LAMBERT Philippe
Simplon Nantes
DevData 2020-2021

SIMPLON
GRAND OUEST

PROJET CHEF D'ŒUVRE

"PROJET SPQR"

RAPPORT

TABLE DES MATIÈRES :

I. Contexte du projet :

A. Notre offre :

Dans le cadre de notre formation, nous sommes amenés à réaliser un projet de création et de gestion de base de données pour un client. Le travail est individuel. Dans ce cadre, je ne dispose pas de grands moyens, seulement d'un ordinateur et de logiciels gratuits. Je suis formé à l'utilisation de Git, d'un environnement virtuel, de Python, de différents systèmes de gestion de base de données.

B. Le client :

Notre client, Historiaweb, est un site Internet éducatif sur le thème de l'Histoire, à destination des jeunes, étudiants ou amateurs d'Histoire. Il aimerait développer un service permettant de trouver facilement des informations sur des personnalités historiques. Dans un premier temps, ce projet sera concentré sur l'antiquité romaine, et ne sera adapté pour d'autres périodes que si c'est un succès.

Le client aimerait au minimum des informations biographiques sur les personnes, des données sur les œuvres que certains ont pu produire, et leur domaine d'activité. Il voudrait aussi que l'on explore ces données pour voir si l'on peut y découvrir des informations nouvelles capables d'éveiller la curiosité des utilisateurs.

Il dispose déjà d'équipes consacrées au développement web, capables de mettre au point une interface pour la recherche et l'affichage des données. Il est intéressé cependant à ce qu'on conçoive pour lui et que l'on gère la base de données à partir de laquelle il pourra proposer cette interface à ses utilisateurs.

II. Préparation et organisation du projet :

A. Identification des sources de données

1. État de l'art

Il n'y a pas vraiment d'équivalent à ce que veut notre client. Toutes les informations sur les personnages historiques de l'Antiquité romaine sont librement accessibles, mais sous forme de livres ou d'articles, éventuellement de listes, pas d'une base de données.

Il existe en revanche des bases de données universitaires sur le thème de l'Antiquité romaine, mais à destination des chercheurs, pas du grand public. On compte par exemple des interfaces élaborées telles que romanopendata.eu, consacrée aux traces d'épigraphie latine sur des murs, amphores etc. retrouvés par les archéologues. Non seulement elle est trop complexe pour être utilisée par des profanes, mais elle l'est même trop pour pouvoir alimenter notre propre projet. Il faut s'en tenir à des informations simples, du moins pour commencer.

2. Source choisie

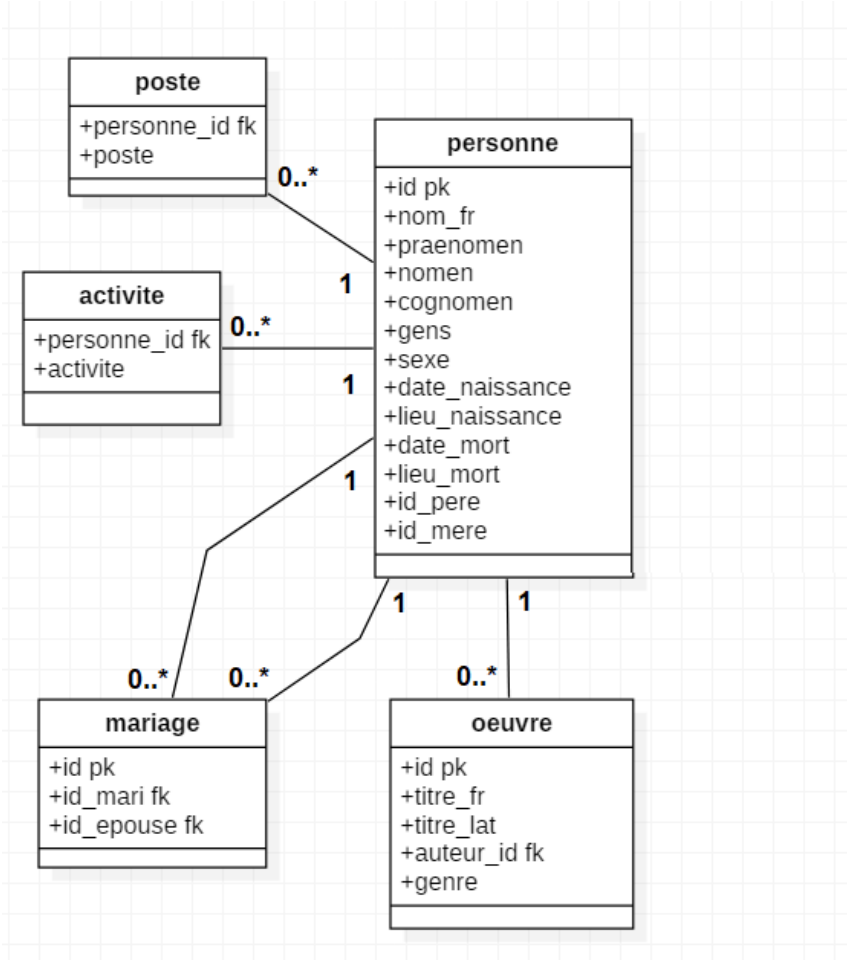
A la recherche d'une source principale de données, mon choix s'est porté sur Wikidata. Projet frère de Wikipédia, chapeauté par la même fondation et lui servant à stocker et structurer les informations, Wikidata possède plus de 93 millions de pages d'items en tous genres en relation les uns avec les autres.

L'information disponible correspond parfaitement au niveau de complexité voulu par le client : des données biographiques, des liens avec des œuvres etc. Mais en même temps Wikidata ne gênera pas le développement du service voulu par le client, car les recherches y sont très fastidieuses, à l'aide d'un langage peu intuitif et de codes alphanumériques. Le site est plus indiqué comme source brute pour une base plus facilement navigable. Le côté très brut des données fournies donne aussi un challenge intéressant de nettoyage.

3. Questions légales et réglementaires

Ce projet et la source choisie ne posent a priori pas de risque légal ou réglementaire, que ce soit dans le cadre de la RGPD, car tous les individus recensés sont morts il y a des siècles, ou dans le cadre de la propriété intellectuelle, car Wikidata est très explicitement sous licence libre et autorise à réutiliser ses informations, y compris de façon commerciale, du moment qu'il est fait mention de la source.

B. Modélisation des données



J'ai décidé de fonder ma base autour d'une table principale 'personne', regroupant les informations principales. Ensuite, le respect des formes normales pousse à créer des tables annexes pour tout ce qui pourrait autrement créer des doublons et des redondances : par exemple un même individu peut avoir exercé plusieurs activités, plusieurs postes dans l'administration ou l'armée, été marié à plusieurs personnes et écrit plusieurs œuvres.

C. Organisation du projet

1. Procédure incrémentale

Suite à une formation sur les méthodes agiles, je me suis dit qu'il serait sans doute préférable d'organiser mon projet par incréments, de livrer de manière assez régulière des versions d'abord basiques puis progressivement plus élaborées de mon travail. Cela a plusieurs avantages sur l'alternative principale, qui consisterait à planifier d'avance la réalisation d'un produit fini : déjà il n'y aurait pas de risque de prendre du retard et de devoir en conséquence bousculer son planning, stresser et se hâter de terminer quelque chose avant le rendu. J'avancerais à rythme plus ou moins constant et quand bien même je n'aurais pas fini tout ce que je comptais faire, j'aurais toujours une version fonctionnelle à présenter.

Mais aussi, cette approche permet de montrer plus fréquemment au client où l'on en est, d'obtenir sa réaction et de pivoter si nécessaire, en changeant les attentes et réorientant le planning vers de nouvelles priorités.

2. Backlog produit

Le client aimerait, dans l'ordre d'importance :

- une liste des personnages historiques de l'antiquité romaine
- leur sexe, dates de naissance et de mort
- une base de données fonctionnelle
- leur nom romain complet
- leur activité et leurs postes
- leurs oeuvres écrites
- leur gens
- leurs relations familiales
- leur localisation

3. Planning prévisionnel

- Fin mars : tests d'extraction de données avec SPARQL, réflexion sur le projet et l'organisation de la base.
- Début avril : collecte et nettoyage des données
- Mi-avril : création d'une version basique de la base de données
- Fin avril : base bien formatée et exploitable
- Début mai : peaufinage du script, ajout de fonctionnalités, analyse des données

III. Mise en oeuvre

A. Extraction

J'ai d'abord cherché à me familiariser avec Wikidata et son système de requête intégré, le Wikidata Query Service. Celui-ci permet de rechercher et afficher des données du site à l'aide du langage SPARQL. Or, ce langage appartenant au Web sémantique n'est pas au programme de la formation, et bien qu'il soit proche dans son principe de SQL, il s'en différencie largement et est beaucoup moins intuitif. Sur Wikidata, il est même complètement inintelligible de premier abord car chaque propriété et chaque item sont exprimés par leur code identifiant unique, et pas par un nom, même anglais. Heureusement, l'interface de Wikidata Query Service aide beaucoup à se repérer[1], grâce notamment à une longue liste d'exemples de requêtes[2], et en affichant la signification de chaque identifiant lors du survol par la souris. J'ai aussi bien profité d'une vidéo tutorielle d'Ewan McAndrew de l'université d'Edinburgh.[3]

J'ai identifié toute une série de propriétés Wikidata qui pourraient être d'intérêt, dans un incrément futur si le client le désire, mais beaucoup ne sont renseignées (voire pertinentes) que pour très peu de personnes, comme "tué par", donc les ajouter à la base n'est pas forcément une bonne idée. A la place je me suis concentré sur les informations biographiques de base vues plus haut. La requête se présente comme ceci :

[1] A gentle introduction to the Wikidata Query Service
[2] Exemples de requêtes
[3] How to use Wikidata Query Service

Les résultats de la requête s'affichent sur l'interface, et un bouton à droite permet de les télécharger directement, au format JSON, TSV, CSV ou HTML. C'est avec le format CSV que je suis le plus familier, et il me semble adapté à ce que je compte en faire. J'en tire donc un fichier 'personne.csv'. Je procède de la même manière pour les données que je compte insérer dans chacune des tables.

Pour la table 'activite' :

```
SELECT ?item ?occupationLabel
WHERE
{
  ?item wdt:P27 wd:Q1747689 .
  OPTIONAL {?item wdt:P106 ?occupation.}

  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Pour la table 'poste' :

```
SELECT ?item ?positionLabel
WHERE
{
  ?item wdt:P27 wd:Q1747689 .
  OPTIONAL {?item wdt:P39 ?position.}

  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Pour la table 'oeuvre' :

```
SELECT ?oeuvreLabel ?auteurLabel ?titre ?genreLabel
WHERE
{
  ?auteur wdt:P31 wd:Q5.
  ?auteur wdt:P27 wd:Q1747689.
  ?auteur wdt:P106 wd:Q36180.
  ?auteur wdt:P800 ?oeuvre.
  OPTIONAL {?oeuvre wdt:P1476 ?titre}
  OPTIONAL {?oeuvre wdt:P136 ?genre}


  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Pour la table 'mariage' :

```
SELECT ?item ?epouse
WHERE
{
  ?item wdt:P27 wd:Q1747689 .
  ?item wdt:P21 wd:Q6581097 .
  ?item wdt:P26 ?epouse.

  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Voyons maintenant à quoi ressemblent les données téléchargées. Voici par exemple quelques aperçus du fichier 'personne.csv' :

	item	itemLabel	praenomenLabel	nomenLabel	cognomenLabel	gensLabel
1	http://www.wikidata.org/en…	Marcus Vale…	Marcus	Valerius		Valerii
2	http://www.wikidata.org/en…	Saint Marcia…				
3	http://www.wikidata.org/en…	Lucius Cani…	Lucius	Caninius		Caninii
4	http://www.wikidata.org/en…	Severinus		Flavius		Flavii
5	http://www.wikidata.org/en…	Gaius Vibius…	Caius	Vibius	Marsus	Vibii
6	http://www.wikidata.org/en…	Caius Aemili…	Caius	Aemilius		Aemilii
7	http://www.wikidata.org/en…	Spurius Furi…	Spurius	Furius	Medullinus	Furii
8	http://www.wikidata.org/en…	Spurius Furi…	Spurius	Furius	Fusus	Furii
9	http://www.wikidata.org/en…	Quintus Hat…	Quintus	Haterius		Haterii
10	http://www.wikidata.org/en…	Publius Corn…	Publius	Cornelius		Cornelii
11	http://www.wikidata.org/en…	Sosthénès				

sexeLabel	date_naissance	lieu_naissance	date_mort	lieu_mort	pere	mere
masculin	0200-01-01T00:00:00Z	http://www.wi…	t2303561388			
masculin	0050-01-01T00:00:00Z	http://www.wi…	0068-01-01T00:00:00Z	http://w…		
masculin	-0100-01-01T00:00:00Z		-0060-01-01T00:00:00Z		http://www…	
masculin	0450-01-01T00:00:00Z					
masculin	-0100-01-01T00:00:00Z		-0060-01-01T00:00:00Z			
masculin	-0450-01-01T00:00:00Z	http://www.wi…	-0450-01-01T00:00:00Z	t21664…	t21664218…	t2166421799
masculin	-0500-01-01T00:00:00Z	http://www.wi…	-0500-01-01T00:00:00Z	http://w…	t2119068970	t2119069002
masculin	-0500-01-01T00:00:00Z	http://www.wi…	-0500-01-01T00:00:00Z	http://w…	t2119068970	t2119069002
masculin	-0060-01-01T00:00:00Z		0026-01-01T00:00:00Z			
masculin	-0350-01-01T00:00:00Z	http://www.wi…	-0350-01-01T00:00:00Z	t21664…	t21664146…	t2166414668
masculin			0100-01-01T00:00:00Z			
féminin	0301-01-01T00:00:00Z		0365-01-01T00:00:00Z			
masculin	-0600-01-01T00:00:00Z	t2222716854	-0508-01-01T00:00:00Z	http://w…	http://www…	http://www.wikidata.or…
masculin	-0050-01-01T00:00:00Z		-0020-01-01T00:00:00Z		http://www…	

Comme on peut le voir, les données sont encore très brutes, et inexploitables en l'état. Il va falloir les nettoyer sérieusement. Pour commencer,

les intitulés des colonnes, avec notamment les "Label" imposés par la syntaxe SPARQL, ne correspondent pas à ce que l'on veut pour nos tables. Ensuite, au lieu des identifiants uniques que l'on souhaitait et qui s'affichaient sur l'interface de Wikidata Query Service, on obtient des adresses de liens vers la page Wikidata de l'item. Les dates ont également un format inutilement précis et compliqué. Enfin, on remarque çà et là des valeurs étranges, un t minuscule suivi d'un code numérique, toujours différent. Cela ressemble aux codes utilisés pour les différents objets sur Wikidata mais ceux-là ne renvoient à rien. Au contraire ils s'affichent justement quand telle propriété est invoquée sur la page d'un objet, mais n'est reliée à aucune valeur (par exemple "Père : [vide]").

Ce qu'on ne voit pas sur les captures d'écran précédentes, mais qui pose aussi problème, est qu'il y a un certain nombre de doublons. En effet, même si on a évacué la plupart des risques de doublons lors de la conception des différentes tables, ce n'est pas suffisant : sur Wikidata, certains individus peuvent avoir plusieurs dates de naissance ou de mort possibles, et c'est d'autant plus fréquent pour des personnages historiques de l'Antiquité, à cause de l'incertitude des historiens et des contradictions entre les sources. Ainsi, une personne à deux dates de naissance possible sera enregistrée deux fois.

B. Version basique

1. Premier nettoyage des données

Pour nettoyer facilement les données de ces fichiers CSV, je décide de procéder via Python, sur un Notebook Jupyter, un outil très pratique pour faire des essais puisqu'on peut activer chaque cellule l'une après l'autre et recevoir si besoin des messages d'erreur. Je décide d'utiliser des dataframes pour manipuler mes données. Pour cela j'installe et importe la librairie pandas qui a cette fonctionnalité.

Je définis un dataframe lisant les données d'un fichier CSV avec la méthode pd.read_table. La lecture se fait sans problème. Je peux procéder aux premières modifications :

- renommer les colonnes, avec pandas.DataFrame.rename[4]
- supprimer l'adresse de lien Wikidata et le Q au début des codes d'items, pour ne garder que l'identifiant numérique unique. En fait, plus concrètement, je remplace la chaîne de caractères 'http://www.wikidata.org/entity/Q' par une chaîne vide, avec pandas.Series.str.replace[5] (trouvé via StackOverFlow).[6]

[4] Documentation Pandas : pandas.DataFrame.rename
[5] Documentation Pandas : pandas.Series.str.replace
[6] StackOverFlow : Pandas: replace substring in string

- supprimer les doublons, avec pandas.DataFrame.drop_duplicates[7] (trouvé via GeeksforGeeks[8]). La question se pose de quelle entrée conserver, mais comme il n'y a pas de bonne réponse, je choisis de garder la première (paramètre 'keep="first"').

A ce stade, quoiqu'il y ait un 'regex=True' dans mon code précédent, je ne sais pas encore utiliser d'expression régulière, et je sais qu'il faudra que je m'en serve pour éliminer les valeurs du type 't151626261' qui polluent mes données. Mais ce sera lors d'un futur incrément.

Je remarque aussi une subtilité supplémentaire : les cases qui étaient simplement vides dans le fichier deviennent 'NaN' dans le dataframe, ce qui me posera un problème de format plus tard.

2. Création d'une base Postgres et import

Pour l'heure, pour justement éviter ces problèmes de format et fournir rapidement une version basique de ma base de données, je décide de traiter toutes mes données comme des chaînes de caractères. Je décide aussi de n'avoir aucune clé primaire ou étrangère, pas de contrainte. Cela n'empêche pas le bon déroulement d'une requête SQL ou d'une visualisation avec matplotlib, c'est-à-dire une démonstration au client.

Pour travailler avec PostgreSQL j'ai d'abord installé PostgreSQL version 13.1 sur mon terminal, et psycopg2-binary version 2.8.6 dans mon environnement virtuel. J'ai défini un mot de passe secret dans mes variables systèmes, pour éviter d'avoir à l'écrire dans le code. Pour l'utiliser j'importe le module os, et définis mot_passe = os.environ.get('pg_psw').

Je dois d'abord me connecter à une base, par défaut 'postgres', avant de pouvoir en créer une nouvelle, via une requête SQL '"CREATE"'. C'est ce que je fais :

```
conn = psycopg2.connect(
   database="postgres", user='postgres', password=mot_passe, host='localhost', port= '5432'
)
conn.autocommit = True

cursor = conn.cursor()

sql = '''CREATE database ProjetSPQR''';

cursor.execute(sql)

conn.close()
```

[7] Documentation Pandas : pandas.DataFrame.drop_duplicates
[8] GeeksforGeeks : Python | Pandas dataframe.drop_duplicates()

Je me connecte à cette nouvelle base et crée chaque table de la même manière, via une requête SQL exécutée par un curseur connecté à la base de données. Pour chaque table je me contente de lister les colonnes. Ensuite, je peux procéder à l'insertion des données des dataframes, là encore via une requête SQL, par exemple :

```
sql_insert_oeuvre = """INSERT INTO oeuvre (
    id,
    titre_fr,
    titre_lat,
    auteur_id,
    genre) VALUES (%s, %s, %s, %s, %s)"""
```

Mais au curseur on joint pandas.DataFrame.iterrows pour parcourir ligne à ligne le dataframe[9] :

```
for index, row in df_oeuvre.iterrows():
    cursor.execute(sql_insert_oeuvre, tuple(row))
```

L'insertion fonctionne, ainsi que des requêtes test d'exploration des données, en présentation devant le client. Cependant il est nécessaire de formater la base de données de façon plus rigoureuse.

C. Incrément : formatage

Je commence à me familiariser avec les Regex ou expressions régulières, pour me permettre de détecter et supprimer les codes d'erreur de type t minuscule suivi d'une suite de chiffres. Je m'exerce avec regex101.com, et finis par trouver que '^t[0-9]+' correspond à ce que je recherche. J'utilise à nouveau pandas.DataFrame.replace() pour remplacer ces chaînes de caractère, cette fois-ci en les remplaçant par np.nan, c'est-à-dire le 'NaN' qui correspond aux valeurs manquantes dans le dataframe. Cela me semblait alors naturel d'harmoniser de cette façon. Je supprime également les 'T00:00:00Z' inutiles à la fin de toutes les dates.

Je tâche ensuite de formater convenablement la base de données. Je reprends mes requêtes SQL de création de tables et les modifie comme ceci, selon ce que j'avais déjà prévu :

[9] Documentation Pandas : pandas.DataFrame.iterrows

```
sql_creer_table_personne = """
    CREATE TABLE IF NOT EXISTS personne (
    id integer,
    nom_fr text,
    praenomen text,
    nomen text,
    cognomen text,
    gens text,
    sexe text,
    date_naissance date,
    lieu_naissance integer,
    date_mort date,
    lieu_mort integer,
    id_pere integer,
    id_mere integer,
    PRIMARY KEY(id)
    );
"""
```

```
sql_creer_table_activite = """
    CREATE TABLE IF NOT EXISTS activite (
    personne_id integer,
    activite text,
    CONSTRAINT fk_personne
        FOREIGN KEY(personne_id)
            REFERENCES personne(id)
    );
"""
```

```
sql_creer_table_poste = """
    CREATE TABLE IF NOT EXISTS poste (
    personne_id integer,
    poste text,
    CONSTRAINT fk_personne
        FOREIGN KEY(personne_id)
            REFERENCES personne(id)
    );
"""
```

```
sql_creer_table_mariage = """
    CREATE TABLE IF NOT EXISTS mariage (
    id serial,
    id_mari integer,
    id_epouse integer,
    PRIMARY KEY(id),
    CONSTRAINT fk_personne
        FOREIGN KEY(id_mari)
            REFERENCES personne(id),
        FOREIGN KEY(id_epouse)
            REFERENCES personne(id)
    );
"""
```

```
sql_creer_table_oeuvre = """
    CREATE TABLE IF NOT EXISTS oeuvre (
    id integer,
    titre_fr text,
    titre_lat text,
    auteur_id integer,
    genre text,
    PRIMARY KEY(id),
    CONSTRAINT fk_personne
        FOREIGN KEY(auteur_id)
            REFERENCES personne(id)
);
"""
```

Les requêtes sont bien formulées, mais l'insertion ne fonctionne pas, pour de multiples raisons - les données ne sont pas encore assez bien nettoyées :

a) Quelques codes Wikidata en Q suivi de chiffres se trouvent à la place de véritables valeurs, y compris dans les colonnes de type date ou integer. Cela se résout facilement en utilisant la même Regex que pour les codes en t minuscule.

b) On voulait doter la table 'mariage' d'une colonne supplémentaire avec un identifiant unique par mariage ; selon le formateur elle devrait se remplir elle-même sur PostgreSQL lors de l'insertion même sans avoir à la créer dans le dataframe, car elle est de type serial, Cependant il semble que ça ne soit pas le cas. Le problème est réglé en la rajoutant dans le dataframe, avec pandas.DataFrame.insert[10] (trouvé via StackOverFlow[11]), et en remplissant cette colonne par les nombres de l'intervalle entre 0 et la longueur du dataframe :

```
df_mariage.insert(0, 'id', range(0, len(df_mariage)))
```

c) Les valeurs numériques sont considérées comme en-dehors des limites acceptées. Comme j'utilise des identifiants uniques parfois assez grands, j'essaie d'abord de voir s'ils dépassent la limite des 'integers' et si je dois plutôt utiliser le type 'bigint', fait pour les grands nombres. Mais ce n'est pas ça. En réalité, le problème vient du 'NaN' que le dataframe pandas a imposé d'office à toutes les valeurs manquantes : il appartient au type 'float', c'est-à-dire aux nombres décimaux. Heureusement, il est possible, toujours avec pandas.DataFrame.replace(), de remplacer les NaN (np.nan) par des None, qui eux sont acceptés dans tout type de colonne.

[10] Documentation Pandas : pandas.DataFrame.insert
[11] StackOverFlow : How to Add Incremental Numbers to a New Column Using Pandas

d) Les tables sont reliées entre elles par des clés étrangères. Dans le cas de la table 'mariage', ce lien est double : et le mari et l'épouse renvoient à un identifiant de la table 'personne'. Tout identifiant présent dans l'une ou l'autre colonne doit aussi se retrouver dans la colonne 'id' de la table 'personne' sous peine d'erreur. Or, les données de la table 'personne' correspondent aux personnalités qui, sur Wikidata, sont notées comme appartenant à la Rome antique, alors que la colonne 'épouse' de la table 'mariage' correspond aux femmes qui ont été mariées à une personne notée comme appartenant à la Rome antique - rien ne garantit qu'elles soient elles-mêmes notées comme y appartenant, donc rien ne garantit qu'on retrouve leur identifiant dans la table 'personne'. Et effectivement, beaucoup n'y sont pas.

Que faire ? C'est le signe que notre table centrale manque de données et qu'il faudrait l'enrichir. On rajoute ça à notre "backlog" pour un incrément futur. En attendant, la priorité est de mettre en place les clés étrangères. Pour cela, je crée une liste de tous les id répertoriés dans le dataframe df_personne, avec pandas.Series.tolist (une colonne d'un dataframe est une série)

```
personne_id_list = df_personne['id'].tolist()
```

puis je supprime les entrées pour lesquelles l'identifiant de l'épouse ne se trouve pas dans la liste, avec pandas.DataFrame.loc[12] pour localiser une ligne et pandas.DataFrame.drop[13] pour la supprimer :

```
for e in df_mariage['id_epouse']:
    if e not in personne_id_list:
        df_mariage.drop(df_mariage.loc[df_mariage['id_epouse']==e].index, inplace=True)
```

e) Les dates, ayant pourtant perdu toute trace d'heure et de fuseau horaire, et cherchant à être insérées dans des colonnes de format 'date' (donc sans heure), soulèvent une erreur concernant le fuseau horaire : "Invalid TimeZone Displacement Value".

En attendant de résoudre ce problème, je laisse les colonnes de dates de naissance et de mort en format texte, et vérifie que l'insertion des données fonctionne bien, dans une base de données relationnelles où les tables sont convenablement reliées.

[12] Documentation Pandas : pandas.DataFrame.loc
[13] Documentation Pandas : pandas.DataFrame.drop

D. Incrément : dates exploitables

Le problème de dates s'explique : PostgreSQL croit que celles-ci cherchent à indiquer un décalage horaire, et supérieur aux limites fixées. La cause est le signe négatif placé devant les années qui précèdent la naissance du Christ. Bien que la documentation des formats de date de PostgreSQL n'en parle pas, et que les années négatives soient acceptées dans la norme ISO utilisée, c'est manifestement de là que vient le problème. PostgreSQL attend que les années négatives soit exprimées par BC (Before Christ), sous le format "yyy-mm-ddBC".

Pour régler ce problème, je dois encore une fois avoir recours aux Regex et à pandas.DataFrame.replace(). Cependant, la situation est un peu plus complexe car il faut à la fois supprimer le signe négatif en début de chaîne de caractères et rajouter BC à la fin, sans toucher à la partie centrale. Pour conserver et réutiliser une partie de la chaîne, je lis qu'il faut d'abord la mettre entre parenthèses pour la sélectionner, puis pour la réutiliser écrire $1, selon certains, ou \1 selon d'autres. Ce qui donne "^-([0-9]*-[0-9]*-[0-9]*)$" comme sélection, et "$1BC" ou "\1BC" comme remplacement. Mais ça ne marche pas, la regex de remplacement n'est pas interprétée comme regex mais comme chaîne de caractères à substituer à la sélection (alors même que j'ai spécifié regex=True). Je finis par découvrir que pour garantir la lecture comme regex, je peux précéder la chaîne de caractères d'un r minuscule[14]. Le remplacement fonctionne, les dates sont au bon format.

Cependant l'insertion ne fonctionne toujours pas. Une date en particulier pose problème : l'an 0000, qui n'existe pas et pose problème à PostgreSQL. A l'aide d'une autre regex, je la transforme en 1 avant Jésus-Christ.

Dès lors, l'insertion au format de date ne pose plus problème.

E. Incrément : base élargie

Plus tôt, lors d'une tentative d'insertion dans la base de données, je m'étais aperçu que pour certaines entrées dans la table 'mariage', l'identifiant de l'épouse ne correspondait à personne de listé dans la table 'personne'. Cela posait problème car j'avais défini cet identifiant de l'épouse comme clef étrangère se rapportant à la table 'personne'. J'avais évacué le problème en créant une liste des identifiants de la table 'personne' et en supprimant de celle

[14] StackOverFlow : pandas applying regex to replace values

'mariage' toutes les entrées qui ne correspondaient à rien. Mais c'était un pis-aller, et l'absence de ces femmes montrait que ma requête originelle n'était pas assez large, et manquait des personnes pourtant référencées sur Wikidata.

J'ai donc décidé d'élargir la base, avec une nouvelle extraction. J'ai d'abord recherché sur Wikidata Query Service toutes les personnes dont le conjoint était romain. J'ajoutais les informations du nouveau fichier CSV au dataframe df_personne, puis supprimais les (évidemment très nombreux) doublons. Mais ça n'avait pas un grand intérêt : après dédoublonnage il y avait assez peu de nouvelles entrées, et certaines étaient d'encore plus mauvaise qualité, ne présentant même pas de nom.

Cependant, en regardant certaines pages d'épouses qui étaient absentes de ma première extraction, j'en compris la raison et un moyen d'y remédier : elles n'avaient pas de propriété "pays de citoyenneté" correspondant à la Rome antique, mais une propriété "période historique" correspondant à l'antiquité romaine. La même information était transmise différemment.

En fait, il y avait en tout six périodes historiques concernées : la monarchie, le début, le milieu et la fin de la République, puis le Haut et le Bas-Empire. De la même manière que précédemment, j'en tirais des fichiers CSV puis ajoutais les données au dataframe df_personne et supprimais les nombreux doublons ; mais cette fois-ci le nombre d'entrées a augmenté de manière significative.

F. Incrément : script optimisé

Le script de nettoyage des données, création de la base de données et insertion est réécrit en respectant les consignes PEP8, optimisé pour la performance, et commenté. Les données sont sauvegardées.

G. Incrément : données géographiques

Initialement, j'avais prévu d'ajouter une table de lieux géographiques, en ne me basant plus uniquement sur Wikidata mais aussi sur l'Atlas digital de l'empire romain réalisé par l'université de Lund. Malheureusement ça s'est avéré plus difficile que prévu, car les données ne concordent pas entre elles. Le nom d'une ville (latin ou moderne) a telle graphie sur une source et telle graphie sur l'autre. Je pensais me baser sur les coordonnées, qui sont a priori plus universelles, mais là aussi c'était moins évident que prévu, car les coordonnées

précises indiquées sont différentes. Il faut trouver le bon niveau de précision, pour pouvoir tout de même distinguer des localités proches. Un autre problème est que nombre de lieux données comme lieux de naissance ou de mort ne sont pas des villes aux coordonnées précises, mais des régions parfois très grandes, aux contours flous et donc aux coordonnées de référence aléatoires, sans concordance possible entre les deux sources.

IV. Exploitation

A. Exploration des données

On peut commencer à explorer les données de la base, via des requêtes SQL, sur pgAdmin4, DBeaver ou éventuellement sur un Notebook. Pour commencer, une requête assez naturelle serait d'avoir la liste des empereurs romains :

```sql
SELECT DISTINCT personne.id, personne.nom_fr, date_naissance, date_mort
FROM personne
INNER JOIN poste ON personne.id = poste.personne_id
WHERE poste.poste = 'empereur romain'
ORDER BY date_mort
```

On obtient effectivement une liste, dont on peut voir un aperçu ici :

	id	nom_fr	date_naissance	date_mort
1	1 405	Auguste	0062-01-01	0014-08-17
2	1 407	Tibère	0041-11-14	0037-03-14
3	1 409	Caligula	0012-08-29	0041-01-22
4	1 411	Claude	0009-07-30	0054-10-11
5	154 732	Agrippine la Jeur	0015-11-04	0059-03-15
6	1 414	Galba	0002-12-22	0069-01-13
7	1 416	Othon	0032-04-26	0069-04-14
8	1 417	Vitellius	0015-09-22	0069-12-20
9	1 419	Vespasien	0009-11-15	0079-06-21
10	1 421	Titus	0039-12-28	0081-09-11
11	1 424	Nerva	0032-01-01	0098-01-25
12	1 425	Trajan	0053-09-16	0117-08-07
13	1 427	Hadrien	0076-01-22	0138-07-09
14	1 429	Antonin le Pieux	0086-09-17	0161-03-06
15	1 433	Lucius Verus	0130-12-14	0169-01-01
16	1 430	Marc Aurèle	0121-04-25	0180-03-16
17	1 434	Commode	0161-08-30	0192-12-30

Autre requête, cette fois-ci un peu plus complexe car demandant à se re-joindre une deuxième fois sur la table 'personne' : la liste des femmes d'empereur romain, et leur époux.

```sql
SELECT DISTINCT p2.id, p2.nom_fr AS "Impératrice", personne.nom_fr AS "Empereur"
FROM personne
INNER JOIN poste ON personne.id = poste.personne_id
INNER JOIN mariage ON personne.id = mariage.id_mari
INNER JOIN personne p2 ON mariage.id_epouse = p2.id
WHERE poste.poste = 'empereur romain'
```

	id	Impératrice	Empereur
1	2 259	Julia Caesaris filia	Tibère
2	45 522	Minervina	Constantin Ier
3	45 530	Eutropia	Maximien Hercule
4	45 535	Valeria Maximilla	Maxence
5	63 533	Faustina	Constance II
6	154 732	Agrippine la Jeune	Claude
7	164 210	Prisca	Dioclétien
8	170 164	Hélène	Constance Chlore
9	174 323	Placidia	Flavius Anicius Olybrius
10	229 246	Julia Domna	Septime Sévère
11	229 871	Messaline	Claude
12	230 716	Poppée	Othon
13	231 063	Fausta	Constantin Ier
14	232 090	Vipsania Agrippina	Tibère
15	232 094	Théodora	Constance Chlore
16	232 329	Licinia Eudoxia	Pétrone Maxime
17	232 329	Licinia Eudoxia	Valentinien III

On peut chercher à savoir quelles sont les familles les plus représentées au Sénat (parmi les sénateurs dont nous ayons gardé une trace) :

```sql
SELECT DISTINCT personne.gens, COUNT(*)
FROM personne
INNER JOIN poste ON personne.id = poste.personne_id
WHERE poste.poste = 'sénateur romain'
GROUP BY personne.gens
ORDER BY COUNT(*) DESC
```

	gens	count
1	[NULL]	254
2	Cornelii	135
3	Flavii	119
4	Valerii	77
5	Julii	72
6	Claudii	66
7	Fabii	47
8	Junii	43

Autre comptage plus parlant : celui du sexe des personnes répertoriées :

```
sql_sexes = """
    SELECT DISTINCT personne.sexe, COUNT(*)
    from personne
    group by personne.sexe
    order by count(*) desc;
    """

results = pd.read_sql_query(sql_sexes, engine)
results.head(10)
```

	sexe	count
0	masculin	6664
1	féminin	637
2	None	144
3	eunuque	2
4	intersexuation	1

Le résultat est impressionnant : les hommes sont très largement majoritaires. Mais bien plus surprenant et intéressant est cette apparition d'eunuques et surtout d'un individu intersexué. Qui cela peut-il bien être ? Pour le savoir, il suffit de demander à la base :

```
SELECT personne.id, personne.nom_fr, personne.sexe, personne.date_naissance,
personne.date_mort, activite.activite
FROM personne
inner JOIN activite ON activite.personne_id=personne.id
WHERE personne.sexe = 'intersexuation';
```

On découvre ainsi Favorinus d'Arles, un philosophe, écrivain et poète du deuxième siècle après Jésus-Christ, dont on dit effectivement qu'il était hermaphrodite.

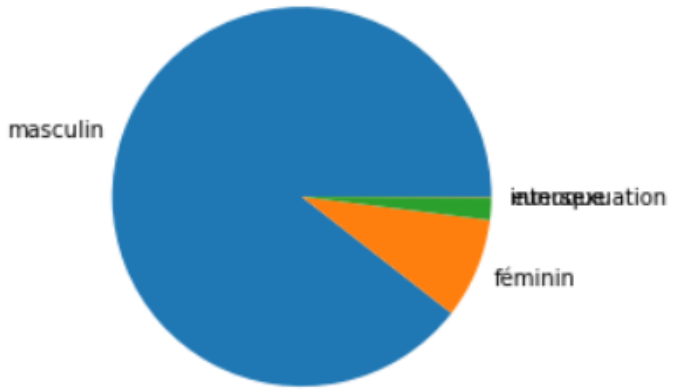
	id	nom_fr	sexe	date_naissance	date_mort	activite
1	554 387	Favorinus d'Arles	intersexuation	0085-01-01	0200-01-01	écrivain
2	554 387	Favorinus d'Arles	intersexuation	0085-01-01	0200-01-01	poète
3	554 387	Favorinus d'Arles	intersexuation	0085-01-01	0200-01-01	philosophe

B. Visualisations

On peut représenter la forte disproportion entre les sexes par un graphique. Pour cela on utilise la librairie matplotlib.

```
x = results['sexe']
y = results['count']
```

```
plt.figure()
plt.pie (y, labels=x)
plt.show()
```



On peut aussi représenter graphiquement le nombre de personnages recensés par année de naissance (ou plutôt décennie, vu la durée totale couverte, mais on peut régler ça directement au niveau du graphique) :

```
sql_annee = """
    SELECT date_part('year', date_naissance) AS "Année", COUNT(*) AS "Nombre de personnes listées"
    FROM personne
    WHERE date_naissance IS NOT NULL
    GROUP BY date_part('year', date_naissance)
    ORDER BY date_part('year', date_naissance) DESC;
    """

results = pd.read_sql_query(sql_annee, engine)
results.head(10)
```

	Année	Nombre de personnes listées
0	1415.0	1
1	1000.0	3
2	630.0	1
3	600.0	2
4	550.0	2
5	524.0	1
6	500.0	32
7	498.0	1
8	495.0	1
9	493.0	1

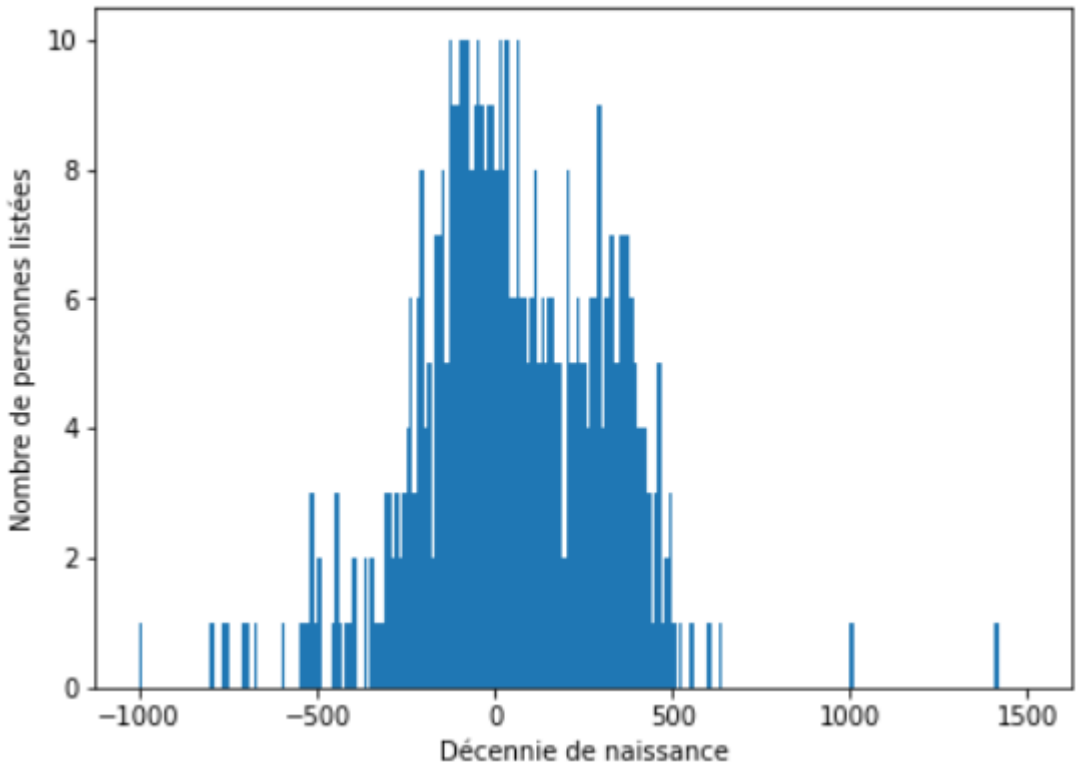
Je vais représenter cela à l'aide d'un histogramme, qui se définit par un découpage de l'abscisse x en intervalles. Je choisis de découper l'intervalle entre -1000 et 1500 en 250 périodes de dix ans.

```
data = results

x = data["Année"]
y = data["Nombre de personnes listées"]
```

```
fig, ax = plt.subplots(figsize=(7, 5))
ax.hist(x, bins=250, range=(-1000, 1500))
ax.set_xlabel('Décennie de naissance')
ax.yaxis.set_label_text('Nombre de personnes listées')

plt.show()
```



Comme on peut le voir l'immense majorité des individus listés sont nés entre 500 avant et 500 après Jésus-Christ, ce qui est normal, c'est la période couverte. En revanche, plus intéressant est la division en deux pics : le premier siècle avant Jésus-Christ et le quatrième siècle après. La fameuse "crise du troisième siècle" semble visible sur ce graphique.

C. Critique des sources

Au cours de cette exploration et visualisation des données, on aura facilement remarqué un certain nombre de résultats aberrants. Dans la liste des empereurs, on trouve Agrippine la Jeune, qui n'a jamais régné en son nom propre. Favorinus d'Arles n'a pas vécu 125 ans. Jean Argyropoulos, né en 1415 et compté tout à droite du graphique précédent, n'a pas vécu dans l'antiquité romaine.

Tout cela permet de voir que la qualité des données est loin d'être excellente. Wikidata souffre des mêmes limites que Wikipédia, mais est aussi moins édité, donc moins corrigé.

V. Conclusions

Le client dispose d'une base de données fonctionnelle et bien formatée de d'informations biographiques sur les personnages historiques notables de l'antiquité romaine. Son exploration permet de dégager des curiosités intéressantes pour le public, ainsi que de réaliser facilement des représentations graphiques.

Il reste beaucoup de possibilités d'amélioration et d'enrichissement de la base, d'abord par l'ajout des données géographiques. Mais le client pourrait aussi émettre des doutes sur la qualité des informations de Wikidata, et demander l'utilisation d'une deuxième source complémentaire pour vérifier les informations fournies, ou bien ne se servir des données de Wikidata que comme une base à enrichir ensuite de données spécialisées plus fiables.

J'aimerais remercier mes formateurs, les camarades qui m'ont aidé et motivé d'une manière ou d'une autre, l'équipe de Simplon et mes maîtres de stage pour tout ce qu'ils m'ont apporté pendant cette période de formation.