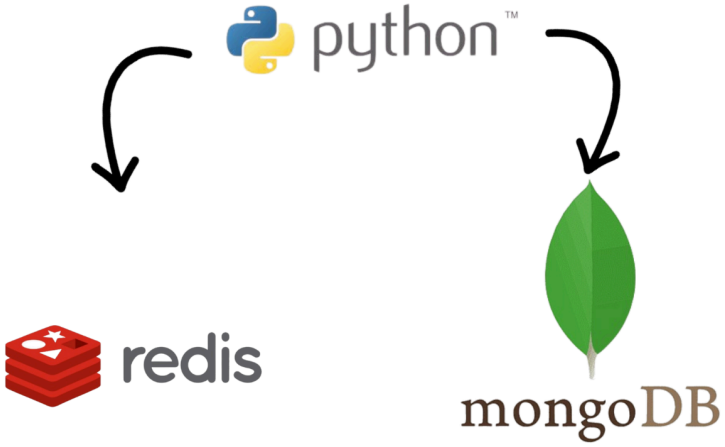
Tharindu PERERA

Implémentation et Étude de Bases de Données NoSQL pour la Gestion des Réservations Aériennes : Comparaison entre Redis et MongoDB



BUT 3 – GERYON 2024

I) Introduction

Contexte et objectifs du projet

Dans le cadre de la mission pour notre client, une entreprise de gestion des réservations de vols, notre équipe a cherché une solution visant à améliorer la performance de leur système de gestion de données. Notre production offre des fonctionnalités telles que **`GetPilotes`**, **`GetVols`**, **`GetClients`** `et d'autres fonctionnalités` permettant une manipulation efficace des données.

Pour ce faire, nous avons étudié deux options de bases de données NoSQL : Redis et MongoDB. L'objectif principal est de modéliser et manipuler les données liées aux vols, aux clients, et aux réservations de manière efficace, tout en exploitant les avantages de chaque technologie NoSQL pour le traitement de grandes quantités de données.

Le projet a été divisé en plusieurs étapes :

- **Dénormalisation et modélisation des données** en vue de leur stockage dans Redis et MongoDB.
- **Développement de fonctions Python** pour le requêtage et la manipulation des données, permettant de tester et de comparer les deux bases.

II) Matériels utilisés

Pour mener à bien ce projet, nous avons utilisé un ensemble d'outils et de technologies visant à garantir une gestion efficace des données dans un environnement NoSQL. L'environnement de travail, les outils logiciels ainsi que les procédures suivies sont décrits ci-dessous :

1. Redis (version 7.0.15)

Environnement utilisé : Linux

Le projet a été réalisé sous un système d'exploitation Linux, reconnu pour sa stabilité, ses capacités et ses performances. Toutes les opérations ont été effectuées dans cet environnement.

Base de données : Redis

La base de données Redis a été choisie pour son architecture NoSQL qui permet une gestion rapide et flexible des données sous forme de paires clé-valeur.

Langage de programmation :

Python et utilisation du module Redis-Python Toutes les opérations sur la base Redis ont été effectuées en Python, avec l'utilisation du module Redis-Python. Ce module permet de se connecter à Redis et de manipuler les données stockées en créant des programmes en Python.

Procédures suivies

- **Installation des outils :**
  - Installation de Redis et configuration sur Linux.
  - Installation de Python et du module Redis-Python dans un environnement virtuel.
- **Préparation des données :**
  - Les données relatives aux réservations ont été transformées en objets JSON et insérées dans Redis.
- **Requêtes et manipulations :**
  - Des scripts Python ont été utilisés pour récupérer et manipuler les données, ainsi que pour tester différentes structures de données (Bloom filter et Liste)

2. MongoDB

Environnement utilisé : Linux

Base de données : MongoDB (version 2.3.3)

MongoDB est une base de données NoSQL orientée documents, qui stocke les données sous forme de **documents JSON** regroupés en **collections**. Chaque document contient des paires clé-valeur, et les collections, semblables aux tables SQL, permettent de rassembler différents types de documents sans structure fixe. Ce modèle flexible facilite l'ajout et la modification de champs sans affecter les autres enregistrements. MongoDB offre également des options d'indexation et un puissant système de requêtes et d'agrégation pour des recherches rapides et des analyses avancées.

Langage de programmation : Python et utilisation du module PyMongo

Procédures suivies : Installation, configuration de l'environnement, et préparation des données

III) Méthode et Résultats

Dénormalisation des données

Pour répondre aux exigences de gestion des données liées aux vols et aux réservations, une approches de dénormalisation a été envisagée :

Modélisation centrée sur les réservations

La structure de données adoptée dans cette solution est centrée sur les réservations, comme on peut le voir dans la figure 1, on rassemble en un seul objet JSON toutes les informations nécessaires, telles que le client, le vol, le pilote et l'avion. Ce compte rendu détaillera les étapes de mise en place des bases de données
Redis et MongoDB, de l'installation et la configuration jusqu'à l'implémentation des données et l'évaluation des performances.

Pour Redis, la Décomposition des éléments se fera de la manière suivante:

"**id**" : identifiant unique de la réservation

"**client**" : contient les détails du client, comme l'ID, le nom, l'adresse (numéro de rue, nom de rue, code postal, ville),

"**vol**" : inclut des informations sur le vol, telles que l'ID, les villes de départ et d'arrivée, les dates et heures de départ et d'arrivée, ainsi que le pilote et l'avion,
"**classe**" : type de billet avec un coefficient de prix,
"**places**" : nombre de places réservées**.**


```
reservations:1
{
  "id": "1001_V690",
  "client": {
    "id": "1001",
    "nom": "Becquerel",
    "numeroRue": "24",
    "nomRue": "Rue De Fabre",
    "codePostal": "13015",
    "ville": "Marseille"
  },
  "vol": {
    "id": "V690",
    "villeDepart": "Marseille",
    "villeArrivee": "Pekin",
    "dateDepart": "19/04/07",
    "heureDepart": "18:00",
    "dateArrivee": "20/04/07",
    "heureArrivee": "6:15",
    "pilote": {
      "id": "4050",
      "nom": "Dumas",
      "naissance": "1949",
      "ville": "Marseille"
    },
    "avion": {
      "id": "440",
      "nom": "Concorde",
      "capacite": "100",
      "ville": "Marseille"
    }
  },
  "classe": {
    "nom": "Business",
    "coeffPrix": 2
  },
  "places": 3
}
```


Figure 1 :Objet JSON représentant une valeur de la table réservations après transformation (Redis)

Pour MongoDB, c'est la même chose, cependant la figure 2 montre que .l'affichage sera différent, comme on peut le voir dans l'image ci dessous, une fois dans la BD réservations si on prends une ligne de la BD, on va avoir un affichage avec les principaux objets

Figure 2: : Objet JSON représentant une entrée de la collection réservations

Et si on sélectionne un des objets comme dans l'image 3 avec la commande findOne() pour voir le contenu d'une table on aura ses informations, juste après l'exécution de la commande, avec son ID unique et le reste, et on peut tout à fait mettre d'autres Objets JSON dedans comme sur la dernière ligne

Figure 3 : Exemple d'une ligne dans une collections

1) Redis

Dans un premier temps, pour se connecter à redis il suffit simplement de faire “Redis-cli” depuis le terminal et si les ports ont été changé il faudra ajouter la mention -p [numéro du port]

1.1 Requêtage de la base Redis

Une fois les données injectées dans Redis, nous avons développé plusieurs fonctionnalités Python pour répondre aux besoins de nos clients concernant la manipulation et l'interrogation des informations stockées.

Fonctionnalité 1 : Lister toutes les villes d’arrivée et de départs

```python
import redis
import json
redis_client = redis.Redis(host='localhost', port=6380, db=0)

# Ensemble pour stocker les villes de départ et d'arrivée
villes_depart = set()
villes_arrivee = set()

# Parcours des clés de réservations stockées dans Redis
for i in range(1, 100):  # Supposons qu'il y a moins de 100 réservations
    reservation_key = f"reservations:{i}"

    # Vérification si la clé existe dans Redis
    if redis_client.exists(reservation_key):
        # Récupérer la réservation en JSON et la charger en dict Python
        reservation_data = json.loads(redis_client.get(reservation_key))

        # Ajouter les villes de départ et d'arrivée aux ensembles correspondants
        villes_depart.add(reservation_data['vol']['villeDepart'])
        villes_arrivee.add(reservation_data['vol']['villeArrivee'])
    else:
        # Si aucune clé supplémentaire n'existe, on peut arrêter la boucle
        break

# Affichage des résultats
print("Villes de départ:", villes_depart)
print("Villes d'arrivée:", villes_arrivee)
```

Figure 4 : Programme listant les provenances et destinations des avions

Cette fonctionnalité montrée dans la figure a pour but d'extraire la liste de toutes les villes d’arrivée et de départs des vols stockés dans la BD Redis. Elle est essentielle pour permettre une vue d'ensemble des provenances et destinations disponibles, facilitant ainsi les requêtes et l'analyse des données de réservation.

Explication du code de la figure 5 : Ce script liste les villes de départ et d'arrivée dans Redis. Deux ensembles villes_depart et villes_arrivee sont créés pour stocker les villes sans doublons (Ligne 6-7). Pour chaque clé de réservation (reservations:1 à reservations:99 ligne 10), la ville de départ et d'arrivée est ajoutée à l'ensemble correspondant (lignes 19-20).

```
tharindu@tharindu-VirtualBox:~/Bureau/bdd./bdd$ python3 villes.py
Villes de départ: {'Marseille', 'Strasbourg', 'Amsterdam', 'Ajaccio', 'Paris', '
Metz'}
Villes d'arrivée: {'Marseille', 'Amsterdam', 'Pekin', 'Nice'}
```

Figure 4a : Exécution du programme

La figure 4a montre l'exécution du programme et comme on le voit dans la ligne suivante cela nous renvoie toutes les villes d'arrivées et de départ enregistrés

Fonctionnalité 2 : Compter le nombre de pilotes

```
1  import redis
2  import json
3
4  # Connexion à Redis
5  redis_client = redis.Redis(host='localhost', port=6380, db=0)
6
7  def get_NBpilotes():
8      pilotes = set()  # Utilisation d'un ensemble pour éviter les doublons
9
10     # Parcours de toutes les clés correspondant au modèle "reservations:*"
11     for reservation_key in redis_client.scan_iter("reservations:*"):
12         # Récupérer la réservation en JSON et la charger
13         reservation_data = json.loads(redis_client.get(reservation_key))
14         # Accéder au nom du pilote dans les informations de vol
15         pilotes.add(reservation_data['vol']['pilote']['nom'])
16
17     nombre_total_pilotes = len(pilotes)
18
19     return pilotes, nombre_total_pilotes
20
21 # Appel de la fonction et affichage des résultats
22 pilotes_trouves, nombre_total = get_NBpilotes()
23 print("Noms des pilotes :", pilotes_trouves)
24 print("Nombre total de pilotes :", nombre_total)
```

Figure 5 :Programme Noms et Nombres de Pilotes

Cette fonction issue de la figure 5 parcourt les objets enregistrés dans Redis et compte le nombre de pilotes différent en activité. Cela permet de filtrer rapidement les pilotes, ce qui est utile pour des rapports.

Explication du code de la figure 5 : Le script utilise Redis pour compter les pilotes uniques. Il initialise un ensemble pilotes, parcourt les clés de réservation (reservations:* ligne 10), récupère le nom de chaque pilote, et évite les doublons (lignes 7-14). Le nombre total est affiché ensuite (lignes 22).

```
tharindu@tharindu-VirtualBox:~/Bureau/bdd./bdd$ python3 nomspilotes.py
Noms des pilotes : {'Delalande', 'Duval', 'Dubois', 'Dumas', 'Leblanc'}
Nombre total de pilotes : 5
```

Figure 5a : Exécution du programme

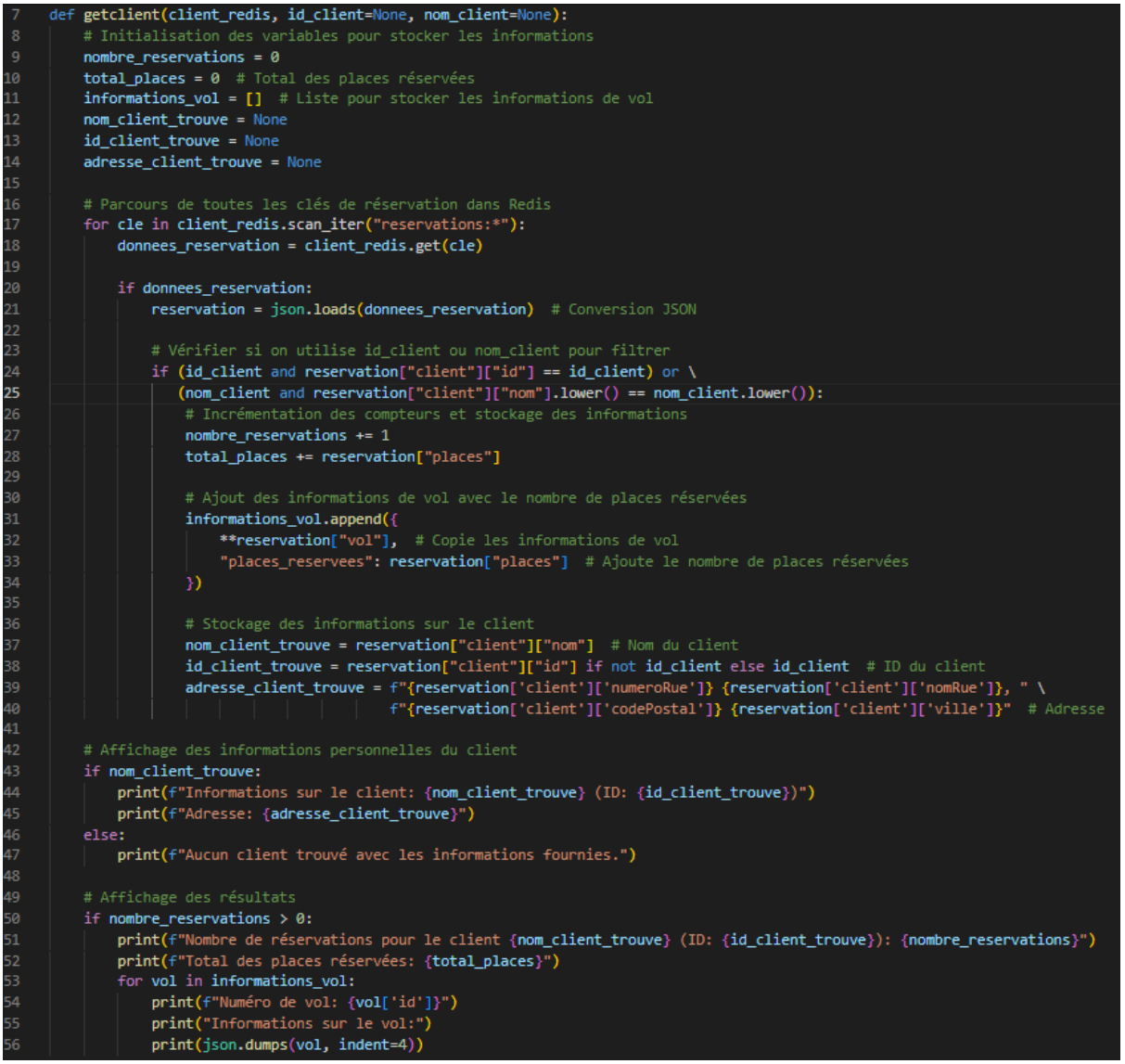
La figure 5a représente l'exécution du programme qui va dans la ligne suivante nous donner le noms des pilotes en activités

1.2 Opération de jointure

Une des tâches suivantes consistait à simuler une opération de jointure entre deux ensembles de données. Redis étant un système NoSQL qui ne supporte pas par défaut les jointures comme les bases de données relationnelles, nous n'avons pas trouvé de moyen pour faire fonctionner le programme de jointure à chaque fois, en effet des fois il affichait des résultats semblants corrects et d'autres fois non.

Pour palier à ce manque nous avons créé 3 fonctions de recherches permettant de remplacer des jointures interne (**INNER JOIN)**

La Fonctionnalité interactive GETCLIENT de la figure 6 remplace donc un **INNER JOIN** entre les tables `reservations` et `clients` avec une condition de filtre sur l'identité du client.



```python
def getclient(client_redis, id_client=None, nom_client=None):
    # Initialisation des variables pour stocker les informations
    nombre_reservations = 0
    total_places = 0  # Total des places réservées
    informations_vol = []  # Liste pour stocker les informations de vol
    nom_client_trouve = None
    id_client_trouve = None
    adresse_client_trouve = None

    # Parcours de toutes les clés de réservation dans Redis
    for cle in client_redis.scan_iter("reservations:*"):
        donnees_reservation = client_redis.get(cle)

        if donnees_reservation:
            reservation = json.loads(donnees_reservation)  # Conversion JSON

            # Vérifier si on utilise id_client ou nom_client pour filtrer
            if (id_client and reservation["client"]["id"] == id_client) or \
               (nom_client and reservation["client"]["nom"].lower() == nom_client.lower()):
                # Incrémentation des compteurs et stockage des informations
                nombre_reservations += 1
                total_places += reservation["places"]

                # Ajout des informations de vol avec le nombre de places réservées
                informations_vol.append({
                    **reservation["vol"],  # Copie les informations de vol
                    "places_reservees": reservation["places"]  # Ajoute le nombre de places réservées
                })

                # Stockage des informations sur le client
                nom_client_trouve = reservation["client"]["nom"]  # Nom du client
                id_client_trouve = reservation["client"]["id"] if not id_client else id_client  # ID du client
                adresse_client_trouve = f"{reservation['client']['numeroRue']} {reservation['client']['nomRue']}, " \
                                        f"{reservation['client']['codePostal']} {reservation['client']['ville']}"  # Adresse

    # Affichage des informations personnelles du client
    if nom_client_trouve:
        print(f"Informations sur le client: {nom_client_trouve} (ID: {id_client_trouve})")
        print(f"Adresse: {adresse_client_trouve}")
    else:
        print(f"Aucun client trouvé avec les informations fournies.")

    # Affichage des résultats
    if nombre_reservations > 0:
        print(f"Nombre de réservations pour le client {nom_client_trouve} (ID: {id_client_trouve}): {nombre_reservations}")
        print(f"Total des places réservées: {total_places}")
        for vol in informations_vol:
            print(f"Numéro de vol: {vol['id']}")
            print("Informations sur le vol:")
            print(json.dumps(vol, indent=4))
```

Figure 6 : Fonction GetClient

Explication du code de la figure 6 : Après avoir importé les bibliothèques nécessaires, on définit la fonction principale getClient à la ligne 7. Elle prend trois paramètres : client_redis (la connexion à Redis), un id_client, et un nom_client.

Dans les lignes 9 à 14, on prépare plusieurs variables pour stocker :
le nombre total de réservations (nombre_reservations),
le nombre total de places réservées (total_places),
les infos de chaque vol (informations_vol),
et les infos personnelles du client comme le nom, l'ID et l'adresse.

Ensuite, la fonction boucle sur toutes les clés de réservation dans Redis avec le format reservations:* (ligne 17). Pour chaque clé, elle récupère les données de réservation en les convertissant en dictionnaire Python (ligne 21). Elle vérifie si le client de la réservation correspond soit à l'id_client, soit au nom_client fourni (lignes 24-25). Si c'est bien le cas, elle met à jour :

le nombre total de réservations (nombre_reservations est incrémenté à la ligne 26),
le nombre total de places (total_places augmente du nombre de places réservées, ligne 27),
et les informations de vol, qui sont ajoutées à la liste informations_vol (lignes 31-34).
Les infos du client sont alors récupérées et stockées dans nom_client_trouve, id_client_trouve et adresse_client_trouve (lignes 37-40). Une fois la boucle terminée, la fonction vérifie si elle a bien trouvé un client. Si oui, elle affiche les infos personnelles du client (lignes 42-44). Si aucun client n'est trouvé, elle indique que le client n'a pas été trouvé (ligne 47).

Si le client a au moins une réservation, la fonction affiche le nombre total de réservations et le total des places réservées (lignes 49-51). Pour chaque réservation, elle affiche aussi les détails du vol (lignes 52-55).

Suite à l'exécution de ce code dans le terminal de la figure 6a, le client pourra choisir de rentrer l'id du client **OU** son nom et pourra récupérer toutes les informations le concernant soit : Son nom et ID, l'adresse, le nombre de réservations qu'il a faites ainsi que les informations sur chacune des réservations

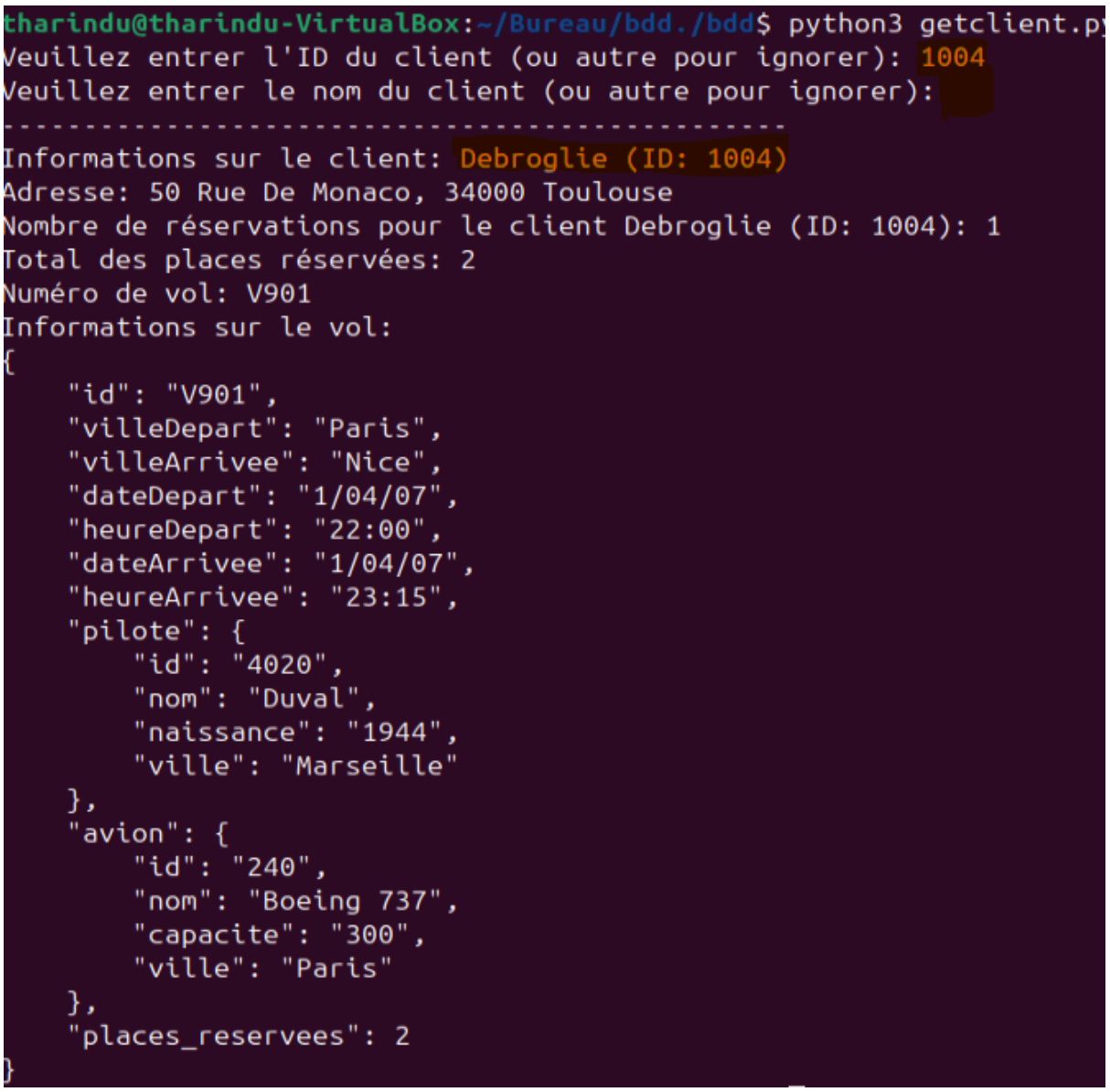


Figure 6a : execution du programme et son résultat

La Fonction GETPILOTES est une fonction interactive qui remplace donc un **INNER JOIN** entre les tables `reservations` et `pilotes` avec une condition de filtre sur l'identité du pilote.

```python
def getpilotes(client_redis, id_pilote=None, nom_pilote=None):
    # Liste pour stocker les informations de vol
    informations_vol = []

    # Variables pour stocker les informations du pilote trouvé
    nom_pilote_trouve = None
    id_pilote_trouve = None
    date_naissance_pilote_trouve = None
    ville_pilote_trouve = None

    # Parcours de toutes les clés de réservation dans Redis
    for cle in client_redis.scan_iter("reservations:*"):
        donnees_reservation = client_redis.get(cle)

        if donnees_reservation:
            reservation = json.loads(donnees_reservation)
            informations_pilote = reservation["vol"]["pilote"]

            # Vérifier si on utilise id_pilote ou nom_pilote pour filtrer
            if (id_pilote and informations_pilote["id"] == id_pilote) or \
            (nom_pilote and informations_pilote["nom"].lower() == nom_pilote.lower()):

                informations_vol.append(reservation["vol"])  # Ajoute les informations de vol à la liste
                nom_pilote_trouve = informations_pilote["nom"]
                id_pilote_trouve = informations_pilote["id"] if id_pilote else id_pilote
                date_naissance_pilote_trouve = informations_pilote["naissance"]
                ville_pilote_trouve = informations_pilote["ville"]


    # Utiliser un ensemble pour éviter les doublons dans les informations de vol
    vols_uniques = {vol['id']: vol for vol in informations_vol}.values()

    # Affichage des informations personnelles du pilote
    if nom_pilote_trouve:
        print(f"Informations sur le pilote: {nom_pilote_trouve} (ID: {id_pilote_trouve})")
        print(f"Date de naissance: {date_naissance_pilote_trouve}, Ville: {ville_pilote_trouve}")
    else:
        print("Aucun pilote trouvé avec les informations fournies.")

    # Affichage des résultats des vols
    if vols_uniques:
        for vol in vols_uniques:
            print(f"Numéro de vol: {vol['id']}, "
                  f"Avion: {vol['avion']['nom']}, "
                  f"Ville de départ: {vol['villeDepart']}, "
                  f"Ville d'arrivée: {vol['villeArrivee']}, "
                  f"Date de départ: {vol['dateDepart']}, "
                  f"Heure de départ: {vol['heureDepart']}, "
                  f"Date d'arrivée: {vol['dateArrivee']}, "
                  f"Heure d'arrivée: {vol['heureArrivee']}")
    else:
        print(f"Aucun vol trouvé pour le pilote {nom_pilote_trouve} (ID: {id_pilote_trouve}).")

```

Figure 7 : Fonction GetPilotes

Explication du code de la figure 7 : La fonction principale getPilotes, définie à la ligne 7, prend client_redis, id_pilote, et nom_pilote comme paramètres et sert à récupérer les infos d'un pilote spécifique. Elle initialise une liste vide informations_vol (ligne 9) pour y ajouter les détails des vols trouvés. Les variables pour les infos du pilote (nom_pilote_trouve, id_pilote_trouve, date_naissance_pilote_trouve, et ville_pilote_trouve) sont aussi prêtes à accueillir les infos trouvées (lignes 12-15).

La fonction parcourt toutes les réservations dans Redis (reservations:* ligne 18) pour chaque clé de réservation, et elle convertit les données en JSON (ligne 22). Ensuite, elle vérifie si l'id_pilote correspond (lignes 25-28) ou si le nom_pilote correspond (lignes 30-34) et met à jour les infos et la liste informations_vol.

Pour éviter les doublons, un ensemble vols_uniques est créé (ligne 37). Si le pilote est trouvé, ses infos sont affichées (lignes 40-43) ; si non, un message d'erreur apparaît (ligne 45). Ensuite, les détails des vols sont affichés (lignes 48-56), ou un message indique qu'aucun vol n'a été trouvé (ligne 57).

On peut observer sur la figure 7a que lorsque l'on saisit le nom du pilote (ou l'ID) notre client aura toutes les informations le concernant, à savoir : son nom et ID, année de naissance, sa ville de résidence, ainsi que les détails de chaque vol associé, incluant le numéro de vol, le type d'avion, les villes de départ et d'arrivée, ainsi que les dates et heures de départ et d'arrivée.

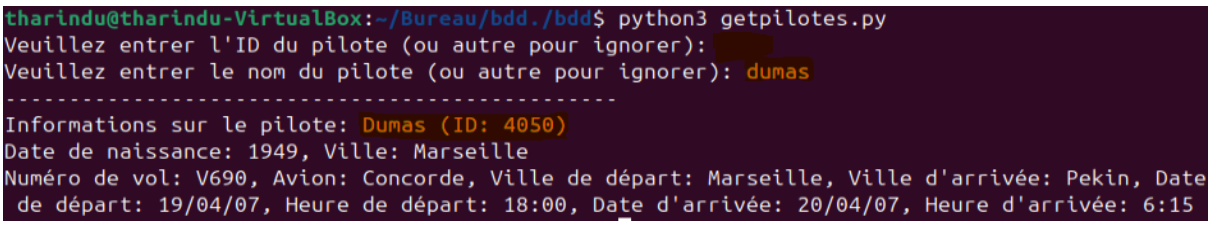
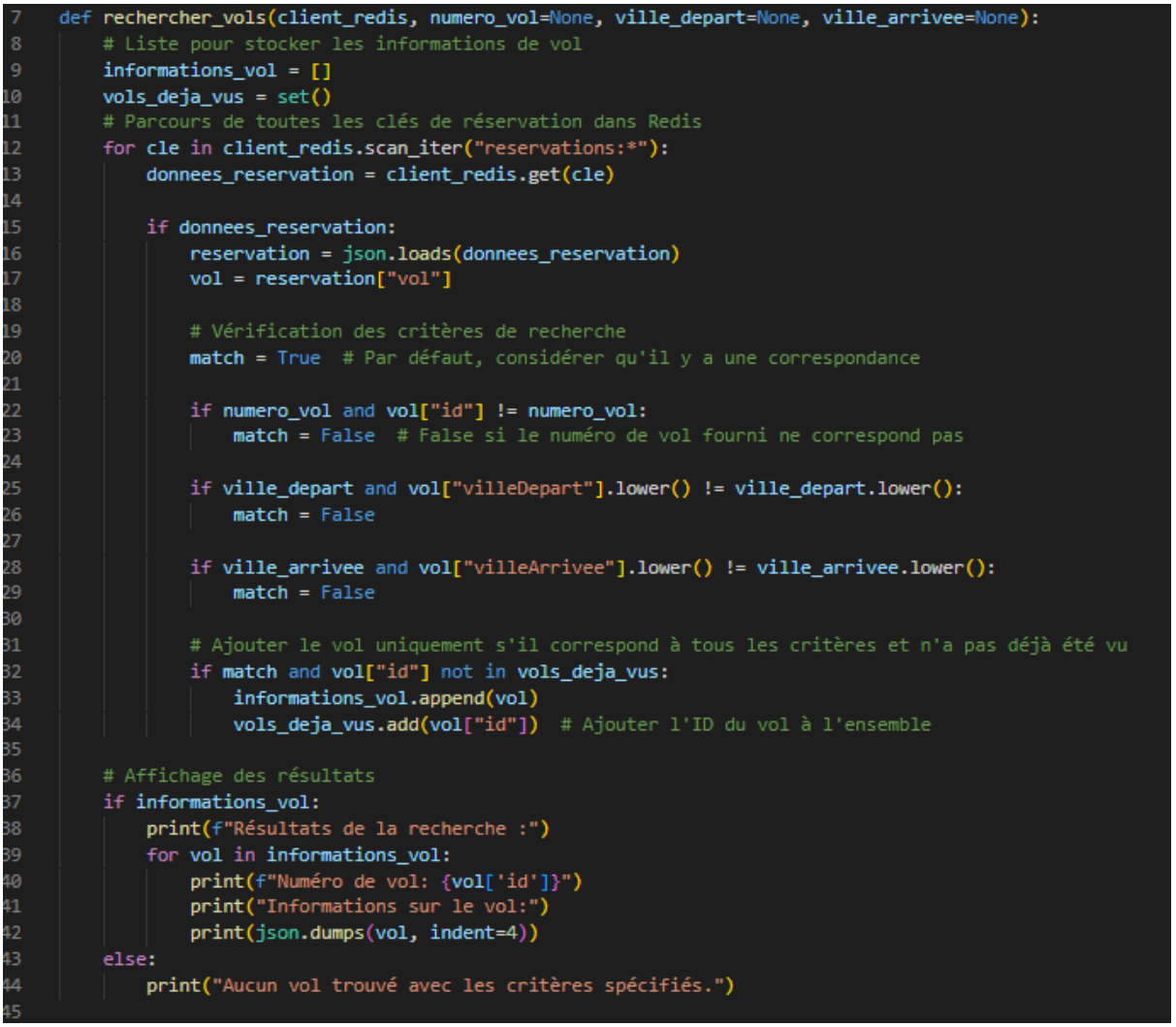


Figure 7a : exécution du programme et son résultat

La Fonction interactive GETVOLS remplace un **INNER JOIN** entre les tables `reservations` et `vols`, en appliquant des filtres sur les colonnes “`id`, `villeDepart`, et `villeArrivee`.”



```
7   def rechercher_vols(client_redis, numero_vol=None, ville_depart=None, ville_arrivee=None):
8       # Liste pour stocker les informations de vol
9       informations_vol = []
10      vols_deja_vus = set()
11      # Parcours de toutes les clés de réservation dans Redis
12      for cle in client_redis.scan_iter("reservations:*"):
13          donnees_reservation = client_redis.get(cle)
14
15          if donnees_reservation:
16              reservation = json.loads(donnees_reservation)
17              vol = reservation["vol"]
18
19              # Vérification des critères de recherche
20              match = True  # Par défaut, considérer qu'il y a une correspondance
21
22              if numero_vol and vol["id"] != numero_vol:
23                  match = False  # False si le numéro de vol fourni ne correspond pas
24
25              if ville_depart and vol["villeDepart"].lower() != ville_depart.lower():
26                  match = False
27
28              if ville_arrivee and vol["villeArrivee"].lower() != ville_arrivee.lower():
29                  match = False
30
31              # Ajouter le vol uniquement s'il correspond à tous les critères et n'a pas déjà été vu
32              if match and vol["id"] not in vols_deja_vus:
33                  informations_vol.append(vol)
34                  vols_deja_vus.add(vol["id"])  # Ajouter l'ID du vol à l'ensemble
35
36      # Affichage des résultats
37      if informations_vol:
38          print(f"Résultats de la recherche :")
39          for vol in informations_vol:
40              print(f"Numéro de vol: {vol['id']}")
41              print("Informations sur le vol:")
42              print(json.dumps(vol, indent=4))
43      else:
44          print("Aucun vol trouvé avec les critères spécifiés.")
45
```

Figure 8 : Fonction GetVols

Explication du code de la figure 8 : Après les importations et la connexion à Redis. La fonction rechercher_vols (ligne 7) prend numero_vol, ville_depart, et ville_arrivee pour trouver les vols correspondants. Une liste informations_vol et un ensemble vols_deja_vus sont créés pour stocker les vols uniques (ligne 9).

La fonction parcourt toutes les réservations (ligne 12), convertit les données en dictionnaire (ligne 16) et applique les critères de recherche successifs (lignes 19, 21, 24 et 27) pour vérifier si le vol correspond. Si c’est le cas et que le vol est unique, il est ajouté à informations_vol (ligne 30).

À la fin, si des vols correspondent, ils sont affichés (lignes 34-39) ; sinon, un message le signale (ligne 41). La fonction obtenir_criteres_recherche (ligne 43)

permet de demander les critères et d'appeler rechercher_vols avec ces critères (ligne 53), puis est exécutée à la ligne 55.

Suite à l'exécution de ce code dans le terminal Linux, on peut constater que lorsque l'on saisit la ville de départ Metz (sans fournir de numéro de vol ni de ville d'arrivée), le programme retourne les informations concernant le vol suivant :

- Numéro de vol, Ville de départ, Ville d'arrivée, Date de départ, Heure de départ, Date d'arrivée, Heure d'arrivée

De plus, les détails sur le pilote et l'avion associés au vol sont fournis, incluant :

- Pilote : Nom, ID, Date de naissance, Ville
- Avion : Nom, ID, Capacité, Ville

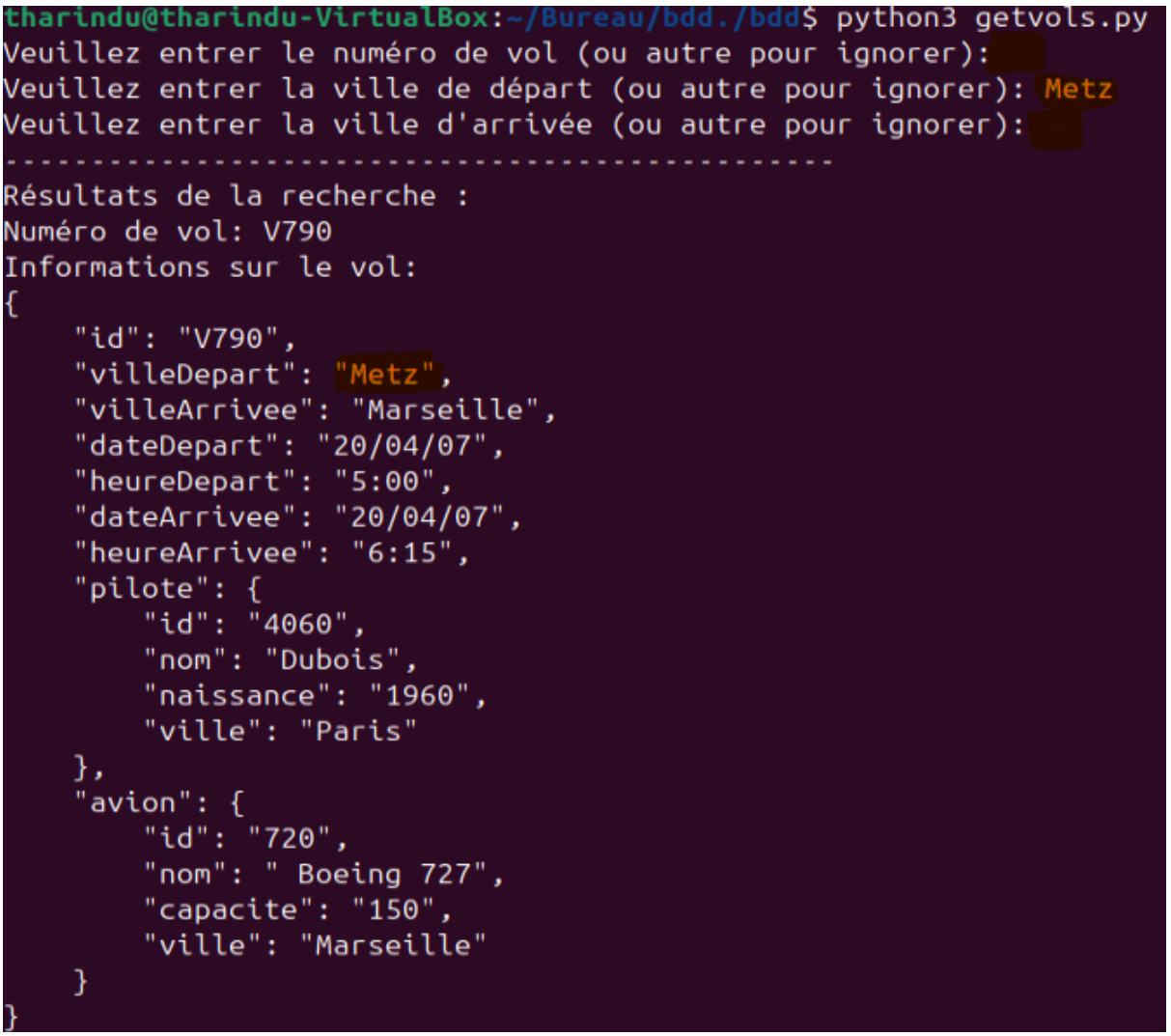


Figure 8a : execution du programme et son résultat

1.3 Comparaison structures de données : Bloom filter vs Liste

Dans le cadre de l'optimisation des tests d'appartenance à un dictionnaire, nous avons comparé deux structures de données : le Bloom filter et la Liste. L'objectif était de charger les données d'un dictionnaire de mots et de mesurer l'efficacité de ces deux structures lors de tests d'appartenance sur un grand volume de données.

Récupération et chargement des données :
Le fichier **DEM-1_1.csv** contient des couples (mot, définition), que nous avons insérés dans un Bloom filter et dans une Liste.

Test d'appartenance :
Un test a été effectué sur 100 000 mots pour comparer le temps d'appartenance dans chaque structure (Bloom filter et Liste). Cette comparaison est essentielle pour déterminer la meilleure structure à utiliser.

```python
r = redis.StrictRedis(host='localhost', port=6381, db=0)

# Nom du filtre de Bloom
filter_name = 'my_filter'

# Supprimer le filtre de Bloom s'il existe déjà
try:
    # Vérifier si le filtre de Bloom existe avec un mot fictif
    if r.execute_command('BF.EXISTS', filter_name, 'word'):
        r.execute_command('BF.DEL', filter_name)  # Supprime le filtre de Bloom s'il existe
except redis.exceptions.ResponseError:
    # Ignore l'erreur si le filtre de Bloom n'existe pas
    pass

# Création du filtre de Bloom uniquement s'il n'existe pas
try:
    r.execute_command('BF.RESERVE', filter_name, 0.01, 100000)
except redis.exceptions.ResponseError as e:
    # Gérer le cas où le filtre existe déjà
    print(f"Erreur lors de la création du filtre de Bloom : {e}")

# Liste pour stocker les mots
word_list = []

# Lecture du fichier CSV
with open('DEM-1_1.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        if row and len(row) >= 2:  # Vérifier que la ligne n'est pas vide et qu'elle a au moins 2 colonnes
            word = row[0]  # Supposons que le mot est dans la première colonne
            definition = row[1]  # Supposons que la définition est dans la deuxième colonne
            # Ajout au filtre de Bloom
            r.execute_command('BF.ADD', filter_name, word)
            # Ajout à la liste
            word_list.append(word)

# Mesurer le temps d'appartenance pour le filtre de Bloom
start_time_bloom = time.time()
for i in range(100000):
    r.execute_command('BF.EXISTS', filter_name, word_list[i % len(word_list)])
end_time_bloom = time.time()

# Mesurer le temps d'appartenance pour la liste
start_time_list = time.time()
for i in range(100000):
    word_list[i % len(word_list)] in word_list
end_time_list = time.time()

# Afficher les résultats
print(f'Temps d\'appartenance pour le Bloom filter : {end_time_bloom - start_time_bloom} secondes')
print(f'Temps d\'appartenance pour la liste : {end_time_list - start_time_list} secondes')
```

Figure 9 : Programmes de test des blooms filter

Explication du code de la figure 9 : Après les importations, une connexion à Redis est établie (ligne 6), et un filtre de Bloom est créé pour éviter les doublons (lignes 9-20). Les mots du fichier CSV sont ajoutés au filtre (ligne 31-40), puis deux tests de performance comparant la vitesse de vérification avec la fonction time de python dans le filtre de Bloom et dans une liste Python classique en (lignes 43-52).

Sur la figure 9a ci dessous les résultats des tests de performance montrent que :
Liste : elle s'est avérée plus rapide que le Bloom filter (17sec) lors des tests d'appartenance. Cette structure a donc montré de meilleures performances en termes de temps de réponse, malgré sa simplicité.
Bloom filter : bien que très efficace en termes de mémoire, il s'est avéré plus lent que la liste (21sec) pour ces tests. Nous supposons que c'est dû à l'utilisation des machines virtuelles ralentissent les capacités de l'ordinateur

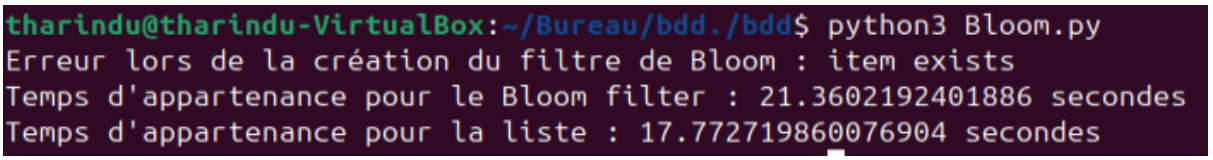


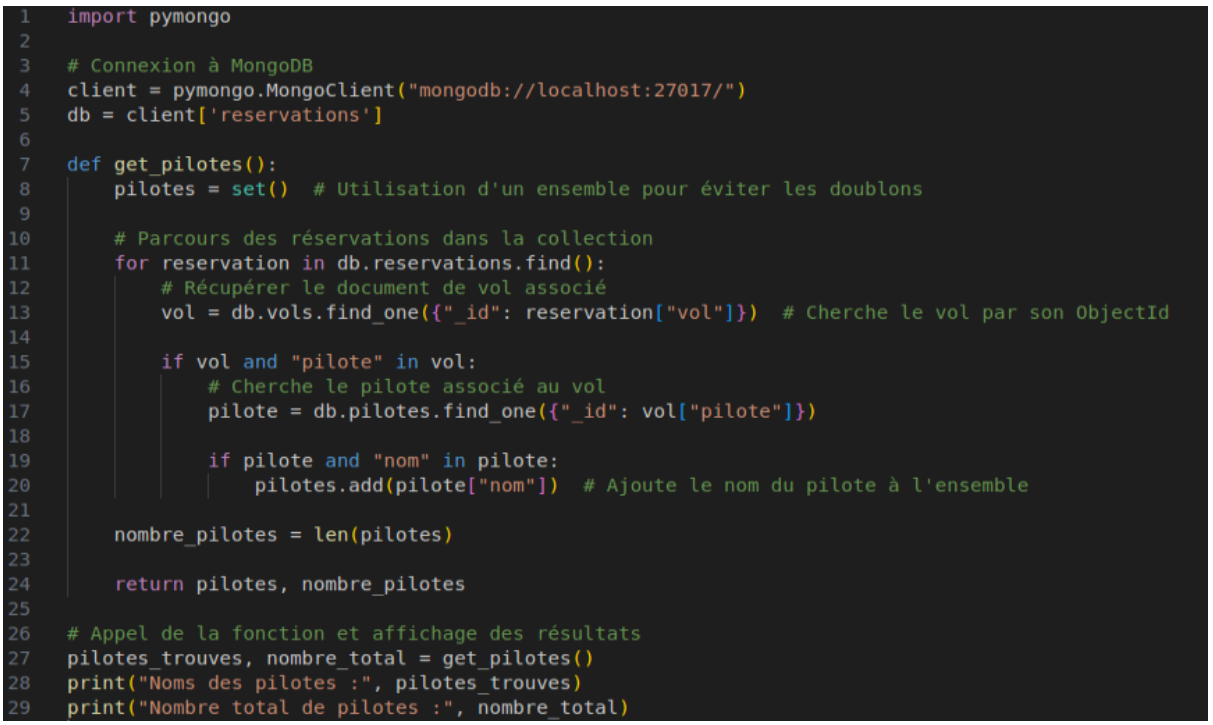
Figure 9a : exécution du programme et son résultat

2) MongoDB

Maintenant pour la partie MongoDB celle ci sera plus courte, l'objectif premier était de savoir si il était possible de faire la même chose avec Redis et Mongo.

2.1 Requêtage de la base MongoDB

Ici nous avons un code similaire à celui de Redis, le but est d'obtenir la liste des pilotes actifs dans la Base de données que nous avons faite au préalable.



```python
import pymongo

# Connexion à MongoDB
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client['reservations']

def get_pilotes():
    pilotes = set()  # Utilisation d'un ensemble pour éviter les doublons

    # Parcours des réservations dans la collection
    for reservation in db.reservations.find():
        # Récupérer le document de vol associé
        vol = db.vols.find_one({"_id": reservation["vol"]})  # Cherche le vol par son ObjectId

        if vol and "pilote" in vol:
            # Cherche le pilote associé au vol
            pilote = db.pilotes.find_one({"_id": vol["pilote"]})

            if pilote and "nom" in pilote:
                pilotes.add(pilote["nom"])  # Ajoute le nom du pilote à l'ensemble

    nombre_pilotes = len(pilotes)

    return pilotes, nombre_pilotes

# Appel de la fonction et affichage des résultats
pilotes_trouves, nombre_total = get_pilotes()
print("Noms des pilotes :", pilotes_trouves)
print("Nombre total de pilotes :", nombre_total)
```

Figure 10 :Programme Pilotes (MongoDB)

Explication du code de la figure 10 : Ce script utilise MongoDB pour récupérer les noms des pilotes. Une connexion est établie (ligne 4), et la fonction get_pilotes (ligne 7) parcourt chaque document de la collection reservations pour trouver les pilotes associés et les stocke dans un ensemble (lignes 10-20. Les noms et le nombre de pilotes sont affichés (lignes 25-26).

2.2 Opération de Jointures

A l'inverse de Redis, Mongo lui prends en charge les jointures par défaut quand nous sommes connectés dans la base de données ce qui rends donc les opérations beaucoup plus faciles

La jointure se note sous la forme suivante :

```
db.collection_principale.aggregate([
        {
        $lookup: {
        from: "collection_secondaire",  // Nom de la collection que nous allons joindre
        localField: "champ_local",        // Champ de la collection principale utilisé pour faire la correspondance
```

foreignField: "champ_etranger", // _id
as: "nom_du_resultat" // Résultat de la jointure sera stocké dans ce champ
}
}
])

Maintenant dans la figure 11 nous allons donner un exemple d'une jointure qui fait le lien entre vols et avions afin de récupérer les informations présents dans la collections avions grâce à l'id commun

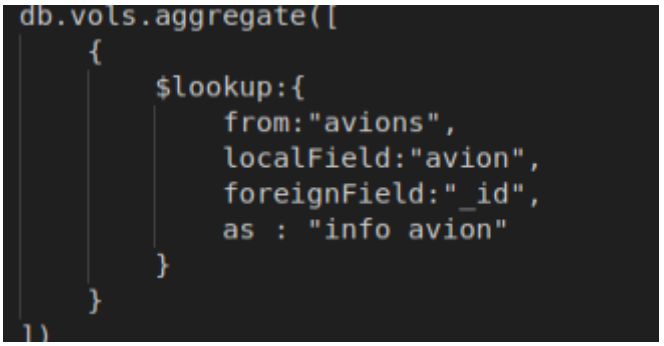


Figure 11 : Jointures entre vol et avions

Explication de la jointure de la figure 11 : Ce script utilise l'agrégation pour lier les collections vols et avions en MongoDB. On définit la jointure $lookup pour enrichir chaque document vol avec les détails de l'avion associé (lignes 1-7).

Et le résultat dans la figure 11a, comme prévu est le suivant, nous récupérons bien toutes les informations de l'avion présent dans la table vol.

Figure 11a : Exécution du code de jointure

IV. Discussion

Résumé des résultats principaux

Redis s'est révélé performant pour les opérations simples grâce à son architecture clé-valeur, facilitant une gestion rapide des données via des fonctions Python dédiées. Sa structure orientée sur les réservations centralise efficacement les informations, ce qui est avantageux pour les requêtes directes. MongoDB, de son côté, a montré un avantage notable pour les requêtes complexes nécessitant des jointures, notamment grâce à sa prise en charge native des opérations d’agrégation.

Résultats inattendus

Redis a présenté des défis pour les opérations de jointures simulées, produisant parfois des résultats incohérents. Cela a nécessité le développement de fonctions spécifiques (GETCLIENT, GETPILOTES, GETVOLS) pour contourner cette limitation. De plus, la Liste a surperformé le Bloom filter lors des tests d'appartenance, ce qui peut être dû à l’impact des machines virtuelles utilisées sur les performances de ce dernier.

Limites et pistes d’amélioration

Les limites de Redis pour les jointures et la simulation des tests dans un environnement de développement pourraient influencer les résultats. Pour affiner l’évaluation, il serait intéressant de tester ces solutions en conditions proches de la production, avec des volumes de données plus importants, et d'envisager d'autres options NoSQL pour compléter la comparaison.

V. Conclusion

Dans cette étude, nous avons comparé les performances de Redis et MongoDB pour une application de gestion de réservations aériennes, en examinant l’efficacité de chaque base de données pour optimiser les requêtes.

Redis, basé sur une architecture clé-valeur, offre une grande rapidité d’accès pour des opérations simples. Cependant, ses limitations sont significatives : toutes les données sont stockées en RAM, ce qui signifie qu’elles disparaissent dès que la machine est éteinte, rendant Redis moins fiable pour des applications nécessitant une conservation durable des données. De plus, l’absence de support natif pour les jointures complique son usage dans des contextes où les relations entre données sont essentielles.

En comparaison, MongoDB stocke les données directement sur disque, garantissant leur persistance même après un redémarrage du système. Sa prise en charge des jointures en fait également un choix bien plus flexible et adapté aux applications nécessitant des relations complexes, comme la gestion des réservations aériennes.

Ainsi, pour une application de gestion de réservations nécessitant des jointures et une durabilité des données, MongoDB est le choix le plus pertinent. Redis, malgré sa rapidité, se révèle limité pour des usages où la persistance des données est cruciale.

À l'avenir, une solution hybride, donc utilisée redis **ET** mongoDB pourrait également être envisagée pour tirer parti des points forts de chaque technologie en fonction des besoins spécifiques en matière de performance et de stockage.