

# Redis/MongoDB

## Le NoSQL à la rescousse

Ryan Ramassamy

12 octobre 2024

### Table des matières

- 1 Introduction
- 2 Matériel et méthodes
  - 2.1 Installation des différents environnements
  - 2.2 Lancement des bases
- 3 Résultats
  - 3.1 Dénormalisation et implémentation de la base de donnée
  - 3.2 Résolution de problèmes
    - 3.2.1 Retrouver les détails des réservations
    - 3.2.2 Retrouver les villes d'arrivées des vols
    - 3.2.3 Retrouver les différents pilotes
    - 3.2.4 Retrouver les villes de départ en fonction des villes d'arrivées
    - 3.2.5 Mesurer le temps d'exécution du script de dénormalisation & complexité
- 4 Discussion
- 5 Conclusion

## Résumé

*Ce rapport traite de l'utilisation de Redis et MongoDB, deux systèmes de gestion de base de données, faisant partie de la famille des système Nosql ou Not Only SQL. Nous démarrons alors par la demande de notre client, travaillant dans une entreprise tournant autour du transport aérien, ayant besoin d'aide pour stocker et visualiser ses données dans un base de données. Il est alors montré comment installer Redis et MongoDB, démarrer un serveur côté client et comment interagir et gérer les base de données à l'aide de python.*

## 1 Introduction

Le client, opérant dans le secteur du transport aérien, rencontre des difficultés pour gérer efficacement l'ensemble de ses données critiques. Entre la gestion des itinéraires, des réservations, des horaires de vols et les données relatives aux avions, le client éprouve des difficultés dans la gestion de ses bases de données actuelles. Les systèmes de gestion traditionnels peinent à répondre aux besoins en termes de rapidité d'accès, de fiabilité, et d'évolutivité.

Notre entreprise propose alors deux solutions, basée sur Redis et MongoDB, deux système de gestion de bases de données NoSQL, reconnu pour leur rapidité et leur capacité à gérer de grands volumes de données en temps réel. Ces derniers peuvent améliorer les performances globales de l'infrastructure de données du client, notamment en ce qui concerne la gestion des caches, la session utilisateur, et le stockage temporaire de données critiques à haute fréquence d'accès.

Ce projet est composé de deux étapes : tout d'abord, 2) il faut mettre en place les environnements Redis et MongoDB capable de supporter les demandes et les besoins du client. 3) Ensuite, à l'aide de Python, il faut alors harmoniser les données et permettre au client d'y avoir accès de manière simple et claire. Ces deux étapes permettent alors de répondre aux mieux aux besoins. Nous devons nous assurer de l'optimisation de ce qui aura été produit. Une fois que Redis et MongoDB sont pleinement opérationnels et que les problématiques de gestion de données sont résolues, nous serons alors en capacité de livrer un produit fonctionnel au client.

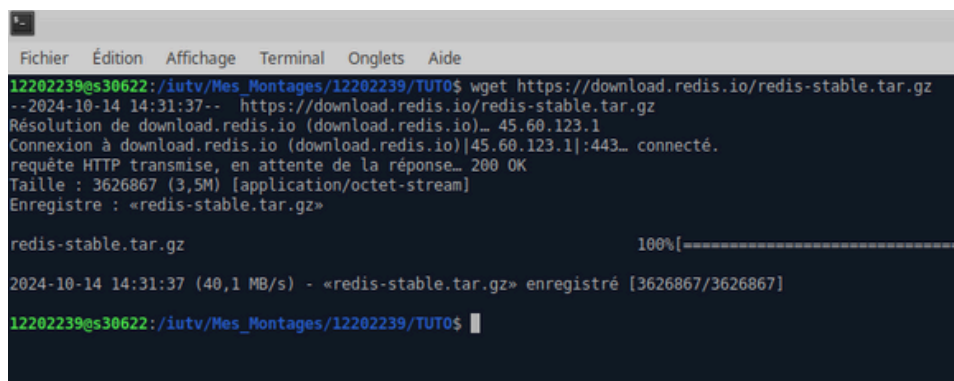
## 2 Matériel et Méthodes

Il est primordial tout d'abord de poser un contexte de travail, il faut donc tout d'abord que nous nous procurions tout les outils nécessaires aux développement afin que l'on puisse par la suite commencer à répondre pleinement aux demandes du client.

### 2.1 Installation des différents environnements

L'installation d'un environnement Redis sur Linux peut-être effectuée en suivant un guide trouvé en ligne. L'installation se fait ici sur une machine de l'entreprise, utilisant Linux comme système d'exploitation.

La première étape consiste alors à récupérer les fichiers sources de Redis à l'aide de la commande : `wget https://download.redis.io/redis-stable.tar.gz`



```
Fichier  Édition  Affichage  Terminal  Onglets  Aide
12202239@30622:/iutv/Mes_Montages/12202239/TUT0$ wget https://download.redis.io/redis-stable.tar.gz
--2024-10-14 14:31:37-- https://download.redis.io/redis-stable.tar.gz
Résolution de download.redis.io (download.redis.io)... 45.60.123.1
Connexion à download.redis.io (download.redis.io)[45.60.123.1]:443... connecté.
requête HTTP transmise, en attente de la réponse... 200 OK
Taille : 3626867 (3,5M) [application/octet-stream]
Enregistre : «redis-stable.tar.gz»

redis-stable.tar.gz                               100%[=====]
2024-10-14 14:31:37 (40,1 MB/s) - «redis-stable.tar.gz» enregistré [3626867/3626867]
12202239@30622:/iutv/Mes_Montages/12202239/TUT0$
```

Comme on peut le voir ici, cette commande télécharge une archive sur notre système.

Il faut ensuite décompresser l'archive obtenue à l'aide de :  
`tar -xzf redis-stable.tar.gz`

Une fois le guide terminé, nous avons alors la possibilité de passer à l'étape suivante.

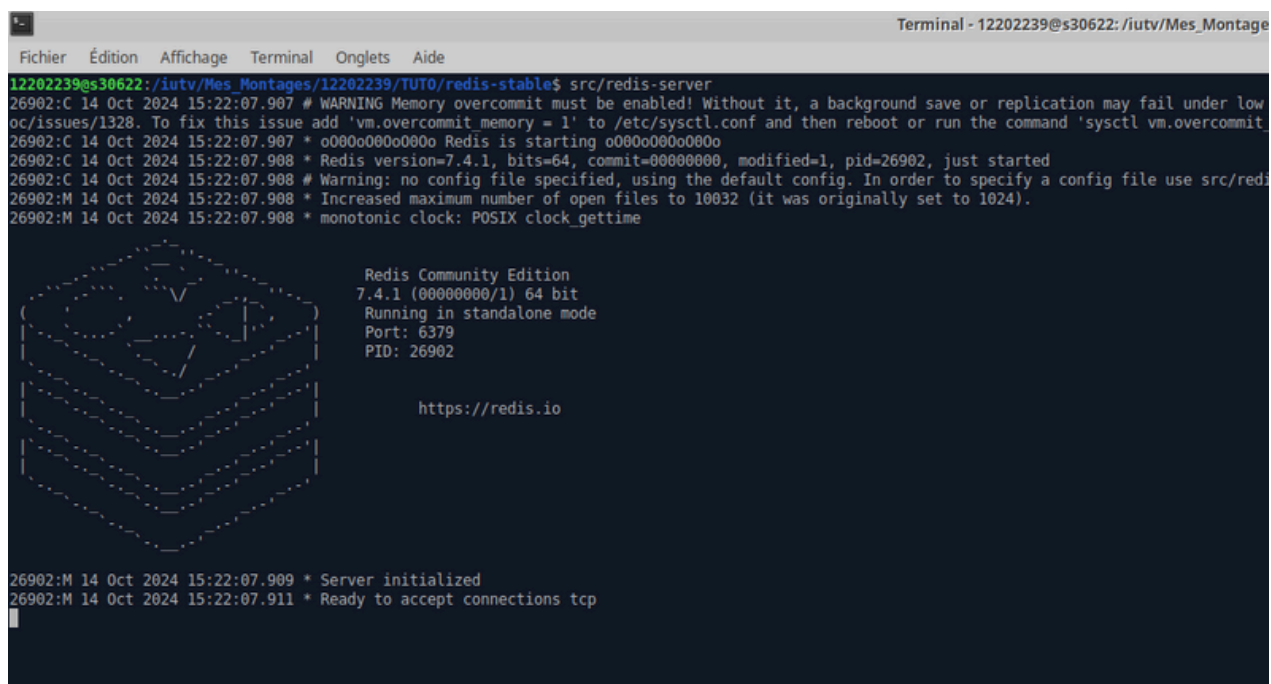
## 2.2 Lancement des bases

En ouvrant le terminal, on peut alors lancer notre serveur redis à l'aide de cette commande : `src/redis-server`

Cette commande nous permet alors de lancer notre serveur qui nous permettra par la suite de nous connecter à notre base de données.

On se retrouve alors sous la même interface que sur la **Figure 1**, ci dessous :

**Figure 1 : Lancement du serveur et interface**



```
Terminal - 12202239@s30622: /iutv/Mes_Montage
Fichier  Édition  Affichage  Terminal  Onglets  Aide
12202239@s30622:/iutv/Mes_Montages/12202239/TUTO/redis-stable$ src/redis-server
26902:C 14 Oct 2024 15:22:07.907 # WARNING Memory overcommit must be enabled! Without it, a background save or replication may fail under low
oc/issues/1328. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit
26902:C 14 Oct 2024 15:22:07.907 * o000o000o000o Redis is starting o000o000o000o
26902:C 14 Oct 2024 15:22:07.908 * Redis version=7.4.1, bits=64, commit=00000000, modified=1, pid=26902, just started
26902:C 14 Oct 2024 15:22:07.908 # Warning: no config file specified, using the default config. In order to specify a config file use src/redi
26902:M 14 Oct 2024 15:22:07.908 * Increased maximum number of open files to 10032 (it was originally set to 1024).
26902:M 14 Oct 2024 15:22:07.908 * monotonic clock: POSIX clock_gettime

Redis Community Edition
7.4.1 (00000000/1) 64 bit
Running in standalone mode
Port: 6379
PID: 26902

https://redis.io

26902:M 14 Oct 2024 15:22:07.909 * Server initialized
26902:M 14 Oct 2024 15:22:07.911 * Ready to accept connections tcp
```

Comme on peut le voir sur la **Figure 1**, le serveur Redis utilise le port 6379 et nous notifie qu'il est bel et bien opérationnel et prêt à répondre aux requêtes

Il nous est alors possible de nous connecter à la base de données (voir **Figure 2**) à l'aide de la commande suivante : `src/redis-cli`.

**Figure 2 : Connexion à la base de données Redis**

```
12202239@es30115:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/redis-stable$ src/redis-cli
127.0.0.1:6379> PING
PONG
127.0.0.1:6379> █
```

Comme on peut le voir sur la **Figure 2**, la connexion à la base de données est un succès et le SGBD répond correctement à notre "PING" en répondant par "PONG".

Du côté de MongoDB, la décision a été d'utiliser le cloud et MongoDB Atlas afin de pouvoir effectuer la mission qui nous a été confiée.

**Figure 3 : Connexion à la base de données MongoDB**

```
MongoDB > connexion.py >...
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3 from dotenv import load_dotenv
4 from os import getenv
5 import json
6
7 load_dotenv()
8 uri = 'mongodb+srv://ryan:GDDnICV360XxvL06@cluster0.tifzs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0'
9
10 # Créez un nouveau client et connectez-vous au serveur
11 client = MongoClient(uri, server_api=ServerApi('1'))
12 db = client["MongoPython"] # Remplacez par le nom de votre base de données
13
14 # Envoyez un ping pour confirmer une connexion réussie
15 try:
16     db.command('ping')
17     print("Pinged your deployment. You successfully connected to MongoDB!")
18 except Exception as e:
19     print(e)
```

**Figure 4 : Connexion à la base de données MongoDB**

```
12202239@es30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 connexion.py
Pinged your deployment. You successfully connected to MongoDB!
12202239@es30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ █
```

Comme on peut le voir sur la **Figure 3**, de la ligne 1 à 5 on commence par importer l'ensemble des modules requis. Les deux premières lignes permettent d'établir la connexion avec la base MongoDB, les lignes 3 et 4 permettent d'utiliser un fichier .env ou l'on peut stocker des variables d'environnement et la ligne 5 permet d'utiliser des outils pour travailler sur JSON. La ligne 8 sert à récupérer le lien de connexion à MongoDB. Ce lien est ensuite utilisé à la ligne 11 où l'on crée un objet MongoClient permettant la connexion. Les lignes 15 à 19 permettent de donner un aperçu de la tâche : elles nous disent si la connexion est un succès ou alors renvoie une erreur en fonction de ce qui cloche. Enfin, comme on peut le voir dans la **Figure 4**, lors de l'exécution du script, on nous indique que nous sommes bien connecté.

Ces deux environnements vont par la suite nous permettre de nous concentrer sur des problèmes plus spécifiques qui nous ont été exposés par notre client. Nous nous sommes tout d'abord assuré que l'insertion des données se réalise sans accro. Ensuite nous avons pu nous tourner vers une analyse plus poussée des données et nous avons pu expérimenter différents scénarios. Nous avons pu alors proposer au client différents moyens de répondre à ses besoins. Nous avons commencé par trouver les différents éléments à cibler afin de les afficher et ensuite nous nous sommes concentré sur la comparaison des deux approches afin de pouvoir aidé le client du mieux que possible à effectuer son choix entre Redis et MongoDB.

### 3 Résultats

Avant de commencer, il nous faut installer Redis pour python avec la commande "pip install Redis" et MongoDB avec la commande " pip install pymongo[srv]" python-dotenv"

#### 3.1 Dénormalisation et implémentation de la base de données

Une fois que le serveur est lancé, que l'on s'est bien connecté à la base de données, il a ensuite fallu dénormaliser et implémenter notre base dans Redis. La **Figure 5** ci dessous, contient le script qui a permis de réaliser ces deux actions.

**Figure 5 : Dénormalisation et implémentation de la base de données avec Redis**

```
script.py > ...
1 import redis
2 import json
3
4 server = redis.Redis(host='localhost', decode_responses=True, port="6379")
5
6 #Traitement du fichier Pilotes.txt
7 piloteFile = open("PILOTES.txt", 'r', encoding='utf-8')
8 pilotes = {}
9
10 for line in piloteFile:
11     line = line.split('\t')
12     pilotes[line[0]] = {"nom": line[1], "naissance": line[2], "ville": line[3].rstrip()}
13
14 #Traitement du fichier Clients.txt
15 clientFile = open("CLIENTS.txt", 'r', encoding='utf-8')
16 clients = {}
17
18 for line in clientFile:
19     line = line.split('\t')
20     clients[line[0]] = {"nom": line[1], "numeroRue": line[2], "nomRue": line[3], "codePostal": line[4], "ville": line[5].rstrip()}
21
22 #Traitement du fichier DefClasses.txt
23 classesFile = open("DEFCLASSES.txt", 'r', encoding='utf-8')
24 classes = {}
25
26 for line in classesFile:
27     line = line.split('\t')
28     if line[0] not in classes:
29         classes[line[0]] = {line[1]: int(line[2].rstrip())}
30     else:
31         classes[line[0]][line[1]] = int(line[2].rstrip())
32
33 #Traitement du fichier Avions.txt
34 avionsFile = open("AVIONS.txt", 'r', encoding='utf-8')
35 avions = {}
36
37 for line in avionsFile:
38     line = line.rstrip().split("\t")
39     avions[line[0]] = {"nom": line[1], "capacite": line[2], "ville": line[3]}
40
41 #Traitement du fichier Vols.txt
42 volsFile = open("VOLS.txt", 'r', encoding='utf-8')
43 vols = {}
44
45 for line in volsFile:
46     line = line.split("\t")
47     vols[line[0]] = {"villeDepart": line[1], "villeArrivee": line[2], "dateDepart": line[3], "heureDepart": line[4], "dateArrivee": line[5],
48                     "heureArrivee": line[6], "pilote": pilotes[line[7]], "avion": avions[line[8].rstrip()]}
49
50 #Traitement du fichier Reservations.txt
51 reservationFile = open("RESERVATIONS.txt", 'r', encoding='utf-8')
52 reservations = []
53
54 for line in reservationFile:
55     line = line.split('\t')
56     reservations.append({"client": clients[line[0]], "vol": vols[line[1]], "classe": {"nom": line[2], "coeffPrix": classes[line[1]][line[2]]},
57                         "places": int(line[3].rstrip())})
58
59 #Ajout des données dans la base
60 for i in range(len(reservations)):
61     server.set(f"reservations:{i+1}", json.dumps({"reservations": reservations[i]}))
62
```

Comme le montre la Figure 5, les deux premières lignes permettent d'importer JSON pour manipuler les données, et Redis, pour utiliser le SGBD. La ligne 4 permet d'initialiser la connexion au serveur Redis en utilisant le port 6379 comme vu dans la Figure 1. Les lignes 7, 15, 23, 34, 42 et 51 permettent d'utiliser les fichiers ".txt". La première boucle for (lignes 10 à 12) permet de lire chaque ligne du fichier "PILOTES.txt", de diviser chaque ligne de ce fichier en une liste d'éléments et de créer pour chaque pilotes, une clé associé à un dictionnaire contenant toutes les informations du pilote. L'instruction 'r.strip()' permet de supprimer les espaces non désirés. Les autres boucles for (lignes 18 à 20, 26 à 31, 37 à 39 et 45 à 48) répètent le même processus que celui décrit précédemment : on lit chaque ligne du fichier, on divise le contenu en une liste d'éléments et on associe une clé à un dictionnaire. Les lignes 54 à 57 permettent de récupérer les données de chacun des dictionnaires obtenus en amont afin de reformer un derniers dictionnaires.

*Ce dernier dictionnaire est ensuite parcouru ligne 60 et chaque ligne de ce dictionnaire est alors stocké dans la base de données (à l'aide de 'server.set') sous la forme d'un dictionnaire converti en JSON (à l'aide de 'json.dump').*

Il nous est alors possible de regarder dans notre base de données et on se retrouve alors avec des données telles que présentées dans la **Figure 6** ci dessous.

**Figure 6 : Exemple d'entrées qui ont été ajoutées à la base**

```
11) "reservations:20"  
12) "reservations:30"  
13) "reservations:28"  
14) "reservations:2"  
15) "reservations:11"  
16) "reservations:3"  
17) "reservations:9"  
18) "reservations:31"  
19) "reservations:17"  
20) "reservations:15"  
21) "reservations:13"  
22) "reservations:23"
```



Du côté de MongoDB, la **Figure 7** ci dessous retrace l'implémentation des données

## Figure 7 : Dénormalisation et implémentation de la base de données avec MongoDB

```
MongoDB > script.py > ...
1  from pymongo.mongo_client import MongoClient
2  from pymongo.server_api import ServerApi
3  from dotenv import load_dotenv
4  from os import getenv
5  import json
6
7  load_dotenv()
8  uri = 'mongodb+srv://ryan:GDnICV360XxvL06@cluster0.t1fzs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0'
9
10 # Créez un nouveau client et connectez-vous au serveur
11 client = MongoClient(uri, server_api=ServerApi('1'))
12 db = client["MongoPython"] # Remplacez par le nom de votre base de données
13
14 # Création des collections
15 pilotes_collection = db["pilotes"]
16 clients_collection = db["clients"]
17 classes_collection = db["classes"]
18 avions_collection = db["avions"]
19 vols_collection = db["vols"]
20 reservations_collection = db["reservations"]
21
22 # Envoyez un ping pour confirmer une connexion réussie
23 try:
24     db.command('ping')
25     print("Pinged your deployment. You successfully connected to MongoDB!")
26 except Exception as e:
27     print(e)
28
29
30 # Traitement du fichier Pilotes.txt
31 with open("PILOTES.txt", 'r', encoding='utf-8') as piloteFile:
32     for line in piloteFile:
33         line = line.split('\t')
34         pilote_data = {"id": line[0], "nom": line[1], "naissance": line[2], "ville": line[3].rstrip()}
35         pilotes_collection.insert_one(pilote_data)
36
MongoDB > script.py > ...
37 # Traitement du fichier Clients.txt
38 with open("CLIENTS.txt", 'r', encoding='utf-8') as clientFile:
39     for line in clientFile:
40         line = line.split('\t')
41         client_data = {
42             "id": line[0],
43             "nom": line[1],
44             "numeroRue": line[2],
45             "nomRue": line[3],
46             "codePostal": line[4],
47             "ville": line[5].rstrip()
48         }
49         clients_collection.insert_one(client_data)
50
51 # Traitement du fichier DefClasses.txt
52 with open("DEFCLASSES.txt", 'r', encoding='utf-8') as classesFile:
53     for line in classesFile:
54         line = line.split('\t')
55         class_data = {"id": line[0], "nom": line[1], "coeffPrix": int(line[2].rstrip())}
56         classes_collection.insert_one(class_data)
57
58 # Traitement du fichier Avions.txt
59 with open("AVIONS.txt", 'r', encoding='utf-8') as avionsFile:
60     for line in avionsFile:
61         line = line.rstrip().split("\t")
62         avion_data = {"id": line[0], "nom": line[1], "capacite": line[2], "ville": line[3]}
63         avions_collection.insert_one(avion_data)
64
65 # Traitement du fichier Vols.txt
66 with open("VOLS.txt", 'r', encoding="utf-8") as volsFile:
67     for line in volsFile:
68         line = line.split("\t")
69         vol_data = {
70             "id": line[0],
71             "villeDepart": line[1],
72             "villeArrivee": line[2],
73             "dateDepart": line[3],
74             "heureDepart": line[4],
75             "dateArrivee": line[5],
76             "heureArrivee": line[6],
77             "pilote": line[7].rstrip(),
78             "avion": line[8].rstrip()
79         }
80         vols_collection.insert_one(vol_data)
```

```

MongoDB > script.py > ...
82 # Traitement du fichier Reservations.txt
83 with open("RESERVATIONS.txt", 'r', encoding='utf-8') as reservationFile:
84     for line in reservationFile:
85         line = line.split('\t')
86         reservation_data = {
87             "client": line[0],
88             "vol": line[1],
89             "classe": {"nom": line[2]}, # Vous devrez probablement récupérer le coeffPrix des classes ici
90             "places": int(line[3].rstrip())
91         }
92         reservations_collection.insert_one(reservation_data)
93
94 print("Données insérées dans MongoDB avec succès.")
95
96
97 # Liste les différents bases de données
98 print(client.list_database_names())

```

On remarque ici que l'approche utilisée pour le traitement des données est quasiment la même que celle utilisée pour Redis. La ligne 12 permet d'accéder à la base de données ou de la créer si cette dernière n'existe pas. Les lignes 15 à 20 permettent de définir les différentes collections pour les données ou de les créer si elles n'existent pas. Les lignes 23 à 27 permettent d'ouvrir le fichier 'PILOTES.txt', de créer un dictionnaire contenant les données du pilote à l'aide de la boucle for et d'insérer chaque pilote dans la collection. Les autres boucles situées lignes 30 à 41, 44 à 48, 51 à 55, 58 à 72 et 75 à 84 suivent cette même logique : on ouvre les fichiers .txt, on divise les données en utilisant 'rstrip().split("\t")' et on crée un dictionnaire qui va ensuite être inséré dans la collection correspondante. La commande 'insert\_one' permet ces opérations : elle permet d'insérer un élément dans une collection passé en paramètre. Enfin, la ligne 86 nous notifie que toutes les données ont bien été insérées.

```

12202239@es30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 script.py
Données insérées dans MongoDB avec succès.
12202239@es30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ █

```

En exécutant le script, on obtient alors cette réponse qui nous informe que toutes les données ont bien été insérées dans notre base de données.

## 3.2 Résolution des problèmes

### 3.2.1 Retrouver les détails des réservations

Les données étant maintenant bien implémentée dans la base de données, il est maintenant possible de répondre aux différentes demandes du client.

La première requête du client consistait alors à ce que le client puisse avoir accès à toutes les réservations enregistrées. L'objectif était donc de donner la bonne requête afin que le client puisse avoir accès à une réservation spécifique avec toutes les informations qu'elles contiennent, comme c'est le cas dans la **Figure 8**.

**Figure 8 : Exemple de données stockées dans chacune des entrées.**

```
127.0.0.1:6379> get reservations:5
{"reservations": [{"client": {"nom": "Laveran", "numéro": "123", "nom": "Place L. Payel", "codePostal": "13100", "ville": "Aix-en-Provence"}, "ville": {"villeDepart": "Paris", "villeArrivee": "Nice", "dateDepart": "11/04/07", "heureDepart": "22:00", "dateArrivee": "11/04/07", "heureArrivee": "22:15", "pilote": {"nom": "Goulet", "naissance": "1944", "ville": "Marseille"}, "avion": {"nom": "Boeing 737", "capacite": "350", "ville": "Paris"}, "classe": {"nom": "Business", "coeffPrix": 2, "places": 5}}]}
127.0.0.1:6379> get reservations:15
{"reservations": [{"client": {"nom": "Lorentz", "numéro": "123", "nom": "Rond Point Du Prado", "codePostal": "13000", "ville": "Marseille"}, "ville": {"villeDepart": "Amsterdam", "villeArrivee": "Marseille", "dateDepart": "10/04/07", "heureDepart": "22:20", "dateArrivee": "11/04/07", "heureArrivee": "11:10", "pilote": {"nom": "Leblanc", "naissance": "1960", "ville": "Marseille"}, "avion": {"nom": "Boeing 707", "capacite": "350", "ville": "Beyrouth"}, "classe": {"nom": "Business", "coeffPrix": 3, "places": 1}}]}
127.0.0.1:6379> get reservations:25
{"reservations": [{"client": {"nom": "Léonard", "numéro": "145", "nom": "Rond Point Du Prado", "codePostal": "13000", "ville": "Marseille"}, "ville": {"villeDepart": "Marseille", "villeArrivee": "Amsterdam", "dateDepart": "11/04/07", "heureDepart": "18:10", "dateArrivee": "11/04/07", "heureArrivee": "9:10", "pilote": {"nom": "Leblanc", "naissance": "1960", "ville": "Marseille"}, "avion": {"nom": "Airbus A320", "capacite": "230", "ville": "Nice"}, "classe": {"nom": "Touriste", "coeffPrix": 3, "places": 6}}]}
127.0.0.1:6379>
```

La requête "get reservations:" accompagnée du numéro de réservation permet d'avoir accès à toutes les informations concernant la réservation souhaitée.

Et du côté de MongoDB, la **Figure 9.1 et 9.2** nous permettent d'obtenir un aperçu de nos données.

**Figure 9.1 : Fragment de code permettant d'afficher les données dans la base MongoDB.**

```
collection = db["reservations"] # Remplacez par le nom de votre collection

# Récupérer tous les documents
documents = collection.find()

for doc in documents:
    print(doc)
```

La première ligne permet de cibler la bonne collection où nos données sont stockées. La commande '**.find()**' nous permet alors de récupérer l'ensemble des données et la boucle for qui suit permet de lister les réservations les unes à la suite des autres.

Figure 9.2 : Affichage des données stockées dans la base.

```
12202239es30108: /iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 get.py
{"id": "ObjectId('6723aad6627ed3fbb65a2bfe')", "client": "1001", "vol": "V690", "classe": {"nom": "Business", "places": 3},
{"id": "ObjectId('6723aad6627ed3fbb65a2bff')", "client": "1031", "vol": "V101", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c00')", "client": "1006", "vol": "V790", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c01')", "client": "1008", "vol": "V790", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c02')", "client": "1009", "vol": "V901", "classe": {"nom": "Business", "places": 5},
{"id": "ObjectId('6723aad6627ed3fbb65a2c03')", "client": "1011", "vol": "V601", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c04')", "client": "1013", "vol": "V601", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c05')", "client": "1020", "vol": "V601", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c06')", "client": "1017", "vol": "V890", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c07')", "client": "1020", "vol": "V150", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c08')", "client": "1022", "vol": "V601", "classe": {"nom": "Business", "places": 3},
{"id": "ObjectId('6723aad6627ed3fbb65a2c09')", "client": "1027", "vol": "V101", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0a')", "client": "1027", "vol": "V890", "classe": {"nom": "Business", "places": 5},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0b')", "client": "1027", "vol": "V601", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0c')", "client": "1027", "vol": "V150", "classe": {"nom": "Business", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0d')", "client": "1001", "vol": "V141", "classe": {"nom": "Business", "places": 3},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0e')", "client": "1007", "vol": "V690", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c0f')", "client": "1008", "vol": "V690", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c10')", "client": "1031", "vol": "V690", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c11')", "client": "1031", "vol": "V790", "classe": {"nom": "Business", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c12')", "client": "1001", "vol": "V790", "classe": {"nom": "Touriste", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c13')", "client": "1001", "vol": "V150", "classe": {"nom": "Touriste", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c14')", "client": "1031", "vol": "V101", "classe": {"nom": "Touriste", "places": 5},
{"id": "ObjectId('6723aad6627ed3fbb65a2c15')", "client": "1033", "vol": "V101", "classe": {"nom": "Touriste", "places": 7},
{"id": "ObjectId('6723aad6627ed3fbb65a2c16')", "client": "1028", "vol": "V101", "classe": {"nom": "Touriste", "places": 6},
{"id": "ObjectId('6723aad6627ed3fbb65a2c17')", "client": "1021", "vol": "V101", "classe": {"nom": "Touriste", "places": 6},
{"id": "ObjectId('6723aad6627ed3fbb65a2c18')", "client": "1002", "vol": "V101", "classe": {"nom": "Touriste", "places": 5},
{"id": "ObjectId('6723aad6627ed3fbb65a2c19')", "client": "1008", "vol": "V890", "classe": {"nom": "Touriste", "places": 4},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1a')", "client": "1009", "vol": "V141", "classe": {"nom": "Touriste", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1b')", "client": "1009", "vol": "V890", "classe": {"nom": "Touriste", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1c')", "client": "1022", "vol": "V890", "classe": {"nom": "Touriste", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1d')", "client": "1017", "vol": "V890", "classe": {"nom": "Touriste", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1e')", "client": "1001", "vol": "V601", "classe": {"nom": "Econometique", "places": 6},
{"id": "ObjectId('6723aad6627ed3fbb65a2c1f')", "client": "1029", "vol": "V101", "classe": {"nom": "Econometique", "places": 7},
{"id": "ObjectId('6723aad6627ed3fbb65a2c20')", "client": "1029", "vol": "V150", "classe": {"nom": "Econometique", "places": 7},
{"id": "ObjectId('6723aad6627ed3fbb65a2c21')", "client": "1025", "vol": "V890", "classe": {"nom": "Econometique", "places": 4},
{"id": "ObjectId('6723aad6627ed3fbb65a2c22')", "client": "1005", "vol": "V890", "classe": {"nom": "Econometique", "places": 6},
{"id": "ObjectId('6723aad6627ed3fbb65a2c23')", "client": "1015", "vol": "V101", "classe": {"nom": "Econometique", "places": 1},
{"id": "ObjectId('6723aad6627ed3fbb65a2c24')", "client": "1009", "vol": "V790", "classe": {"nom": "Econometique", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c25')", "client": "1017", "vol": "V890", "classe": {"nom": "Econometique", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c26')", "client": "1018", "vol": "V790", "classe": {"nom": "Econometique", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c27')", "client": "1005", "vol": "V601", "classe": {"nom": "Econometique", "places": 5},
{"id": "ObjectId('6723aad6627ed3fbb65a2c28')", "client": "1011", "vol": "V141", "classe": {"nom": "Econometique", "places": 8},
{"id": "ObjectId('6723aad6627ed3fbb65a2c29')", "client": "1006", "vol": "V101", "classe": {"nom": "Econometique", "places": 4},
{"id": "ObjectId('6723aad6627ed3fbb65a2c2a')", "client": "1006", "vol": "V601", "classe": {"nom": "Econometique", "places": 3},
{"id": "ObjectId('6723aad6627ed3fbb65a2c2b')", "client": "1004", "vol": "V901", "classe": {"nom": "Econometique", "places": 2},
{"id": "ObjectId('6723aad6627ed3fbb65a2c2c')", "client": "1002", "vol": "V901", "classe": {"nom": "Econometique", "places": 5}}
```

Comme on peut le voir, l'ensemble des réservations sont listées ici.

### 3.2.2 Retrouver les villes d'arrivées des vols

Cependant, cela n'était pas l'unique problème que rencontrait le client. En effet, ce dernier éprouvait des difficultés à retrouver les détails des vols enregistrés dans la base et plus précisément les destinations de ces derniers. Il était donc nécessaire de trouver un moyen de retrouver les villes d'arrivées. Les **Figures 10.1 et 10.2** ci-dessous contiennent premièrement la fonction python permettant de récupérer les noms de toutes les villes et ensuite de les afficher dans notre base de données.

**Figure 10.1 : Fonction permettant de retrouver les villes d'arrivée sur Redis.**

```
1 import redis
2 import json
3
4 server = redis.Redis(host='localhost', decode_responses=True, port="6379")
5
6 def villes_arrivee_reservations():
7     # Récupérer toutes les clés des réservations
8     keys = server.keys('reservations:*)
9
10    villes_arrivee = set() # Utilisation d'un set pour éviter les doublons
11
12    # Parcourir chaque réservation pour extraire la ville d'arrivée
13    for i in keys:
14        # Récupérer les données de la réservation (au format JSON)
15        data = json.loads(server.get(i))
16
17        # Extraire la ville d'arrivée du vol associé à la réservation
18        ville_arrivee = data['reservations']['vol']['villeArrivee']
19
20        # Ajouter la ville d'arrivée au set
21        villes_arrivee.add(ville_arrivee)
22
23    # Retourner la liste des villes d'arrivée
24    return list(villes_arrivee)
25
26 villes_arrivee = villes_arrivee_reservations()
27 server.set('villes_arrivee', json.dumps(villes_arrivee))
28
29 print(villes_arrivee)
```

Les deux premières lignes importent les bibliothèques nécessaires au développement. La ligne 4 nous permet d'établir la connexion avec le serveur Redis. De la ligne 6 à la ligne 27, on retrouve la fonction qui nous permet de traiter la demande. La ligne 8 nous permet de récupérer l'ensemble des réservations présentes dans la base. La ligne 10 crée un ensemble vide à l'aide de 'set()', ce qui nous permettra d'éviter les doublons. La boucle for à la ligne 13 nous permet de parcourir chacune des réservations. La ligne 14 nous permet de récupérer la valeur associée à la clé i depuis Redis, qui est une chaîne JSON, et la charge en tant qu'objet Python avec 'json.loads'. Le résultat est un dictionnaire nommé data qui contient les informations de réservation. On peut alors accéder à la ville d'arrivée comme le montre la ligne 18 on l'on se retrouve dans les données du vol contenu dans la réservation. La ligne 21 nous permet alors d'enregistrer cette d'arrivée dans le set défini au préalable. Enfin la ligne 27 nous permet de définir une variable qui va recevoir la liste des villes d'arrivées et charger cette variable dans notre base de données Redis.



Une fois le script de la **Figure 10.1** exécuté, en retournant dans notre base, la requête "get:" nous renvoie alors cette liste qui contient alors toutes les villes d'arrivées .

**Figure 10.2 : Résultat de la fonction dans la base de données**

```
127.0.0.1:6379> get villes_arrivee
["\\Amsterdam\\", "\\Pekin\\", "\\Marseille\\", "\\Nice\\"]
127.0.0.1:6379> 
```

Dans le cas de MongoDB, l'approche est similaire à celle de Redis, cependant, comme on peut le voir dans la **Figure 11**, la manière de récupérer les données étant propre à MongoDB (voir ligne 18).

**Figure 11 : Fonction permettant de retrouver les villes d'arrivée sur MongoDB.**

```
MongoDB > fonction1.py > ...
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3 from dotenv import load_dotenv
4 from os import getenv
5 import json
6
7 load_dotenv()
8 uri = 'mongodb+srv://ryan:G00nICV360XxvL06@cluster0.t1fzs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0'
9
10 # Créez un nouveau client et connectez-vous au serveur
11 client = MongoClient(uri, server_api=ServerApi('1'))
12 db = client["MongoPython"] # Remplacez par le nom de votre base de données
13 collection = db["reservations"]
14 vols_collection = db["vols"]
15
16 def villes_arrivee_reservations():
17     # Récupérer toutes les réservations
18     reservations = collection.find()
19
20     villes_arrivee = set() # Utilisation d'un set pour éviter les doublons
21
22     # Parcourir chaque réservation pour extraire la ville d'arrivée
23     for reservation in reservations:
24         vol_id = reservation['vol'] # Obtenez l'ID du vol
25         vol_data = vols_collection.find_one({"id": vol_id}) # Récupérer les détails du vol
26
27         if vol_data and 'villeArrivee' in vol_data: # Vérifiez que vol_data est valide
28             ville_arrivee = vol_data['villeArrivee']
29             villes_arrivee.add(ville_arrivee) # Ajouter la ville d'arrivée au set
30
31     # Retourner la liste des villes d'arrivée
32     return list(villes_arrivee)
33
34 # Appel de la fonction pour obtenir les villes d'arrivée
35 villes_arrivee = villes_arrivee_reservations()
36 |
37 # Affichage des villes d'arrivée
38 print("Villes d'arrivée des réservations :", villes_arrivee)
```

Dans le cas de MongoDB on utilise les collections pour récupérer nos données comme on peut le voir aux lignes 13 et 14 (sans oublier de faire tout le nécessaire pour se connecter à la base (voir ligne 1 à 12)). La ligne 18 nous permet de récupérer toutes les réservations présente dans la collection et la ligne 20 nous permet de définir un set pour éviter les doublons. la boucle for de la ligne 23 à 25 sert à parcourir les réservations et extraire les données de chaque vol grâce à **'find\_one'** qui récupère les données du vol actuellement traité. l'instruction 'if' couvrant les lignes 27 à 29 nous permet de récupérer les villes d'arrivées et de les stocker dans notre set. Enfin il ne nous reste plus qu'à retourner la liste des villes à l'aide de la ligne 32.

**Figure 12 : Résultat de la fonction**

```
12202239@s30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 fonction1.py
Villes d'arrivée : ['Marseille', 'Pekin', 'Amsterdam', 'Nice']
12202239@s30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$
```

Comme on peut le voir, les deux scripts nous renvoient les mêmes villes d'arrivées : Marseille, Nice, Pékin et Amsterdam.

### 3.2.3 Retrouver les différents pilotes

Ensuite, la demande suivante formulée par le client était de retrouver le nombre de pilotes qui étaient renseignés dans la base de données. Il a alors été question de réitérer l'opération effectué pour le problème précédent et de parcourir à nouveau les différentes réservations pour alors obtenir ce qui suit dans les **Figures 13.1 et 13.2**.

**Figure 13.1 : Fonction permettant de retrouver le nombre de pilotes présent dans la base Redis**

```
1 import redis
2 import json
3
4 server = redis.Redis(host='localhost', decode_responses=True, port="6379")
5
6 def compter_pilotes():
7     # Récupérer toutes les clés des réservations
8     keys = server.keys('reservations:*')
9
10    # Créer un set pour stocker les pilotes uniques
11    pilotes = set()
12
13    # Parcourir chaque réservation
14    for i in keys:
15        # Récupérer les données de la réservation (au format JSON)
16        reservation_data = json.loads(server.get(i))
17
18        # Extraire le pilote de la réservation
19        pilote = reservation_data['reservations']['vol']['pilote']
20
21        # Ajouter le pilote au set (les doublons seront ignorés automatiquement)
22        pilotes.add(pilote['nom'])
23
24    # Retourner le nombre de pilotes uniques
25    return len(pilotes)
26
27 nombre_pilotes = compter_pilotes()
28 print(nombre_pilotes)
29
30 server.set("NB_Pilotes",nombre_pilotes)
```

La boucle lignes 14 à 25 permet de récupérer les données de chaque réservation (ligne16), d'en extraire le pilote (ligne 19) et d'ajouter ce dernier au set (ligne22) initialisé au préalable (ligne 11). Enfin il ne nous reste plus qu'à compter le nombre de pilotes présents dans ce set à l'aide de '**len(pilotes)**' à la ligne 25 et de charger cette donnée dans la base à la ligne 30 avec '**server.set**' en lui attribuant la clé '**NB\_Pilotes**' pour qu'elle soit plus simple à retrouver pour l'utilisateur.

**Figure 13.2 : Résultat de la fonction dans la base de données**

```
127.0.0.1:6379> get NB_Pilotes
"5"
127.0.0.1:6379> █
```

Comme nous pouvons le voir sur la Figure 13.2, nous avons 5 pilotes dans notre base de données.



Et du côté de MongoDB, l'approche est une nouvelle fois très similaire à celle de Redis, comme on peut le voir dans les **Figures 14.1** et **14.2**.

**Figure 14.1 : Fonction permettant de retrouver le nombre de pilotes présent dans la base MongoDB**

```
MongoDB > fonction2.py > ...
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3 from dotenv import load_dotenv
4 from os import getenv
5 import json
6
7 load_dotenv()
8 uri = 'mongodb+srv://ryan:G00nICV360XxvL06@cluster0.tlfs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0'
9
10 # Créez un nouveau client et connectez-vous au serveur
11 client = MongoClient(uri, server_api=ServerApi('1'))
12 db = client["MongoPython"] # Remplacez par le nom de votre base de données
13 collection = db["reservations"] # Remplacez par le nom de votre collection
14 vols_collection = db["vols"]
15
16 def compter_pilotes_dans_reservations():
17     pilotes_uniques = set()
18
19     # Récupérer toutes les réservations
20     reservations = collection.find()
21
22     # Parcourir chaque réservation
23     for reservation in reservations:
24         # Vérifiez que reservation est un dictionnaire
25         if isinstance(reservation, dict) and 'vol' in reservation:
26             vol_id = reservation['vol'] # ID du vol
27             vol_data = vols_collection.find_one({"id": vol_id}) # Récupérer les détails du vol
28
29             if vol_data and 'pilote' in vol_data: # Vérifiez que vol_data est valide
30                 pilote_id = vol_data['pilote']
31                 pilotes_uniques.add(pilote_id) # Ajouter l'ID du pilote au set
32             else:
33                 print(f"Format inattendu pour la réservation : {reservation}")
34
35     # Retourner le nombre de pilotes uniques
36     return len(pilotes_uniques)
37
38 # Appel de la fonction pour compter les pilotes
39 nombre_de_pilotes = compter_pilotes_dans_reservations()
40
41 # Affichage du nombre de pilotes
42 print(f"Nombre de pilotes dans les réservations : ", nombre_de_pilotes)
```

Dans le cas de MongoDB, on utilise toujours les collections (lignes 13 et 14) et on récupère les réservations à l'aide de **'find'** (ligne 20). Cependant on doit d'abord s'assurer que l'objet que nous sommes en train de manipuler est bien un dictionnaire (ligne 25) avant de pouvoir extraire les données du vol à l'aide de **'find\_one'** qui récupère un seul élément. (lignes 26 et 27) et ensuite extraire le pilote afin de l'ajouter à notre set à l'aide de **'add'** (lignes 29 à 31). On utilise à nouveau **'len'** afin de compter le nombre de pilotes enregistrés (ligne 36).

**Figure 14.2 : Résultat de la fonction**

```
12202239@30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 fonction2.py
Nombre de pilotes dans la base : 5
12202239@30108:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$
```

Comme on peut à nouveau le voir, les deux scripts nous renvoient les mêmes mêmes réponses.

### 3.2.4 Retrouver les villes de départs en fonction des villes d'arrivées

Une fois l'ensemble des demandes du client traitées, nous avons décidé de lui donner une meilleure idée des opérations qui était susceptible de lui être utile. Nous avons alors commencé par trouver le moyen de retrouver l'ensemble des villes de départ des différents vols à partir des villes d'arrivées recueillies précédemment. Nous nous sommes alors tourné vers cette ancienne fonction afin d'effectuer cette opération comme le montre les **Figures 15.1, 15.2, 16.1 et 16.2**.

**Figure 15.1 : Fonction renvoyant la liste des villes de départ associée aux villes d'arrivée sur MongoDB**

```
fonction3.py > ...
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3 from dotenv import load_dotenv
4 from os import getenv
5 import json
6 from fonction1 import villes_arrivee_reservations
7
8 load_dotenv()
9 uri = 'mongodb+srv://ryan:GD0nICV360XxvL06@cluster0.tlfs.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0'
10
11 client = MongoClient(uri, server_api=ServerApi('1'))
12 db = client["MongoPython"]
13
14 def get_villeDepart_from_villeArrivee():
15
16     # Appel de la fonction pour obtenir la liste des villes d'arrivée
17     arrival_cities = villes_arrivee_reservations()
18
19     # Dictionnaire pour stocker les villes de départ par ville d'arrivée
20     depart_from_arrivee = {}
21
22     # Parcourir chaque ville d'arrivée
23     for i in arrival_cities:
24         # Trouver tous les vols qui arrivent à cette ville
25         vols = db["vols"].find({"villeArrivee": i})
26
27         # Ensemble pour stocker les villes de départ uniques
28         departure_cities = set()
29
30         # Parcourir les vols et extraire les villes de départ
31         for vol in vols:
32             if "villeDepart" in vol: # Vérifier que la ville de départ existe
33                 departure_cities.add(vol["villeDepart"])
34
35         # Ajouter la liste des villes de départ à la clé correspondant à la ville d'arrivée
36         depart_from_arrivee[i] = list(departure_cities)
37
38     return depart_from_arrivee
39
40 if __name__ == "__main__":
41     result = get_villeDepart_from_villeArrivee()
42     for arrival, departures in result.items():
43         print("Villes de départ pour la ville d'arrivée ", arrival, " : ", departures)
```

Avant toute chose, à la ligne 6, on s'assure d'importer la fonction '**villes\_arrivee\_reservations**' que nous avons défini dans un autre fichier afin de pouvoir l'utiliser dans ce script. On appelle ensuite cette fonction à la ligne 17 afin de stocker l'ensemble des villes d'arrivées. Ensuite, on définit un dictionnaire à la ligne 20 afin de pouvoir contenir notre réponse finale. On utilise ensuite une boucle pour parcourir chaque ville d'arrivée à la ligne 23. En utilisant la collection vols, on cherche l'ensemble des vols qui arrivent à cette destination (ligne 25). A la ligne 28, on initialise un set. Enfin la boucle de la ligne 31 à 33 nous permet de vérifier si la ville de départ existe bien et d'ajouter cette dernière au set.

Enfin, la ligne 36, faisant partie de la première boucle (ligne 23) ajoute à notre dictionnaire : la ville d'arrivée actuelle (**depart\_grom\_arrivee[i]**) et la liste des villes de départ correspondantes. la ligne 40 nous permet de s'assurer que le bloc de code suivant ne s'exécute que si le script est lancé directement et non lorsqu'il est importé en tant que module dans un autre script. Le bloc qui suit cette instruction (lignes 41 à 43) nous permet de recevoir un visuel plus joli du résultat de la fonction.

**Figure 15.2 : Résultat de la fonction**

```
12202239@t001(0) /lutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 fonction3.py
Villes d'arrivée : ['Nice', 'Pekin', 'Amsterdam', 'Marseille']
Villes de départ pour la ville d'arrivée Nice : ['Paris']
Villes de départ pour la ville d'arrivée Pekin : ['Paris', 'Marseille']
Villes de départ pour la ville d'arrivée Amsterdam : ['Marseille']
Villes de départ pour la ville d'arrivée Marseille : ['Metz', 'Paris', 'Amsterdam', 'Strasbourg', 'Ajaccio']
12202239@t001(0) /lutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$
```

Une fois encore, les approches entre MongoDB et Redis se ressemblent fortement

**Figure 16.1 : Fonction renvoyant la liste des villes de départ associée aux villes d'arrivée sur Redis**

```
pythonRedis > pythonRedis > fonction3.py > ...
1 import json
2 import redis
3 from fonction1 import villes_arrivee_reservations
4
5 # Connexion au serveur Redis
6 server = redis.Redis(host='localhost', decode_responses=True, port=6379)
7
8 def get_villeDepart_from_villeArrivee():
9     arrival_cities = villes_arrivee_reservations() # Appel de la fonction importée
10    depart_from_arrivee = {} # Dictionnaire pour stocker les villes de départ pour chaque ville d'arrivée
11
12    # Parcourir chaque ville d'arrivée
13    for i in arrival_cities:
14        departure_cities = set() # Utiliser un set pour éviter les doublons
15        reservation_keys = server.keys("reservations:") # Obtenir toutes les réservations
16
17        # Parcourir chaque réservation pour vérifier les villes de départ associées
18        for key in reservation_keys:
19            reservation_data = json.loads(server.get(key)) # Charger les données JSON de la réservation
20            ville_arrivee = reservation_data["reservations"]["vol"]["villeArrivee"]
21            ville_depart = reservation_data["reservations"]["vol"]["villeDepart"]
22
23            # Si la ville d'arrivée correspond, ajouter la ville de départ au set
24            if ville_arrivee == i:
25                departure_cities.add(ville_depart)
26
27        # Ajouter la liste de villes de départ dans le dictionnaire pour cette ville d'arrivée
28        depart_from_arrivee[i] = list(departure_cities)
29
30    return depart_from_arrivee
31
32 # Exemple d'utilisation
33 if __name__ == "__main__":
34     depart_from_arrivee = get_villeDepart_from_villeArrivee()
35     print(depart_from_arrivee)
```

Encore une fois, on importe la fonction '**villes\_arrivee\_reservations**' qui se trouve dans le fichier '**fonction1.py**' à la ligne 3. Cette fonction est ensuite appelée afin de stocker les villes d'arrivées à la ligne 9 et à nouveau on initialise un dictionnaire à la ligne 10. La boucle for de la ligne 18 à 21 nous permet de parcourir chaque réservations de la base et dans chacune de ces réservations on va d'abord charger les données JSON (ligne 19) et on va ensuite récupérer la ville d'arrivée mais également la ville de départ du vol associé à la réservation (lignes 20 et 21) à l'aide de l'instruction '**[\"reservations\"][\"vol\"]**' pour récupérer les données du vol. On s'assure que la ville d'arrivée est bien valide à la ligne 24 et on ajoute notre ville de départ à la ligne 25. Enfin la ligne 28 nous permet d'ajouter la ville d'arrivée et l'ensemble des villes de départ correspondantes à notre dictionnaire. Le bloc de la ligne 32 à 35 nous permet de s'assurer que le bloc de code suivant ne s'exécute que si le script est lancé directement et non lorsqu'il est importé en tant que module dans un autre script.

**Figure 16.2 : Résultat de la fonction**

```
12202239@q10630:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/pythonRedis$ python3 fonction3.py
['Nice', 'Pekin', 'Amsterdam', 'Marseille']
{'Nice': ['Paris'], 'Pekin': ['Marseille'], 'Amsterdam': ['Marseille'], 'Marseille': ['Ajaccio', 'Metz', 'Amsterdam', 'Strasbourg']}
```

### 3.2.5 Mesurer le temps d'exécution du script de dénormalisation & complexité

Toutes les fonctions ayant été implémentée et étant fonctionnelle, le dernier défi qui a été lancée a été de comparer les complexité et les temps d'exécution des deux scripts de dénormalisation. Les **Figures 17.1 et 17.2** expliquent le déroulement sur Redis.

**Figure 17.1 : Fonction renvoyant le temps d'exécution du script de dénormalisation sur Redis**

```
pythonKedis > pythonKedis > time_1.py > ...
1  import subprocess
2  import time
3
4  def execute_script_and_analyze_complexity():
5      # Mesurer le temps de début
6      start_time = time.perf_counter()
7
8      # Exécuter le script `script.py`
9      process = subprocess.run(["python", "script.py"], capture_output=True, text=True)
10
11     # Mesurer le temps de fin
12     end_time = time.perf_counter()
13
14     # Calculer le temps d'exécution
15     execution_time = end_time - start_time
16
17     # Analyser la complexité
18     # Le script effectue les opérations suivantes :
19     # - Lecture des fichiers : suppose que cela prend un temps linéaire O(N)
20     # - Insertion des données dans Redis : suppose que cela prend un temps linéaire O(R)
21     # La complexité temporelle totale est donc de O(N + R), où :
22     # - N est le nombre de lignes dans tous les fichiers d'entrée combinés
23     # - R est le nombre de réservations
24
25     complexity = "O(N + R)" # Déclaration de la complexité
26
27     # Afficher la sortie du script, le temps d'exécution et la complexité estimée
28     print("Sortie du script :", process.stdout)
29     print("Temps d'exécution :", execution_time, "secondes")
30     print("Complexité estimée :", complexity)
31
32     # Retourner le temps d'exécution et la complexité
33     return execution_time, complexity
34
35 # Exécuter la fonction si le fichier est appelé directement
36 if __name__ == "__main__":
37     execute_script_and_analyze_complexity()
```

Dans un premier temps, nous importons les bibliothèques **subprocess** à la ligne 1, afin d'exécuter des commandes système et des scripts externes depuis Python, et **time** à la ligne 2, pour mesurer le temps d'exécution. La fonction que nous déclarons à la ligne 4 va exécuter un script externe, mesurer son temps d'exécution et afficher une estimation de sa complexité. A la ligne 6, on utilise une fonction de la bibliothèque `time` afin d'enregistrer le temps de début. A la ligne 9, on utilise '**subprocess.run()**' en renseignant dans les parenthèses tout d'abord l'interpréteur et ensuite le script à exécuter. La première option qui suit nous permet de capturer la sortie du script pour pouvoir l'afficher ou la traiter plus tard et la seconde permet de convertir la sortie capturée en chaîne de caractères. La ligne 12 nous permet d'enregistrer le temps de fin de l'exécution du script, toujours en utilisant '**time.perf\_counter()**'.



La ligne 15 calcule le temps d'exécution du script python qui a été exécuté. La ligne 25 donne la complexité du script (les détails de l'argumentation sont en commentaire de la ligne 17 à la ligne 23). Enfin les lignes 28 à 33 nous permettent d'afficher les différents résultats obtenus. On peut observer à la ligne 28 que l'on renvoie la sortie de script.py. Le dernier bloc permet d'exécuter `execute_script_and_analyze_complexity()` seulement si le script est lancé directement, et non s'il est importé dans un autre fichier.

**Figure 17.2 : Résultat de la fonction**

```
12202239@q10630:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/pythonRedis$ python3 time_1.py
Sortie du script :
Temps d'exécution : 0.09851604800132918 secondes
Complexité estimée : O(N + R)
12202239@q10630:/iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/pythonRedis$
```

Et dans le cas de MongoDB, les **Figures 18.1 et 18.2** décrivent le comportement de ces scripts.

**Figure 18.1 : Fonction renvoyant le temps d'exécution du script de dénormalisation sur MongoDB**

```
MongoDB > MongoDB > time.py > ...
1 import time
2 import subprocess
3
4 def measure_execution_time(script_name):
5     start_time = time.time() # Démarrer le chronomètre
6
7     # Exécuter le script
8     subprocess.run(['python', script_name], check=True)
9
10    end_time = time.time() # Arrêter le chronomètre
11    execution_time = end_time - start_time # Calculer le temps d'exécution
12
13    return execution_time
14
15 def analyze_complexity():
16     # Évaluation de la complexité temporelle des opérations
17     # 1. Insertion des pilotes - O(n)
18     # 2. Insertion des clients - O(m)
19     # 3. Insertion des classes - O(k)
20     # 4. Insertion des avions - O(p)
21     # 5. Insertion des vols - O(q)
22     # 6. Insertion des réservations - O(r)
23
24     # En considérant que les nombres de lignes dans les fichiers sont respectivement n, m, k, p, q, et r.
25     # Le temps d'exécution est dominé par les opérations d'insertion, donc la complexité globale est:
26     # O(n + m + k + p + q + r)
27
28     print("Complexité temporelle des opérations dans script.py : O(n + m + k + p + q + r)")
29
30 if __name__ == "__main__":
31     script_name = 'script.py'
32     execution_time = measure_execution_time(script_name)
33     print(f"Temps d'exécution de ", script_name, ' : ', execution_time, " secondes")
34     analyze_complexity()
```

Encore une fois nous commençons par importer les bibliothèques **subprocess** à la ligne 1, afin d'exécuter des commandes système et des scripts externes depuis Python, et **time** à la ligne 2, pour mesurer le temps d'exécution. La fonction que nous déclarons à la ligne 4 prend en paramètre le nom du script à exécuter. Cette fonction mesure le temps d'exécution du script. A la ligne 5, on enregistre le temps de début. A la ligne 8, on exécute le script. La ligne 10 nous permet d'enregistrer le temps de fin de l'exécution du script. La ligne 11 calcule le temps d'exécution du script python qui a été exécuté. La ligne 15 définit une fonction qui donne la complexité du script (les détails de l'argumentation sont en commentaire de la ligne 16 à la ligne 26). La ligne 28 renvoie la complexité finale. Le dernier bloc de la ligne 30 à la ligne 34 permet d'exécuter `measure_execution_time()` seulement si le script est lancé directement, et non s'il est importé dans un autre fichier et affiche le temps d'exécution du script.

**Figure 18.2 : Résultat de la fonction**

```
12202239@r30520(0) /iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$ python3 time.py
Données insérées dans MongoDB avec succès.
Temps d'exécution de script.py : 19.039684057235718 secondes
Complexité temporelle des opérations dans script.py :  $O(n + m + k + p + q + r)$ 
12202239@r30520(0) /iutv/Mes_Montages/12202239/BUT3/S5/BDD_AVANCEES/MongoDB$
```

Comme on peut le voir, le script de dénormalisation de MongoDB est beaucoup plus lent que celui de Redis. Les données sont insérées bien plus longuement et cela se voit également à travers la complexité qui est donnée par l'algorithme Mongo qui montre bien que la complexité de l'algorithme de dénormalisation de MongoDB est plus complexe que celui de Redis.

## 4 Discussion

Dans cette étude, l'objectif principal était de comparer les méthodes de récupération de données entre deux systèmes de gestion de bases de données (SGBD) NoSQL, Redis et MongoDB, afin de faciliter la compréhension et l'usage de ces technologies par le client. Nous avons démontré que chaque SGBD dispose de ses propres outils et méthodes de requête. Par exemple, Redis utilise principalement la commande **GET** pour accéder à une valeur associée à une clé spécifique, tandis que MongoDB utilise la commande **FIND** pour récupérer des données correspondant à des filtres qui peuvent être spécifiés.

Ces différences de requêtes reflètent les particularités techniques et conceptuelles de chaque SGBD. Redis est un outil qui se repose sur le principe clé-valeur, ce qui est très utile pour récupérer les données de manières **rapide** et **simple**, ce qui explique pourquoi on utilise des commandes directes et spécifiques comme GET. MongoDB, quant à lui, est un SGBD conçu pour gérer des structures de données plus complexes, d'où l'utilisation de commandes plus flexibles comme FIND, qui permettent d'effectuer des requêtes plus sophistiquées et plus complexe.

En comparant les résultats obtenus par les fonctions, nous remarquons que les approches de développement entre les deux technologies sont effectivement assez **similaires**, surtout en termes de **structure de code**, comme l'utilisation de boucles pour parcourir les données. Toutefois, cette similarité n'efface pas les particularités de chaque système : Redis est souvent utilisé pour des cas où la rapidité d'accès est importante tandis que MongoDB est plus adapté aux applications nécessitant une structure de données flexible et évolutive.

Cependant, certaines contraintes dans la méthodologie utilisée doivent être soulignées. Par exemple, l'expérience se concentre principalement sur des opérations de lecture simples, sans intégrer des scénarios plus **complexes** qui impliqueraient des écritures, des mises à jour massives ou des contraintes de gestion de la concurrence. Pour obtenir une évaluation plus complète, il serait pertinent d'examiner comment chaque SGBD se comporte dans des contextes de charge élevée ou avec des **données volumineuses** bien que nous avons pu voir qu'il y ait des chances que MongoDB gère mieux l'afflux de données en grand nombre que Redis.



Les résultats montrent que chaque système présente des **avantages** spécifiques et suggèrent qu'un choix entre Redis et MongoDB doit être guidé par les besoins particuliers du client, tels que la vitesse d'accès, la structure des données, et la nature des requêtes envisagées. À l'avenir, il serait bénéfique de poursuivre cette recherche en testant d'autres cas d'utilisation, par exemple en incluant des SGBD relationnels pour mieux cerner la place des solutions NoSQL dans un environnement de données diversifié.

Enfin, on a pu voir que les algorithmes de dénormalisation des données mettaient chacun un certain temps à finir de s'exécuter. En effet on s'est aperçu que l'algorithme utilisant MongoDB était beaucoup plus lent que celui de Redis et que la différence de temps d'exécution était assez importante. Il est donc important de prendre cela en compte lors du choix du SGBD à utiliser.

## 5 Conclusion

Pour conclure, notre client se voit alors proposé deux SGBD distincts ayant chacun leurs particularités et leurs avantages. Nos résultats montrent que, bien que les approches de requêtage diffèrent, Redis et MongoDB sont deux SGBD qui offrent des outils efficaces pour répondre aux besoins de stockage et de récupération de données. Redis, étant optimisé pour des requêtes rapides et simples, se révèle particulièrement adapté dans les cas où la vitesse d'accès est recommandée, tandis que MongoDB, de son côté, se distingue par sa flexibilité dans la gestion de données structurées et évolutives mais perd pas mal de point lorsqu'il s'agit de la vitesse d'exécution.

Notre client se voit donc dans la possibilité de choisir entre les deux, son choix devant être tourné vers l'outil qui lui permettra d'optimiser le stockage et la récupération des données, en fonction donc de la quantité et de la complexité des données.