtient alors les résultats suivants :

n74: 1.50% d'erreur

n75: 0.50% d'erreur

n76: 0.49% d'erreur

n77: 0.05% d'erreur

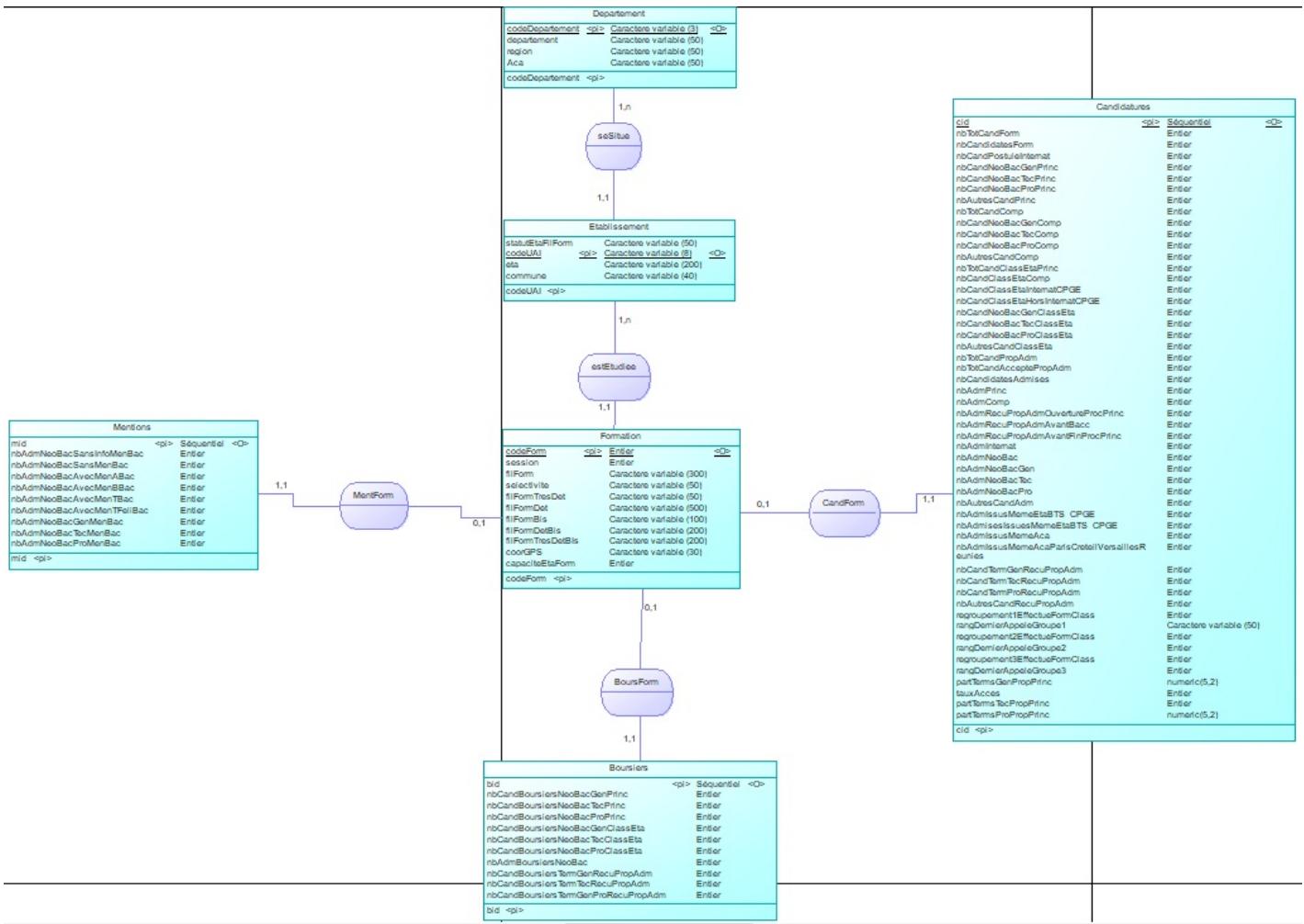
n78: 0.02% d'erreur
n79: 0.02% d'erreur
n80: 0.00% d'erreur
n81: 0.00% d'erreur
n82: 0.06% d'erreur
n83: 0.00% d'erreur
n84: 0.02% d'erreur
n85: 0.00% d'erreur
n86: 0.02% d'erreur
n87: 0.00% d'erreur
n88: 0.00% d'erreur
n89: 0.00% d'erreur
n90: 0.00% d'erreur
n91: 0.00% d'erreur
n92: 0.00% d'erreur
n93: 0.00% d'erreur
n94: 43.46% d'erreur

Comme vous pouvez le constater, pour la plupart des colonnes, nous trouvons un colonene proche voir égal à 0.00% d'erreur. Nous avons donc réussi à économiser une grande quantité de stockage de données ce qui nous permettra également d'alléger notre nouveau système de données.

Création du MCD

Nous avons donc maintenant un nombre de colonnes réduit au nombre de colonnes contenant des valeurs non calculées et receuillies lors de la session Parcoursup de 2023. Nous décidons, pour notre nouveau système de données, de conserver **toutes les données non calculables**. Nous voulons créer un système de données utilisables entièrement utilisables par ParcourSup, il est donc important de ne pas supprimer car chaque donnée à son degré d'importance. Nous allons donc en résumé travailler avec tout les colonnes sauf celles calculables précédemment énoncées. Vous pouvez d'ailleurs retrouver quelles colonnes et calculables ou non dans le fichier **dico.xls**.

Nous avons donc modéliser notre modéliser notre MCD. Nous vous mettons ci-dessous un capture d'écran du MCD réalisé à l'aide du logiciel **PowerAMC** (le MCD étant assez grand et la qualité de la capture d'écran d'assez faible qualité, il est peut-être coompliqué de lire tous les détails, nous nous en excusons) :



Pour cette modélisation, il était tout d'abord important de supprimer toute redondance. Les entités **Departements**, **Etablissements** et **Formations** sont faites pour. Ensuite nous avons divisor chaque formation par thème avec des entités supplémentaires, reliées avec l'entité Formations par des relations bijectives. Ces entités sont **Mentions**, **Boursiers** et **Candidatures** et permettent un répartition cohérente des données.

Les relations, peintes lisibles sur le MCD, sont les suivantes :

Departements (1,n) *seSitué* (1,1) **Etablissements**
Etablissements (1,n) *estEtudiee* (1,1) **Formations**
 Mentions (1,1) *MentForm* (0,1) **Formations**
 Boursiers (1,1) *BoursForm* (0,1) **Formations**
Candidatures (1,1) *CandForm* (0,1) **Formations**

Passage d'un MCD aux tables

Avant de pouvoir créer les tables à l'aide de requêtes SQL, il est important de définir quelles sont ces tables et quel est leur contenu.

Tout d'abord, on note l'absence de lien maillé dans notre MCD. Ainsi, nous pouvons déjà relever le fait que le nombre de tables à créer est le même que le nombre d'entités placées dans notre MCD. Logiquement, les tables seront nommées comme nos entités et elles seront au moins composées des attributs et des identifiants (comme clé primaire) qui définissent les entités correspondantes.

Pour ce qui est des reports de clés primaires :

- La table **Etablissements** admet comme clé étrangère l'attribut **codeDepartement**, clé primaire de la table **Départements**.
 - La table **Formations** admet comme clé étrangère l'attribut **codeUAI**, clé primaire de la table **Etablissements**.
 - Les tables **Candidatures**, **Boursiers** et **Mentions** admettent comme clé étrangère l'attribut **codeForm**, clé primaire

de la table **Formations**.

Ecriture d'un script de création des tables

Maintenant que la structure des tables est définie, nous pouvons passer à la création d'un **script SQL** à exécuter depuis PostGreSQL et qui a pour but de réaliser toutes les tâches nécessaires à la création de la nouvelle structure de données. Ce fichier doit être *idempotent*. Nous allons donc expliquer pour chaque partie les commandes relatives à la réalisation de cette partie ainsi que les commandes permettant de rendre ce fichier utilisables à notre guise. Ces parties sont :

- Téléchargement du fichier source
- Importation des données
- Création des nouvelles tables
- Ventilation des données
- Création d'indexs
- Exportation des données

Note : Nous allons être amenés à exécuter des commandes bash durant le fonctionnement de notre script SQL. Pour pouvoir exécuter une commande bash depuis un script SQL, dans PostGreSql, il faut ajouter avant la commande les caractères : !.

Téléchargement du fichier source

La première étape, avant de pouvoir manipuler les données, est de les télécharger. Toutefois, avant de commencer le téléchargement il faut supprimer le fichier, s'il existe, sous la forme du nom avec lequel il va être téléchargé (csv?lang=fr) et celle du nouveau nom que nous lui attriburons (fr-esr-parcoursup.csv). Cette étape permet d'éviter tout conflits entre fichiers lors des commandes qui suivront si un tel fichier a déjà été téléchargé. On exécute les commandes suivantes :

```
rm -f csv\?lang\=fr*
rm -f fr-esr-parcoursup.csv
```

Une fois ce ménage achevé, on peut passer au téléchargement du fichier avec la commande bash suivante :

```
$(wget <urlFichier>)
```

Il est important de garder la syntaxe \$(<commande>) sinon les commandes suivantes s'exécuteront avant la fin du téléchargement du fichier, ce qui est évidemment pas souhaitable.

Une fois le téléchargement terminé, on peut renommer le fichier et lui donner le nom attribué lors d'un téléchargement manuel avec la commande :

```
mv csv\?lang\=fr fr-esr-parcoursup.csv
```

Importation des données

Maintenant que le fichier est prêt, nous pouvons maintenant importer les données dans une table temporaire nommée import, comme expliqué dans la troisième étape relative à l'importation des données dans PostGreSQL.

Création des nouvelles tables

Avant de créer les tables, afin de rendre le fichier idempotent, il faut supprimer les différentes tables et contraintes associées à celle-ci afin de pouvoir repartir d'une nouvelle base si de telles tables ont déjà étaient créées. On exécute :

```
DROP TABLE IF EXISTS Mentions;
DROP TABLE IF EXISTS Boursiers;
DROP TABLE IF EXISTS Candidatures;
DROP TABLE IF EXISTS Formations;
DROP TABLE IF EXISTS Etablissements;
DROP TABLE IF EXISTS Departements;
```

La suppression des tables se fait dans l'ordre inverse à celui de la création des tables afin d'éviter toute erreur liées à la dépendance des tables entre-elles.

On peut donc maintenant créer les tables. Nous n'allons pas donner, pour une question de lisibilité, l'entiereté des requêtes de création des tables car ces dernières sont trouvables dans le fichier parcoursup.sql. Toutefois, nous en donnerons la structure des requêtes ainsi que la déclaration des contraintes associées aux tables.

```
CREATE TABLE Departements (
    codeDepartement VARCHAR(3),
    departement VARCHAR(50),
    region VARCHAR(50),
    aca VARCHAR(50),
    CONSTRAINT pK_Departements PRIMARY KEY(codeDepartement)
);

CREATE TABLE Etablissements (
    statutEtaFilForm VARCHAR(50),
    codeUAI VARCHAR(8),
    eta VARCHAR(200),
    codeDepartement VARCHAR(3),
    CONSTRAINT pK_Etablissements PRIMARY KEY(codeUAI),
    CONSTRAINT fk_CodeDepartement FOREIGN KEY(codeDepartement) REFERENCES Departements(codeDepartement)
        ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE TABLE Formations (
    codeForm INT,
    codeUAI VARCHAR(8),
    filForm VARCHAR(300),
    selectivite VARCHAR(50),
    ...
    capaciteEtaForm INT,
    CONSTRAINT pK_Formations PRIMARY KEY(codeForm),
    CONSTRAINT fk_CodeUAI FOREIGN KEY(codeUAI) REFERENCES Etablissements(codeUAI)
        ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE TABLE Candidatures (
    cid SERIAL,
    codeForm INT,
    nbTotCandForm INT,
    nbCandidatesForm INT,
    ...
);
```

```

partTermsProPropPrinc NUMERIC(5,2),
CONSTRAINT pK_Candidatures PRIMARY KEY(cid),
CONSTRAINT fk_codeForm FOREIGN KEY(codeForm) REFERENCES Formations(codeForm)
    ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE TABLE Boursiers (
    bid SERIAL,
    codeForm INT,
    nbCandBoursiersNeoBacGenPrinc INT,
    nbCandBoursiersNeoBacTecPrinc INT,
    ...
    nbCandBoursiersTermGenProRecuPropAdm INT,
    CONSTRAINT pK_Boursiers PRIMARY KEY(bid),
    CONSTRAINT fk_codeForm FOREIGN KEY(codeForm) REFERENCES Formations(codeForm)
        ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE TABLE Mentions (
    mid SERIAL,
    codeForm INT,
    nbAdmNeoBacSansInfoMenBac INT,
    nbAdmNeoBacSansMenBac INT,
    ...
    nbAdmNeoBacProMenBac INT,
    CONSTRAINT pK_Mentions PRIMARY KEY(mid),
    CONSTRAINT fk_codeForm FOREIGN KEY(codeForm) REFERENCES Formations(codeForm)
        ON UPDATE CASCADE ON DELETE SET NULL
);

```

Ventilation des données

La dernière étape obligatoire pour la création du système de données est de ventiler les données. Néanmoins, nous allons préalablement faire quelques requêtes pour savoir, en guise de vérification, combien de lignes doivent comporter nos nouvelles tables :

```

-- Departements
SELECT COUNT(DISTINCT n5) FROM import; -> 104

-- Etablissements
SELECT COUNT(DISTINCT n3) FROM import; -> 3965

-- Formations, Candidatures, Boursiers, Candidatures
SELECT COUNT(n110) FROM import; -> 13869

```

On obtient comme résultat : **104, 3965, 13869**. Nous reviendrons sur ces résultats après la ventilation.

Pour ce qui est de la ventilation, chaque ligne correspond à une seule formation et chaque ligne est identifiée par une clé primaire (**n110** dans le table import et **codeForm** dans les nouvelles tables). De plus, les lignes ne sont composées que de colonnes unique. Ainsi, la ventilation peut se faire par des requêtes assez simples. Ces requêtes sont :

```

/* Ventilation de la table Departements */

INSERT INTO Departements (codeDepartement, departement, region, aca)
    SELECT distinct n5, n6, n7, n8 FROM import;

/* Ventilation de la table Etablissements */

INSERT INTO Etablissements (statutetafilform, codeuai, eta, codedepartement)
    SELECT n2, n3, n4, n5 FROM import GROUP BY n2,n3,n4,n5;

/* Ventilation de la table Formations */

INSERT INTO Formations (codeForm, filForm, codeUAI, selectivite, filFormTresDet,
filFormDet, filFormBis, filFormDetBis, filFormTresDetBIs, coorGPS, capaciteEtaForm)
    SELECT n110,n10,n3,n11,n12,n13,n14,n15,n16,n17,n18 FROM import;

/* Ventilation de la table Candidatures */

INSERT INTO Candidatures (
    codeForm, nbTotCandForm, nbCandidatesForm,
    nbCandPostuleInternat, nbCandNeoBacGenPrinc,
    nbCandNeoBacTecPrinc, nbCandNeoBacProPrinc,
    nbAutresCandPrinc, nbTotCandComp, nbCandNeoBacGenComp,
    nbCandNeoBacTecComp, nbCandNeoBacProComp, nbAutresCandComp,
    nbTotCandClassEtaPrinc, nbCandClassEtaComp, nbCandClassEtaInternatCPGE,
    nbCandClassEtaHorsInternatCPGE, nbCandNeoBacGenClassEta,
    nbCandNeoBacTecClassEta, nbCandNeoBacProClassEta, nbAutresCandClassEta,
    nbTotCandPropAdm, nbTotCandAcceptePropAdm, nbCandidatesAdmises,
    nbAdmPrinc, nbAdmComp, nbAdmRecuPropAdmOuvertureProcPrinc,
    nbAdmRecuPropAdmAvantBacc, nbAdmRecuPropAdmAvantFinProcPrinc, nbAdmInternat,
    nbAdmNeoBac, nbAdmNeoBacGen, nbAdmNeoBacTec, nbAdmNeoBacPro, nbAutresCandAdm,
    nbAdmIssusMemeEtaBTS_CPGE, nbAdmisesIssuesMemeEtaBTS_CPGE, nbAdmIssusMemeAca,
    nbAdmIssusMemeAcaParisCreteilVersaillesReunies, nbCandTermGenRecuPropAdm,
    nbCandTermTecRecuPropAdm, nbCandTermProRecuPropAdm, nbAutresCandRecuPropAdm,
    reGROUPement1EffectueFormClass, rangDernierAppeleGROUPe1, reGROUPement2EffectueFormClass,
    rangDernierAppeleGROUPe2, reGROUPement3EffectueFormClass, rangDernierAppeleGROUPe3,
    tauxAcces, partTermsGenPropPrinc, partTermsTecPropPrinc, partTermsProPropPrinc
)
SELECT
    f.codeForm, n19, n20, n22, n23, n25, n27, n29, n30, n31, n32, n33, n34, n35, n36, n37, n38,
    n39, n41, n43, n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n56, n57, n58, n59, n60, n70
    n71, n72, n73, n95, n97, n99, n101, n102, n103, n104, n105, n106, n107, n113, n114, n115, n116
FROM
    import as i
JOIN
    Formations as f ON i.n110 = f.codeForm;

/* Ventilation de la table Boursiers */

INSERT INTO Boursiers (codeForm, nbCandBoursiersNeoBacGenPrinc, nbCandBoursiersNeoBacTecPrinc,
nbCandBoursiersNeoBacProPrinc, nbCandBoursiersNeoBacGenClassEta, nbCandBoursiersNeoBacTecClassEta,
nbCandBoursiersNeoBacProClassEta, nbAdmBoursiersNeoBac, nbCandBoursiersTermGenRecuPropAdm,
nbCandBoursiersTermTecRecuPropAdm, nbCandBoursiersTermGenProRecuPropAdm)
    SELECT f.codeForm, n24, n26, n28, n40, n42, n44, n55, n96, n98, n100
    FROM import as i

```

```

JOIN Formations as f ON i.n110 = f.codeForm;

/* Ventilation de la table Mentions */

INSERT INTO Mentions (codeForm, nbAdmNeoBacSansInfoMenBac, nbAdmNeoBacSansMenBac, nbAdmNeoBacAvecMenABac,
nbAdmNeoBacAvecMenBBac, nbAdmNeoBacAvecMenTBac, nbAdmNeoBacAvecMenTFeliBac, nbAdmNeoBacGenMenBac,
nbAdmNeoBacTecMenBac, nbAdmNeoBacProMenBac)
SELECT f.codeForm, n61, n62, n63, n64, n65, n66, n67, n68, n69
FROM import as i
JOIN Formations as f ON i.n110 = f.codeForm;

```

Nos données sont donc maintenant ventilés. Nous pouvons vérifier que la ventilation s'est bien passées avec les requêtes suivantes :

```

SELECT COUNT(*) FROM Departements;
SELECT COUNT(*) FROM Etablissements;
SELECT COUNT(*) FROM Formations;
SELECT COUNT(*) FROM Candidatures;
SELECT COUNT(*) FROM Boursiers;
SELECT COUNT(*) FROM Mentions;

```

Après exécution on retrouve bien les valeurs attendues (104, 3965, 4x 13869). Notre ventilation est donc réussie.

Création d'indexs

Maintenant que nos données sont importées, il est intéressant (pas obligatoire), de mettre en place quelques indexs qui vont permettre, par le biais d'arbres binaires des recherche, d'optimiser les recherches. En effet, ces indexs permettent d'optimiser les recherches mais pénalisent les mises à jour des données. Toutefois, une fois les données importées, il est très peu probable que cette base de données soit soumise à beaucoup de mises à jour (si tel était le cas il faudrait procéder de temps en temps à une ré-indexation). On exécute :

```

-- Suppression des indexs s'ils existents

DROP INDEX IF EXISTS idxCandidaturesPk;
DROP INDEX IF EXISTS idxCandidaturesFk;
DROP INDEX IF EXISTS idxFormationsPk;
DROP INDEX IF EXISTS idxDepartementsPk;
DROP INDEX IF EXISTS idxEtablissementsPk;
DROP INDEX IF EXISTS idxEtablissementsFk;

-- Création des indexs

CREATE INDEX idxCandidaturesPk ON Candidatures(cid);
CREATE INDEX idxCandidaturesFk ON Candidatures(codeForm);
CREATE INDEX idxFormationsPk ON Formations(codeForm);
CREATE INDEX idxDepartementsPk ON Departements(codeDepartement);
CREATE INDEX idxEtablissementsPk ON Etablissements(codeUAI);
CREATE INDEX idxEtablissementsFk ON Etablissements(codeDepartement);

```

Exportation des données

La dernière étape dans le rédaction du script est d'exporter les données dans des fichiers CSV. On souhaite ranger ces fichiers dans un répertoire **results**. On exécute donc ces commandes pour que soit le répertoire re-créer à chaque lancement du script :

```
\! rm -rdf results
\! mkdir results
```

Enfin, on exporte les données avec les requêtes :

```
\COPY Departements to 'results/departements.csv' WITH (DELIMITER ',', NULL '');
\COPY Etablissements to 'results/etablissements.csv' WITH (DELIMITER ',', NULL '');
\COPY Formations to 'results/formations.csv' WITH (DELIMITER ',', NULL '');
\COPY Candidatures to 'results/candidatures.csv' WITH (DELIMITER ',', NULL '');
\COPY Boursiers to 'results/boursiers.csv' WITH (DELIMITER ',', NULL '');
\COPY Mentions to 'results/mentions.csv' WITH (DELIMITER ',', NULL');
```

Notre script est donc maintenant totalement terminé.

Comparaison des tailles

Le travail de re-structuration est donc maintenant terminé. Nous pouvons de ce fait totalement manipuler ces données à l'aides de requêtes. Mais, tout d'abord, il est intéressant de comparer les différentes tailles, avant et après la ventilation des données. Nous allons comparer à travers des questions.

Quelle taille en octet fait le fichier récupéré ?

Pour répondre à cette question, on utilise la commande bash suivante :

```
ls -l fr-esr-parcoursup.csv
```

La requête nous renvoie 12423586 ce qui correspond à la taille en octets du fichier initial (12,42 Mo).

Attention : La taille peut varier. En effet cette commande a été exécutée sur le fichier téléchargé manuellement. Toutefois, lorsqu'on télécharge le fichier avec un commande, comme dans le script, la ligne d'en-tête diffère ce qui faire légèrement varier le résultat.

Quelle taille en octet fait la table import ?

Pour cette question, on utilise une fonction propre à PostGreSQL qui permet de mesurer la taille d'une relation, en prenant en compte toute la structure de la base de données :

```
pg_total_relation_size('import');
```

On obtient comme résultat 14368768 octets soit une taille supérieure à celle du fichier initial. Toutefois, cela s'explique par la présence des données dans un autre système de données mais également par le fait que toutes les données ne sont pas au format text mais on chacune leur type adapté ce qui fait que les informations ne sont pas toutes encodées de la même manière.

Quelle taille en octet fait la somme des tables créées ?

On exécute, avec le même principe que la question précédente, la requête :

```
SELECT pg_total_relation_size('Departements') +
       pg_total_relation_size('Etablissements') +
       pg_total_relation_size('Formations') +
       pg_total_relation_size('Candidatures') +
       pg_total_relation_size('Boursiers') +
       pg_total_relation_size('Mentions');
```

On obtient comme résultat 13746176 octets ce qui est toujours supérieur à la taille du fichier initial. Toutefois, nous allons montré avec la question suivante que cela est du à la structuration sous forme de base de données et à la décalration de différents types.

Quelle taille en octet fait la somme des tables créées ?

Pour mesurer la taille des fichiers exportés, il faut faire la somme des résultats obtenus par l'exécution des commandes suivantes :

```
stat -c "%s" "results/departements.csv"
stat -c "%s" "results/etablissements.csv"
stat -c "%s" "results/formations.csv"
stat -c "%s" "results/candidatures.csv"
stat -c "%s" "results/boursiers.csv"
stat -c "%s" "results/mentions.csv"
```

On fait donc la somme **4075 + 227681 + 4043437 + 2698493 + 495843 + 419534 = 7889063**. Le résultat obtenu, cette fois totalement comparable avec la taille du fichier précédent en terme d'encodage, est bien plus faible que la taille initiale. Le re-modélisation des données a quasiment permis de diviser leur taille par deux.

On remarquera néanmoins que la suppression de 20 colonnes pour 13869 lignes n'as pas tant fait baisser la taille des données dans PostGreSQL tandis que pour le format CSV la taille a quasiment était divisée par deux. On peut conjecturer le fait, qu'avec la manière dont sont encodées les données avec un tel SGBD, que ce qui prend le plus de taille est la structure et non pas la quantité de données intégrées à ces bases et que ces SGBD sont fait plutôt fait pour de l'optimisation de base de données plutôt conséquentes en termes de quantité de données.

Requêtage

Ecrire une requête qui, à partir de import affiche le contenu de la colonne n56 et le re-calcul de celle-ci à partir d'autres colonnes de import

```
SELECT n56, (n57+n58+n59) FROM import;
```

Quelle requête vous permet de savoir que ce re-calculation est parfaitement exact ?

```
SELECT COUNT(*) FROM import WHERE n56<>(n57+n58+n59);
```

Le re-calculation est parfaitement exact si cette requête retourne 0. Après exécution, on retrouve la valeur 0 ce qui est gage d'un re-calculation parfait

Ecrire une requête qui, à partir de import affiche le contenu de la colonne n74 et le re-calculation de celle-ci à partir d'autres colonnes de import

```
SELECT n74, (n51 / n47)*100 FROM import WHERE n47<>0;
```

Quelle requête vous permet de savoir que ce re-calculation est parfaitement exact ?

```
SELECT COUNT(*) FROM import WHERE n74 <> round((n51 / n47) * 100) AND n47 <> 0;
```

On utilise le même principe que précédemment à la seule différence que les valeurs sont arrondies car les valeurs initiales sont flottantes mais ont tout le temps leurs parties décimales nulles

Ecrire une requête qui, à partir de import affiche le contenu de la colonne n76 et le re-calculation de celle-ci à partir d'autres colonnes de import (2 cols). A partir de combien de décimales ces données sont exactes ?

```
SELECT n76, (n53/n47)*100 FROM import WHERE n47<>0;
```

Fournir la même requête sur vos tables ventilées

```
SELECT round((round(nbAdmRecuPropAdmAvantFinProcPrinc)/round(nbTotCandAcceptePropAdm))*100) AS results
FROM Candidatures
WHERE nbTotCandAcceptePropAdm<>0;
```

Ecrire une requête qui, à partir de import affiche la n81 et la manière de la recalculer. A partir de combien de décimales ces données sont exactes ?

```
SELECT n81, round((round(n55)/round(n56))*100) AS results FROM import
WHERE n56<>0;
```

Fournir la même requête sur vos tables ventilées

```

SELECT round((round(b.nbAdmBoursiersNeoBac)/round(c.nbAdmNeoBac))*100) AS results
FROM Candidatures AS c JOIN Boursiers AS b ON c.codeForm=b.codeForm
WHERE c.nbAdmNeoBac<>0;

```

Annexes

Génération du fichier dico.xls

```

cd scripts
chmod a+x createDico.sh
./createDico.sh ../data/fr-esr-parcoursup.csv

```

```

#!/bin/sh

# Définition des variables d'usage
filename=$1
rawColumnsText=$(cat $filename | head -n 1)

# Génération du fichier SQL de création de la table import
cd ../javaTools
javac -d bin src/GenerateDico.java
java -cp bin GenerateDico $rawColumnsText > ../dicoVierge.xls
echo "Fichier généré. Fin de script."

```

```

public class GenerateDico {

    public static void main(String[] args) {
        String[] tmp = GenerateDico.extractColumnName(GenerateDico.assembleString(args));
        System.out.println(GenerateDico.generateText(tmp));
    }

    public static String assembleString(String[] tab) {
        String tmp = "";
        for (int idxTab = 0; idxTab < tab.length; idxTab++) {
            tmp += ' ' + tab[idxTab];
        }
        return tmp.substring(1);
    }

    public static String[] extractColumnName(String input) {
        return input.split(";");
    }

    public static String generateText(String[] columnName) {
        String chaine = "Nom de colonne dans import;Nom dans le fichier initial\n";
        for (int idxColumn = 0; idxColumn < columnName.length; idxColumn++) {

```

```

        chaine += "\n" + (idxColumn + 1) + ';' + columnsName[idxColumn] + '\n';
    }
    return chaine.substring(0, chaine.length() - 1);
}

}

```

Génération du script de création de la table import

```

cd scripts
chmod a+x createImport.sh
./createImport.sh ../data/fr-esr-parcoursup.csv

```

```

#!/bin/sh

# Définition des variables d'usage
filename=$1
nbColumns=$(cat $filename | head -n 1 | tr ';' '\n' | wc -l)

# Affichage du nombre de colonnes
echo "Il y a $nbColumns colonnes pour la table d'import."

# Génération du fichier SQL de création de la table import
cd ../javaTools
javac -d bin src/GenerateCreateTable.java
java -cp bin GenerateCreateTable $nbColumns > ../sql/createImport.sql
echo "Commande générée. Fin de script."

```

```

public class GenerateCreateTable {

    public static void main(String[] args) {
        if (args.length > 0) {
            int nbColumns = Integer.valueOf(args[0]);
            System.out.println(GenerateCreateTable.generateText(nbColumns));
        }
    }

    public static String generateText(int nbColumns) {
        String chaine = "CREATE temp TABLE import (\n";
        for (int idxCol = 1; idxCol <= nbColumns; idxCol++) {
            if(idxCol==nbColumns){
                chaine += "\tn" + idxCol + " text\n";
            } else {
                chaine += "\tn" + idxCol + " text,\n";
            }
        }
        return chaine + ")";
    }
}

```

