Mise en œuvre d'une solution de gestion de données avec Redis et MongoDB

Réalisé par Lina MEDANI

Le 04/11/2024

Table des Matières

1- Introduction :

Notre projet reposait sur Redis, un système NoSQL ultra-rapide, avec MongoDB pour sa flexibilité de requêtes. Notre équipe de développeurs a été mandatée par une compagnie aérienne ambitieuse pour gérer des informations complexes sur ses vols, pilotes, avions et réservations. Les objectifs étaient clairs : garantir des requêtes aussi rapides que possible avec Redis, structurer les données de manière optimale avec MongoDB, et rester flexibles pour répondre aux besoins évolutifs du projet.

Pour commencer, nous avons analysé plusieurs fichiers texte, chacun contenant des informations essentielles mais dispersées sur les avions, clients, pilotes et vols. Afin de rendre ces données exploitables, nous avons choisi de les convertir en JSON. Nous avons d'abord établi une table de correspondance, associant chaque fichier à ses colonnes spécifiques, qui a servi de guide pour transformer le contenu brut en dictionnaires JSON bien structurés, prêts à être stockés dans Redis et MongoDB.

2- Matériels :

Le matériel utilisant est le suivant :

2.1- Environnement Python dans Visual Studio Code

Le développement du projet s'est effectué dans Visual Studio Code, un éditeur de code puissant. L'intégration de Python a été facilitée par l'extension officielle, permettant une gestion des dépendances via **pip** pour installer les bibliothèques nécessaires comme **`redis`** et **`pymongo`**.

2.2- Installation de Redis sous Windows avec WSL

Redis a été installé en utilisant le Windows Subsystem for Linux (WSL), offrant un environnement Linux sur Windows. Les étapes incluent :

- Activation de WSL : Commande « **wsl –install** » dans PowerShell.
- Installation d'une distribution Linux : Ubuntu via le Microsoft Store.
- Installation de Redis : Mise à jour des paquets avec « **sudo apt update »**, puis installation avec « **sudo apt install redis-server »**. On démarre le serveur avec « **sudo service redis-server start »** et vérifie son bon fonctionnement avec « **redis-cli ping »**, qui doit renvoyer PONG**.**

2.3- MongoDB via MongoDB Atlas

MongoDB a été déployé via MongoDB Atlas, une plateforme cloud offrant une base de données MongoDB gérée. Cela permet d'accéder à MongoDB sans avoir à gérer une installation locale. Les étapes incluent :

- **Création d'un compte MongoDB Atlas** : Inscription sur le site de MongoDB Atlas pour créer un cluster.

- **Configuration du cluster** : Choix d'une région, d'un niveau de service, et création d'une base de données.
- **Connexion à MongoDB Atlas** : Utilisation d'un URI de connexion fourni par Atlas, qui permet d'accéder à la base de données depuis l'application Python. Cet URI contient des informations essentielles, comme le nom d'utilisateur, le mot de passe, et l'adresse du cluster MongoDB. Par exemple, l'option retryWrites=true dans l'URI garantit une tentative de réécriture automatique en cas d'échec, assurant ainsi plus de fiabilité et de sécurité dans la transmission des données.

2.4- Bibliothèques Python

Plusieurs bibliothèques ont été installées pour faciliter les interactions avec Redis et MongoDB :

- **redis-py** : Client Python pour interagir avec Redis, installé via **pip install redis**.
- **pymongo** : Client pour MongoDB, installé via **pip install pymongo**.
- **json** : Bibliothèque standard pour manipuler des données JSON, permettant de convertir entre objets Python et chaînes JSON.

3- Méthodes

Les méthodes utilisées sont les suivantes :

3.1- Chargement et structuration des données :

Pour traiter efficacement les données, plusieurs fichiers .txt ont été chargés, notamment :

- VOLS.txt, CLIENTS.txt, RESERVATIONS.txt, PILOTES.txt, AVIONS.txt et DEFCLASSES.txt.
- **Création d'un dictionnaire de correspondance :** Un dictionnaire **table_correspondance** a été établi pour mapper chaque fichier aux colonnes attendues, garantissant un traitement homogène des données.
- **Lecture et découpage des fichiers :** Chaque fichier a été parcouru, et les enregistrements ont été découpés

3.2- Construction du JSON :

Les données ont été organisées de manière dénormalisée. Chaque vol est représenté par une clé unique dans le dictionnaire principal, contenant des informations complètes sur :

- Les vols
- Les clients
- Les réservations
- Les détails de l'avion

Cette structure facilite l'accès rapide et complet aux informations liées à chaque vol.

4- Résultats

Les résultats obtenus à la suite de ce projet sont les suivants :

4.1- Stockage des données :

Les données ont été stockées à la fois dans Redis et MongoDB :

4.1.1-Redis : Une connexion a été établie à Redis, et chaque vol a été stocké sous forme de chaîne JSON avec une clé unique au format **vol:<NumVol>**. Les données ont été chargées depuis le fichier vols_final.json, et chaque vol a été inséré dans Redis avec un message de confirmation à la fin.

```
1  import redis
2  import json
3  import time
4
5  # Charger json_final depuis un fichier JSON
6  with open('vols_final.json', 'r') as json_file:
7      json_final = json.load(json_file)
8
9  # Connexion à Redis
10 r = redis.StrictRedis(host='localhost', port=6379, db=0, decode_responses=True)
11 # Mesurer le temps d'insertion
12 start_time = time.time()
13 # Stocker chaque vol dans Redis
14 for vol_id, vol_data in json_final.items():
15     r.set(f"vol:{vol_id}", json.dumps(vol_data))
16 insertion_time = time.time() - start_time
17 print(f"Temps d'insertion dans Redis : {insertion_time:.4f} secondes")
18
19 # Mesurer le temps de lecture
20 start_time = time.time()
21 for vol_id in json_final.keys():
22     r.get(f"vol:{vol_id}")
23 reading_time = time.time() - start_time
24 print(f"Temps de lecture dans Redis : {reading_time:.4f} secondes")
```

Figure1 : « Stockage des données dan Redis »

Le code présenté dans la figure1 a pour but de **stocker des données de vols** dans une base de données Redis, puis de mesurer le temps nécessaire pour **insertion** et **lecture** de ces données.

- **Lignes 1 à 3** : Importation des bibliothèques.

Le code importe les bibliothèques redis pour interagir avec Redis, json pour manipuler les données JSON, et time pour mesurer les durées d'exécution.

- **Lignes 6 à 7** : Chargement des données JSON.

Le fichier vols_final.json est ouvert en lecture, et son contenu est chargé dans la variable json_final sous forme de dictionnaire Python.

- **Ligne 10** : Connexion à la base de données Redis.

Une connexion est établie à une instance Redis locale, en utilisant le port 6379.

- **Ligne 12**: Démarrage du chronomètre pour mesurer le temps d'insertion.

Le temps actuel est enregistré dans start_time afin de mesurer la durée d'insertion.

- **Lignes 14 à 15** : Insertion des données dans Redis.

Pour chaque vol dans json_final, une clé unique est générée sous le format vol:<ID>, et les informations du vol sont converties en chaîne JSON et stockées dans Redis.

- **Lignes 16 à 17**: Calcul et affichage du temps d'insertion.

Le temps écoulé pour l'insertion est calculé en soustrayant le temps initial (start_time), et le résultat est affiché avec une précision de quatre décimales.

- **Ligne 20** : Démarrage du chronomètre pour mesurer le temps de lecture.

Le chronomètre est réinitialisé pour mesurer la durée de la lecture des données depuis Redis.

- **Lignes 21 à 22** : Lecture des données depuis Redis.

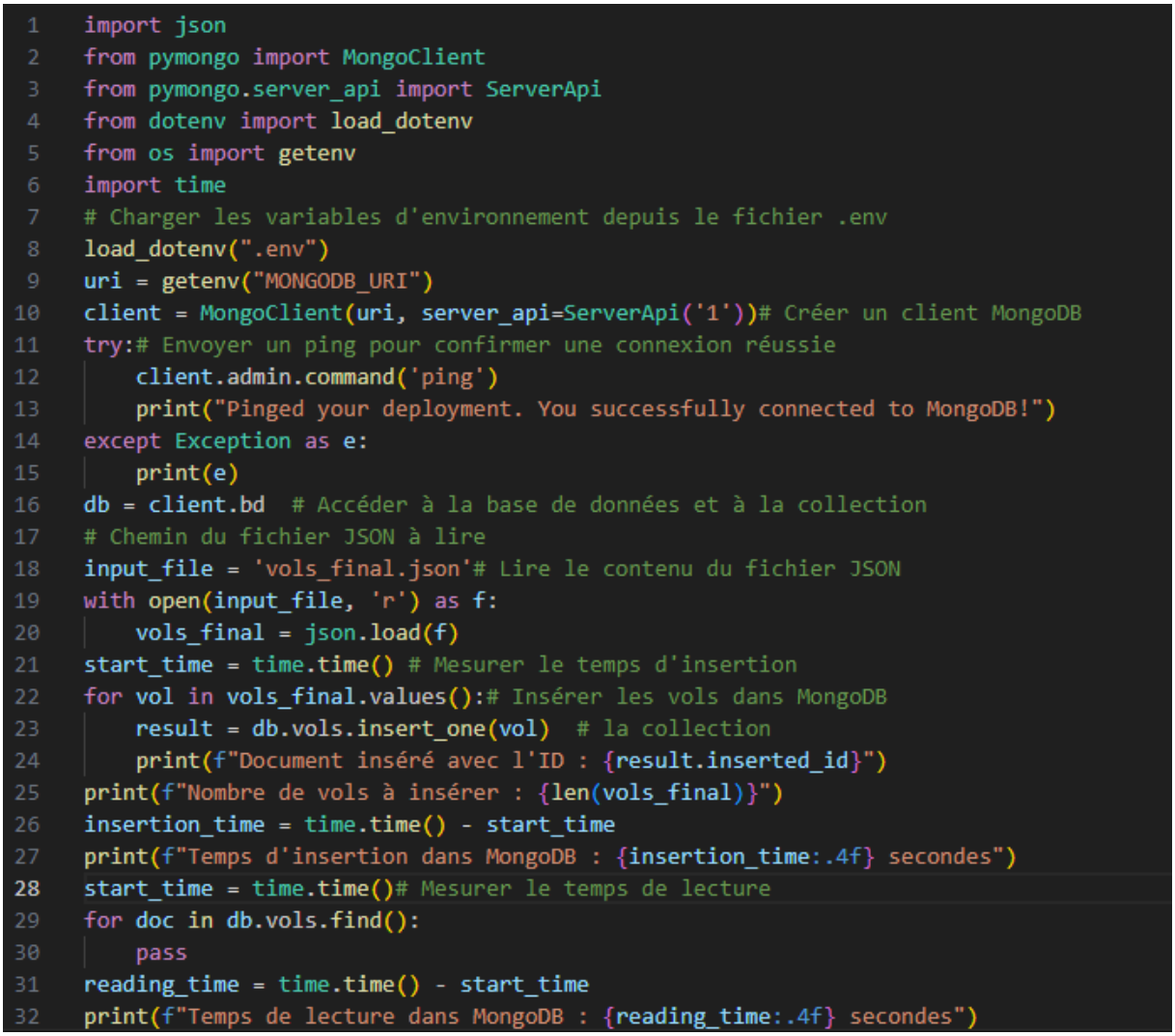
Pour chaque vol, les informations sont récupérées depuis Redis en utilisant la clé unique correspondante.

- **Lignes 23 à 24** : Calcul et affichage du temps de lecture.

Le temps écoulé pour la lecture est calculé et affiché, ce qui montre la durée totale nécessaire pour lire toutes les données stockées

4.1.2- MongoDB :

Une connexion a été établie avec MongoDB via un client Python, et les données ont été lues depuis un fichier JSON. Les informations ont été insérées dans des collections permettant ainsi d'effectuer des requêtes robustes et flexibles.



```python
import json
from pymongo import MongoClient
from pymongo.server_api import ServerApi
from dotenv import load_dotenv
from os import getenv
import time
# Charger les variables d'environnement depuis le fichier .env
load_dotenv(".env")
uri = getenv("MONGODB_URI")
client = MongoClient(uri, server_api=ServerApi('1'))# Créer un client MongoDB
try:# Envoyer un ping pour confirmer une connexion réussie
    client.admin.command('ping')
    print("Pinged your deployment. You successfully connected to MongoDB!")
except Exception as e:
    print(e)
db = client.bd  # Accéder à la base de données et à la collection
# Chemin du fichier JSON à lire
input_file = 'vols_final.json'# Lire le contenu du fichier JSON
with open(input_file, 'r') as f:
    vols_final = json.load(f)
start_time = time.time() # Mesurer le temps d'insertion
for vol in vols_final.values():# Insérer les vols dans MongoDB
    result = db.vols.insert_one(vol)  # la collection
    print(f"Document inséré avec l'ID : {result.inserted_id}")
print(f"Nombre de vols à insérer : {len(vols_final)}")
insertion_time = time.time() - start_time
print(f"Temps d'insertion dans MongoDB : {insertion_time:.4f} secondes")
start_time = time.time()# Mesurer le temps de lecture
for doc in db.vols.find():
    pass
reading_time = time.time() - start_time
print(f"Temps de lecture dans MongoDB : {reading_time:.4f} secondes")
```

Figure2 : « Stockage de données dans MongoDB »

Le code de la **Figure 2** vise à insérer des données de vols dans une base de données MongoDB et à mesurer le temps d'exécution des opérations d'insertion et de lecture de ces données. D'abord, les bibliothèques sont importées : json pour gérer les données JSON, pymongo pour interagir avec MongoDB, dotenv pour charger les variables d'environnement de manière sécurisée, et time pour mesurer les durées d'exécution **(Lignes 1 à 6).** Ensuite, le fichier .env est chargé pour accéder aux variables d'environnement **(ligne 8),** en particulier l'URI de MongoDB, récupéré à la **ligne 9**. La connexion au client MongoDB est établie à la **ligne 10** avec cet URI et en précisant la version de l'API MongoDB avec ServerApi('1'). Une commande de ping est ensuite exécutée pour vérifier la connexion au serveur ; en cas de succès, un message confirme la connexion, sinon une erreur est imprimée **(lignes 11 à 15).** La base de données MongoDB, appelée bd, est ciblée pour les opérations de stockage de vols **(ligne 16).** Les données de vols sont ensuite chargées depuis le fichier vols_final.json, puis converties en dictionnaire Python dans la variable vols_final, prêtes à être insérées dans MongoDB **(lignes 18 à 20).**

Pour mesurer le temps d'insertion, un chronomètre est lancé à la **ligne 21**. **Les lignes 22 à 24** insèrent chaque vol dans la collection vols de bd, et l'ID du document inséré est affiché pour chaque vol inséré. Le nombre total de vols est ensuite imprimé à **la ligne 25**. Une fois tous les vols insérés, le temps total d'insertion est calculé et affiché en secondes, offrant une

mesure précise du temps d'insertion pour évaluer les performances de MongoDB **(lignes 26 et 27)**. Un nouveau chronomètre est lancé à **la ligne 28** pour mesurer le temps de lecture des données. Chaque document est ensuite parcouru dans la collection vols, sans action supplémentaire, permettant de mesurer le temps d'accès aux données (**lignes 29 et 30**). Enfin, le temps de lecture est calculé et affiché en secondes, fournissant une estimation de la rapidité de lecture des données **(lignes 31 et 32).**

4.2- Fonctions de Requête :

Cette section présente les fonctions de requête spécifiques mises en œuvre pour répondre à des besoins opérationnels. Nous illustrerons l'utilisation de Redis et MongoDB pour ces fonctions.

3.4.1. Lister toutes les villes d'arrivée pour une ville de départ donnée :

Cette fonction permet d'obtenir toutes les villes d'arrivée à partir d'une ville de départ spécifiée. Les exemples suivants montrent comment réaliser cette requête à la fois en utilisant Redis et MongoDB.

Code Redis :

```
15  # Fonction pour lister les villes d'arrivée
16  def lister_villes_arrivee(r, ville_depart):
17      villes_arrivee = set()
18      keys = r.keys("vol:*")  # Récupérer toutes les clés de vols
19      for key in keys:
20          vol_data = json.loads(r.get(key))  # Récupérer les données de chaque vol
21          if vol_data.get("VilleD") == ville_depart:  # Vérifier la ville de départ
22              villes_arrivee.add(vol_data.get("VilleA"))  # Ajouter la ville d'arrivée
23      return list(villes_arrivee)
```

Figure 3 : « Lister les villes avec Redis »

La figure3 présente la fonction lister_villes_arrivee qui commence par prendre deux paramètres : le client Redis (r) et la ville de départ (ville_depart) à la **ligne 16**. Ensuite, à **la ligne 17**, un ensemble vide villes_arrivee est créé pour stocker les villes d'arrivée sans doublons. À **la ligne 18**, la commande r.keys("vol:*") est utilisée pour récupérer toutes les clés des vols stockés dans Redis. À **la ligne 19**, une boucle for est lancée pour parcourir chaque clé de vol récupérée. À **la ligne 20**, les données de chaque vol sont extraites avec r.get(key) et converties en dictionnaire à l'aide de json.loads(). Puis, à **la ligne 21**, la fonction vérifie si la ville de départ ("VilleD") correspond à celle passée en paramètre (ville_depart). Si cette condition est vraie, à **la ligne 22**, la ville d'arrivée ("VilleA") est ajoutée à l'ensemble villes_arrivee. Enfin, à **la ligne 23**, l'ensemble villes_arrivee est converti en liste et retourné, garantissant qu'il ne contient pas de doublons.

Le résultat de cette fonction, en prenant « Paris » comme ville d'arrivée avec Redis est le suivant :

```
Villes d'arrivée depuis Paris : ['Pekin', 'Nice', 'Marseille']
```

Code MongoDB :

```
24  def get_arrival_ville(departure_ville):
25      try:
26          villes = db.vols.distinct("VilleA", {"VilleD": departure_ville})
27          if villes:
28              return villes
29          else:
30              return f"Aucune ville d'arrivée trouvée pour la ville de départ : {departure_ville}"
31      except Exception as e:
32          return f"Une erreur s'est produite : {e}"
```

Figure4 : « Lister les villes avec Mongo »

La **figure4** présente la fonction get_arrival_ville prend un paramètre departure_ville et a pour but de récupérer la liste des villes d'arrivée associées à une ville de départ donnée depuis une base de données MongoDB. Voici ce que chaque ligne fait :

- **Ligne 25** : La fonction utilise un bloc try pour tenter d'exécuter le code et gérer les éventuelles exceptions qui pourraient survenir.
- **Ligne 26** : La méthode db.vols.distinct("VilleA", {"VilleD": departure_ville}) est utilisée pour interroger la collection vols de la base de données MongoDB. Elle récupère les valeurs distinctes des villes d'arrivée ("VilleA") pour les documents où la ville de départ ("VilleD") correspond à la ville spécifiée dans departure_ville.
- **Ligne 27-30** : Si des villes d'arrivée sont trouvées, elles sont renvoyées sous forme de liste. Si aucune ville d'arrivée n'est trouvée, un message indiquant "Aucune ville d'arrivée trouvée pour la ville de départ" est retourné.
- **Ligne 31-32** : Si une erreur se produit pendant l'exécution de la requête, l'exception est capturée et un message d'erreur est retourné, indiquant la nature de l'exception.

Le résultat de cette fonction, en prenant « Marseille » comme ville d'arrivée avec MongDB est le suivant :

```
['Amsterdam', 'NewYork', 'Paris', 'Pekin']
```

4.2. Compter le nombre de pilotes ayant une lettre spécifique dans leur nom

Cette fonction permet de compter le nombre de pilotes dont le nom contient une lettre donnée. Voici comment cette fonction est implémentée en utilisant Redis et MongoDB.

Code Redis :

```
25  # Fonction pour compter les pilotes par lettre
26  def compter_pilotes_par_lettre(r, lettre):
27      pilotes_uniques = set()  # Ensemble pour stocker les pilotes uniques
28      keys = r.keys("vol:*")  # Récupérer toutes les clés de vols
29      for key in keys:
30          vol_data = json.loads(r.get(key))  # Récupérer les données de chaque vol
31          pilote = vol_data.get("Pilote", {})  # Récupérer les informations du pilote
32          if pilote and lettre.lower() in pilote["NomPil"].lower():  # Vérifier la présence de la lettre
33              pilotes_uniques.add(pilote["NomPil"])  # Ajouter le nom du pilote à l'ensemble
34      return len(pilotes_uniques)  # Retourner le nombre de pilotes uniques
```

Figure5 : « Compter les pilotes avec Redis »

La fonction compter_pilotes_par_lettre présentée dans la **figure5** commence par prendre deux paramètres : r(le client Redis) et lettre (la lettre à rechercher dans les noms des pilotes) comme indiqué à **la ligne 26**. À **la ligne 27**, un ensemble vide pilotes_uniques est créé pour stocker les noms de pilotes de manière unique. La commande r.keys("vol:*") est utilisée à **la ligne 28** pour récupérer toutes les clés des vols stockés dans Redis. À **la ligne 29**, une boucle for parcourt chacune de ces clés. À **la ligne 30**, les données du vol sont extraites via r.get(key) et converties en dictionnaire avec json.loads(). La fonction cherche ensuite les informations du pilote dans **la ligne 31** en accédant à la clé "Pilote" du dictionnaire vol_data. Si ces informations existent, la condition à **la ligne 32** vérifie si la lettre recherchée est présente dans le nom du pilote, indépendamment de la casse. Si la condition est remplie, **la ligne 33** ajoute le nom du pilote à l'ensemble pilotes_uniques. Enfin, à **la ligne 34**, la fonction retourne la taille de l'ensemble pilotes_uniques avec len(), ce qui donne le nombre de pilotes uniques dont le nom contient la lettre spécifiée.

Le résultat de cette fonction, en utilisant la lettre « a », est le suivant :

```
Nombre de pilotes uniques contenant la lettre a : 7
```

Code MongoDB :

```python
def compter_pilotes_par_lettre_mongo(lettre):
    try:
        # Utiliser l'agrégation pour extraire les noms des pilotes
        pipeline = [
            {
                "$unwind": "$Pilote"  # Déplier les documents liés au pilote
            },
            {
                "$match": {
                    "Pilote.NomPil": {"$regex": lettre, "$options": "i"}
                    # Rechercher par lettre dans le nom du pilote
                }
            },
            {
                "$group": {
                    "_id": "$Pilote.NomPil"  # Regrouper par nom de pilote
                }
            }
        ]

        # Exécuter le pipeline d'agrégation
        resultats = db.vols.aggregate(pipeline)
        # Compter le nombre de pilotes uniques
        count = sum(1 for _ in resultats)
        return count
    except Exception as e:
        return f"Une erreur s'est produite : {e}"
```

Figure6 : « Compter les pilotes avec MongoDB »

La fonction compter_pilotes_par_lettre_mongo présentée dans **la Figure6** est conçue pour compter les pilotes uniques dont le nom contient une lettre spécifique, en utilisant un pipeline d'agrégation MongoDB. À **la ligne 35**, la fonction utilise un bloc try pour gérer les exceptions potentielles. À **la ligne 37**, un pipeline d'agrégation est défini, et la première étape de ce pipeline, à **la ligne 39**, est un opérateur $unwind appliqué à "$Pilote" pour "déplier" les documents et permettre de traiter chaque pilote individuellement dans MongoDB. À **la ligne 42**, l'étape $match utilise l'expression régulière "$regex" pour rechercher la lettre spécifiée dans le champ "Pilote.NomPil", avec l'option "i" pour rendre la recherche insensible à la casse. Ensuite, à **la ligne 48**, l'opérateur $group regroupe les documents par nom de pilote ("_id": "$Pilote.NomPil"), garantissant que chaque pilote unique est compté une seule fois.

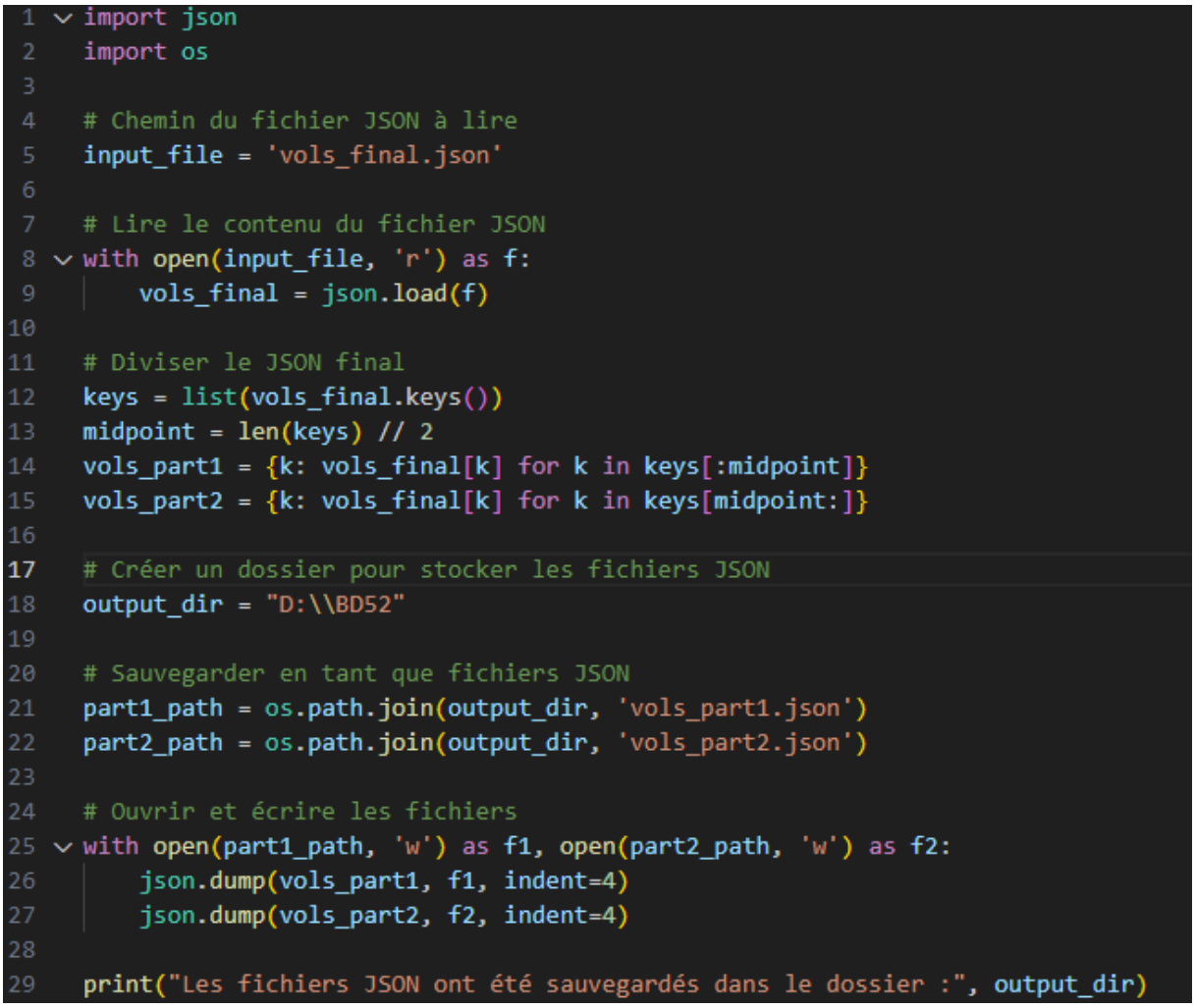
Le pipeline est exécuté à **la ligne 55** avec db.vols.aggregate(pipeline), ce qui applique les étapes définies pour filtrer et regrouper les pilotes. Enfin, le nombre total de pilotes uniques correspondant est renvoyé. En cas d'exception, un message d'erreur personnalisé est retourné à **la ligne 60**, indiquant l'erreur rencontrée.

Le résultat de cette fonction, en utilisant la lettre « a », est le suivant :

```
Nombre de pilotes uniques contenant la lettre a : 7
```

3.3- Partitionnement des Données :

Les données JSON ont été divisées en deux fichiers distincts, chacun contenant la moitié des vols. Cela a facilité la gestion des données et la distribution des tâches.



```python
import json
import os

# Chemin du fichier JSON à lire
input_file = 'vols_final.json'

# Lire le contenu du fichier JSON
with open(input_file, 'r') as f:
    vols_final = json.load(f)

# Diviser le JSON final
keys = list(vols_final.keys())
midpoint = len(keys) // 2
vols_part1 = {k: vols_final[k] for k in keys[:midpoint]}
vols_part2 = {k: vols_final[k] for k in keys[midpoint:]}

# Créer un dossier pour stocker les fichiers JSON
output_dir = "D:\\BD52"

# Sauvegarder en tant que fichiers JSON
part1_path = os.path.join(output_dir, 'vols_part1.json')
part2_path = os.path.join(output_dir, 'vols_part2.json')

# Ouvrir et écrire les fichiers
with open(part1_path, 'w') as f1, open(part2_path, 'w') as f2:
    json.dump(vols_part1, f1, indent=4)
    json.dump(vols_part2, f2, indent=4)

print("Les fichiers JSON ont été sauvegardés dans le dossier :", output_dir)
```

Figure7 : « Partitionnement des données »

Le code de la **Figure7** commence par importer les bibliothèques json et os aux **lignes 1 et 2**. La première sera utilisée pour manipuler des données JSON, et la seconde pour gérer les chemins de fichiers et dossiers. À **la ligne 5**, le chemin du fichier JSON source vols_final.json est défini. Ensuite, **la ligne 8** ouvre ce fichier en mode lecture ('r'), charge son contenu en mémoire en utilisant json.load(f), et stocke les données dans le dictionnaire vols_final.

Pour diviser ce dictionnaire en deux parties, le code commence par créer une liste des clés (keys) à **la ligne 11** et calcule le point médian (midpoint) à **la ligne 12**. Ensuite, à **la ligne 13**, un nouveau dictionnaire vols_part1 est créé pour contenir la première moitié des données, et **à la ligne 14**, vols_part2 est créé pour stocker la seconde moitié.

Le chemin du dossier output_dir où seront sauvegardés les fichiers JSON est défini à **la ligne 17**. Les chemins complets des deux fichiers de sortie (vols_part1.json et vols_part2.json) sont ensuite générés avec os.path.join aux **lignes 20 et 21**.

Enfin, à **la ligne 24**, les deux fichiers sont ouverts en mode écriture ('w'). Les données sont ensuite écrites dans chaque fichier via json.dump, avec indent=4 pour formater le JSON de manière lisible. Une fois les données enregistrées, un message de confirmation est affiché à

la ligne 27, précisant que les fichiers JSON ont bien été sauvegardés dans le dossier spécifié (output_dir).

3.4- Fonction de jointure :

Une fonction de jointure a été développée pour réaliser des jointures entre deux dictionnaires. Cette fonction utilise un attribut commun (dans ce cas, le numéro de vol) pour consolider les données issues de différentes sources, facilitant ainsi l'intégration des informations dans une seule vue cohérente.

```
14  # Jointure et sauvegarde des résultats
15  def jointure(d1, d2, att):
16      resultats_jointure = []  # Liste pour stocker les résultats de jointure
17      for i in d1.keys():
18          for j in d2.keys():
19              if d1[i].get(att) == d2[j].get(att):
20                  resultat = {
21                      'vol_part1': d1[i],
22                      'vol_part2': d2[j],
23                  }
24                  resultats_jointure.append(resultat)
25      return resultats_jointure
```

Figure8 : « Fonction de jointure »

La fonction jointure présentée dans la **Figure8** permet de réaliser une jointure entre deux dictionnaires de données (d1 et d2) en comparant les valeurs d'un attribut commun spécifié par l'argument att. D'abord, à la **ligne 16**, elle initialise une liste vide resultats_jointure qui sera utilisée pour stocker les résultats de la jointure. Ensuite, dans les **lignes 17 et 18**, deux boucles imbriquées parcourent toutes les clés de d1 et d2. Pour chaque combinaison de clés i et j, la fonction compare, à **la ligne 19**, les valeurs de l'attribut att dans les données associées aux clés correspondantes dans d1[i] et d2[j]. Si les valeurs sont égales, cela indique une correspondance entre les deux données. À **la ligne 20**, un dictionnaire resultat est créé, contenant les données de chaque vol correspondant de d1 et d2, respectivement sous les clés 'vol_part1' et 'vol_part2'. Ce dictionnaire est ensuite ajouté à la liste resultats_jointure à **la ligne 24**. Enfin, après avoir parcouru toutes les combinaisons de clés, la fonction retourne, à **la ligne 25**, la liste resultats_jointure qui contient toutes les paires de vols ayant des valeurs égales pour l'attribut att, permettant ainsi de récupérer les résultats de la jointure entre les deux ensembles de données.

5- Discussion

Le projet visait à démontrer une double implémentation pour gérer les données structurées de vols en utilisant Redis et MongoDB. Chaque base de données apporte des atouts uniques, augmentant ainsi la flexibilité et l'efficacité de la solution.

Redis

Le stockage en mémoire de Redis permet un accès ultra-rapide aux données de vol individuelles, ce qui est idéal pour des applications nécessitant des lectures fréquentes. Grâce à une fonction de mesure des performances, nous avons observé un temps d'insertion des vols dans Redis de **2,3025 secondes** et un temps de lecture des vols de **0,1415 secondes**. Redis utilise une structure de stockage clé-valeur, permettant de récupérer directement les informations d'un vol via son identifiant (par exemple, vol). Cette rapidité en fait un choix plus adapté pour les accès directs, mais il est moins adapté aux requêtes complexes impliquant des relations entre collections, ce qui justifie l'utilisation complémentaire de MongoDB pour de telles tâches.

MongoDB

MongoDB, grâce à sa structure NoSQL, propose un modèle de données flexible et évolutif, capable de gérer des relations entre collections sans schéma rigide. Les fonctions de mesure de temps qui ont été mises en place montrent un temps d'insertion des vols de **10,8120 secondes** et un temps de lecture des vols de **0,2288 secondes** pour MongoDB. Bien qu'un peu plus lent en insertion, MongoDB offre de solides capacités de requête pour des opérations complexes, telles que lister les villes d'arrivée pour une ville de départ donnée ou compter les pilotes ayant une lettre spécifique dans leur nom. Ce type de requêtes est facilité par MongoDB, qui est plus performant pour les opérations de jointure et les agrégations de données.

Stratégie de gestion des données

Les deux bases de données prennent en charge le partitionnement des données, ce qui a permis de diviser la structure JSON en deux parties, facilitant ainsi la gestion des ensembles de données volumineux. Cette approche double (Redis et MongoDB) offre de la flexibilité, Redis étant adapté pour les lectures rapides, et MongoDB pour les opérations plus complexes.

Conclusion

La mise en œuvre de la solution de gestion des données de vols via Redis et MongoDB a mis en évidence les avantages distincts de chaque technologie. Redis, avec son stockage en mémoire, a démontré sa capacité à offrir un accès rapide aux données, ce qui est essentiel pour les opérations en temps réel. D'un autre côté, MongoDB a montré sa force en termes de flexibilité et d'évolutivité, permettant des requêtes complexes et une gestion des relations entre différentes entités de données.

Les temps mesurés pour l'insertion et la lecture des données révèlent une nette efficacité, avec Redis présentant des performances supérieures dans les opérations de lecture et d'insertion. Cela souligne l'importance de choisir les bons outils en fonction des exigences spécifiques des tâches de gestion des données.

En somme, cette étude montre que l'intégration de différentes technologies peut grandement améliorer la gestion des données. Les résultats obtenus soulignent l'importance d'une approche bien pensée dans le choix des bases de données pour répondre

efficacement aux besoins de l'organisation, garantissant ainsi une meilleure réactivité et une gestion optimisée des informations de vols.

Pour aller plus loin, il serait intéressant d'explorer l'intégration d'outils d'analyse en temps réel, comme des systèmes de streaming de données (par exemple, Apache Kafka), qui permettraient de traiter et d'analyser les flux de données en direct. Cette approche permettrait non seulement de renforcer les capacités actuelles, mais aussi d'anticiper les besoins futurs de l'organisation en matière de données. Une telle infrastructure évolutive offrirait une base solide pour le développement de nouveaux services, comme la prédiction de la demande de vols ou la gestion proactive des ressources, afin de répondre aux enjeux d'un secteur en constante évolution.

Références

1. Documentation de Redis-py : https://redis-py.readthedocs.io
2. Documentation officielle de MongoDB : https://docs.mongodb.com
3. Module JSON en Python : https://docs.python.org/3/library/json.html