

BUT 3 Informatique

Université Sorbonne Paris-Nord

IUT de Villetaneuse

Transformation des données SQL en NoSQL : Une étude pratique avec Redis, MongoDB

Réalisé par :

Sidi ESSAADOUNI

Encadré par :

Christophe Cérin

Version 1.0

Table des matières

1	Introduction	1
2	Matériels et Méthodes	2
2.1	Outils utilisés	2
2.1.1	Architecture du projet	3
2.2	Stratégie de Migration de SQL vers NoSQL	4
2.2.1	Étape 1 :Regroupement des informations de Vols, Avions et Pilotes	5
2.2.2	Étape 2 :Intégration des Classes de Vols	6
2.2.3	Étape 3 :Gestion des Clients	6
2.2.4	Étape 4 :Représentation des Réservations	7
2.3	Conversion des données TXT en JSON	7
2.3.1	Insertion des documents dans Redis	8
2.3.2	Insertion des documents dans MongoDB	8
3	Résultats	10
3.1	RéPLICATION des résultats	10
3.2	Exemple de requêtes et de jointures	10
3.3	Utilisation des ressources matérielles	11
3.4	Comparaison des Performances	12
4	Discussion	13
4.1	Comparaison des Performances	13
4.1.1	Performances avec le Jeu de Données d'Origine	14

4.1.2	Performances avec un Grand Volume de Données	14
4.1.3	Interprétation des Résultats	14
4.1.4	Conclusion sur le Choix des Bases de Données	14
4.2	Comparaison des Ressources Matérielles	15
4.3	Limites et Améliorations Possibles	15
5	Conclusion	16
A	Annexes	18
A.1	Configuration Docker pour Redis	18
A.2	Configuration Docker pour MongoDB	18
A.3	Replica Set MongoDB	19
A.4	Configuration Docker pour Python	20
A.5	Création des Documents JSON	20
A.6	Insertion des documents dans Redis	23
A.6.1	Insertion des documents dans MongoDB	24
A.7	Résultats des requêtes	24
A.8	Profiling des Requêtes	28
A.9	Dépôt Git	29

Table des figures

2.1	Architecture du projet avec Docker et bases NoSQL	4
2.2	Schéma de la base SQL	5
2.3	Insertion des documents dans Redis	8
2.4	Insertion des documents dans MongoDB	9
3.1	Comparaison des ressources matérielles utilisées avec les données d'origine et avec un grand volume de données	11
3.2	Comparaison des performances avec les données d'origine et avec un grand volume de données . .	12

Liste des tableaux

4.1 Comparaison des performances entre MongoDB, MONGO_JSON et REDIS_JSON	13
4.2 Comparaison des performances avec un grand volume de données	13

Chapitre 1

Introduction

Dans le cadre de ce projet, nous avons entrepris la transformation d'une base de données relationnelle (SQL) classique en une architecture NoSQL, une démarche qui répond aux exigences modernes de flexibilité, d'évolutivité, et de gestion optimisée des données semi-structurées. Face à l'augmentation du volume de données et aux besoins croissants d'agilité dans les traitements, le modèle SQL a montré des limites, notamment en termes de performance et de rigidité. C'est pour cela que nous avons décidé de mettre en place une solution NoSQL qui permet d'adapter la structure des données aux besoins actuels.

Le choix d'une dénormalisation en format JSON, un processus permettant de simplifier et d'optimiser la manipulation des données, est au cœur de cette transformation. En effet, ce format est particulièrement bien adapté aux bases de données NoSQL, car il facilite l'agrégation des informations liées, élimine les jointures coûteuses, et permet une gestion simplifiée des données sans contrainte de schéma rigide.

Dans cette étude, nous avons utilisé deux bases NoSQL majeures : Redis et MongoDB. Redis, avec sa capacité de stockage en mémoire, offre des performances optimisées pour les lectures rapides et les données temporaires, tandis que MongoDB se distingue par sa persistance de données et sa capacité à gérer des structures JSON complexes. Ce rapport détaille les étapes de la migration, les choix techniques, et les avantages obtenus en termes de performance et de flexibilité, illustrant comment ces technologies peuvent répondre aux besoins métiers modernes.

Résumé

Ce rapport décrit le processus de migration d'une base de données SQL vers un environnement NoSQL, en adoptant une approche de dénormalisation au format JSON. En intégrant Redis et MongoDB, nous démontrons les avantages d'une architecture NoSQL pour le stockage et la manipulation de données semi-structurées, avec des gains en flexibilité et en performance. La combinaison de ces technologies permet d'adapter la gestion des données aux exigences actuelles, en simplifiant les traitements tout en assurant une évolutivité pour des volumes de données plus élevés.

Chapitre 2

Matériels et Méthodes

2.1 Outils utilisés

Voici la liste des outils utilisés pour le projet :

- Ubuntu 24.04: L'OS utilisé pour le développement mais non nécessaire suite à l'utilisation de Docker
- Docker / Docker Compose: pour orchestrer l'environnement de développement
- Redis: pour le stockage des données JSON en mémoire
- MongoDB: pour la persistance des documents JSON
- Python: pour le traitement des données et l'injection en base
- Poetry: pour gérer les dépendances
- LaTeX: pour la rédaction du rapport technique
- VSCode: pour le développement
- Git: pour le contrôle de version

Docker et Docker Compose

Docker est utilisé pour orchestrer l'environnement de développement. Le fichier `compose.yaml` définit les services, incluant Redis, MongoDB et Python (Poetry). Chacun a son propre Dockerfile (ex. `Dockerfile.redis`, `Dockerfile.mongo`). Cela garantit l'isolation des services, ce qui rend le projet facilement déployable sur différentes machines.

Redis et Redis CLI

Redis est utilisé comme base NoSQL pour le stockage des données JSON en mémoire. Les fichiers de connexion et de requêtes (`connectionRedis.py` et `requests_redis.py`) gèrent l'interface entre l'API et Redis. Redis CLI est utilisé pour des tests rapides, et un fichier `.env.redis` gère la configuration sécurisée des variables d'environnement. MongoDB est utilisé pour la persistance des documents JSON. Il est intégré via PyMongo, avec des fichiers de connexion et de requêtes spécifiques (`connectionMongo.py` et `requests_json.py`). Les

données semi-structurées sont ainsi manipulées efficacement. Le fichier `Dockerfile.mongo` assure la configuration de MongoDB dans un conteneur.

Python et Poetry

Python est le langage principal pour le traitement des données et l'injection en base. Poetry est utilisé pour gérer les dépendances, avec un fichier `pyproject.toml` pour créer un environnement cohérent.

2.1.1 Architecture du projet

La figure 2.1 représentant l'architecture du projet avec Docker et bases NoSQL. Cette architecture est basé sur un principe de séparation des responsabilités en séparant en différentes couches :

- **API** : Cette couche communique avec la couche Services. Elle traite les requêtes HTTP (Get, Post, Put, Delete) et envoie les réponses en JSON
- **DAL** : Cette couche contient les fonctions de manipulation des données. Elle communique avec les bases de données via des connexions.
- **Services** : Cette couche contient les fonctions de traitement des données. Elle communique avec les bases de données via des connexions.
- **Config** : Cette couche contient les paramètres de configuration. Elle communique avec les bases de données via des connexions.

Cette architecture est bénéfique car elle permet d'avoir un code propre, modulable et réutilisable. En effet, chaque couche peut être modifiée indépendamment, ce qui facilite la maintenance et le développement de l'application. Concernant les performances, la séparation des couches permet de réduire le temps de chargement des données, ce qui améliore la performance de l'application.

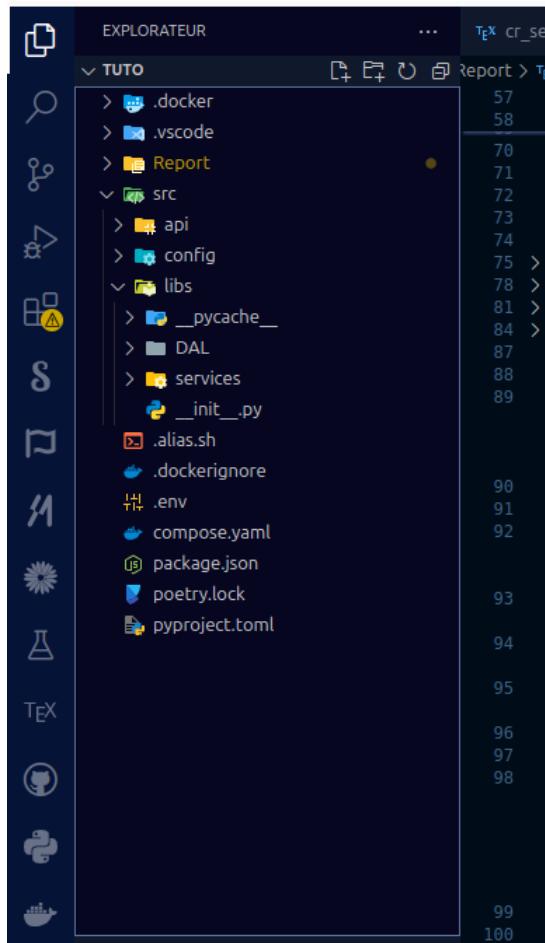


FIGURE 2.1 – Architecture du projet avec Docker et bases NoSQL

2.2 Stratégie de Migration de SQL vers NoSQL

La base de données SQL originale se composait de plusieurs tables interconnectées qui suivaient un modèle relationnel strict. Cependant, ce modèle, bien que robuste pour certains cas d'utilisation, a montré ses limites dans la gestion des données dynamiques et évolutives. Cela nous a conduit à prendre la décision de passer à une architecture NoSQL, plus adaptée aux besoins actuels.

Le schéma suivant illustre la structure de la base SQL avant migration :

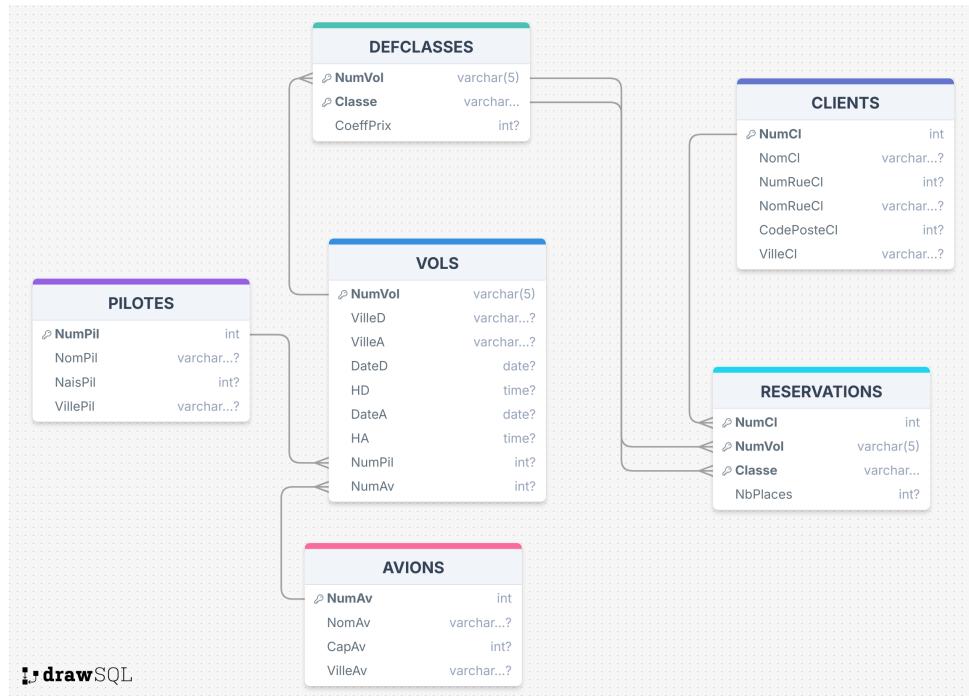


FIGURE 2.2 – Schéma de la base SQL

2.2.1 Étape 1 :Regroupement des informations de Vols, Avions et Pilotes

Dans le modèle SQL, pour récupérer toutes les informations d'un vol, il était nécessaire de joindre les tables 'VOLS', 'AVIONS' et 'PILOTES', car chaque table stockait les données d'une entité spécifique. Par exemple, la table 'VOLS' contenait les informations sur le vol, mais il fallait ensuite faire une jointure avec 'AVIONS' pour connaître les détails de l'avion, et avec 'PILOTES' pour obtenir les informations du pilote.

En NoSQL, nous avons choisi de regrouper toutes ces informations dans un seul document au sein de la collection 'Vols'. Cette approche permet de conserver toutes les données pertinentes pour un vol dans un seul document JSON.

```
Vol = {
    _id : 'V001',
    VilleD : 'Paris',
    VilleA : 'New York',
    DateD : '2024-12-01',
    HD : '08:00',
    DateA : '2024-12-01',
    HA : '14:00',
    Avion : { ... },
    Pilote : { ... }
}
```

Ce choix a été motivé par plusieurs raisons :

- **Élimination des jointures** : En SQL, récupérer ces informations nécessitait de multiples jointures. En NoSQL, en regroupant ces informations dans un seul document, nous réduisons considérablement le coût

des requêtes.

- **Accès optimisé aux données** : Toutes les données relatives à un vol, un avion et un pilote étant dans un seul document, cela permet un accès plus rapide et plus direct.
- **Regroupement logique** : Les informations sur les vols, les avions et les pilotes sont souvent interrogées ensemble, il est donc logique de les stocker ensemble.

2.2.2 Étape 2 :Intégration des Classes de Vols

Dans la base SQL, les différentes classes de service ('Economy', 'Business') étaient stockées dans une table séparée appelée 'DEFCLASSES', et elles étaient liées à la table 'VOLS' par le numéro de vol. Cela nécessitait une jointure supplémentaire pour récupérer les informations des classes disponibles pour chaque vol.

Dans le modèle NoSQL, nous avons décidé d'inclure les classes disponibles directement dans une liste au sein du document 'Vol'. Chaque élément de cette liste contient la classe ('Economy', 'Business') ainsi que son coefficient de prix.

```
/* Classes disponibles pour le vol */
Classes : [
    {
        Classe : 'Economy',
        CoeffPrix : 1.0
    },
    {
        Classe : 'Business',
        CoeffPrix : 1.5
    }
]
```

Ce choix s'explique par les avantages suivants :

- **Simplification des requêtes** : En incluant les informations des classes directement dans le document 'Vol', nous évitons d'avoir à effectuer une jointure supplémentaire pour les récupérer.
- **Centralisation des données du vol** : Toutes les informations pertinentes concernant un vol (y compris les classes de service) sont désormais disponibles en une seule requête.
- **Flexibilité du modèle NoSQL** : Le format JSON nous permet de structurer ces informations sous forme de liste, ce qui correspond bien à la nature dynamique des classes de vol, et permet d'ajouter facilement de nouvelles classes si nécessaire.

2.2.3 Étape 3 :Gestion des Clients

Dans le modèle SQL, la table 'CLIENTS' stockait des informations sur les clients, notamment leur nom, adresse, et autres coordonnées. Les champs d'adresse (numéro de rue, nom de rue, code postal, ville) étaient représentés par plusieurs colonnes.

Dans la structure NoSQL, nous avons décidé d'imbriquer les informations d'adresse dans un sous-document appelé 'Adresse' pour chaque document 'Client'.

```

Client = {
    _id : 789,
    NomCl : 'Alice Dupont',
    Adresse : { ... },
    Email : 'alice.dupont@example.com',
    Telephone : '0123456789'
}

```

Ce choix a été fait pour les raisons suivantes :

- **Regroupement logique des informations** : En imbriquant les informations d'adresse dans un sous-document, nous améliorons la cohérence et la gestion des données.
- **Flexibilité** : Si des champs supplémentaires liés à l'adresse sont nécessaires à l'avenir (comme le pays ou la région), ils peuvent être ajoutés sans avoir à modifier toute la structure.

2.2.4 Étape 4 :Représentation des Réservations

Dans la base SQL, la table ‘RESERVATIONS’ établissait des relations entre les clients et les vols via des clés étrangères, reliant également les classes de service. Dans le modèle NoSQL, nous avons conservé ce concept de relation en utilisant des identifiants (‘VolId’ et ‘ClientId’) pour relier les documents ‘Reservation’ aux documents ‘Vol’ et ‘Client’.

```

Reservation = {
    _id : 'R001',
    VolId : 'V001',
    ClientId : 789,
    NbPlaces : 2,
    Classe : 'Economy'
}

```

Nous avons fait ce choix pour les raisons suivantes :

- **Conservation des relations critiques** : Même dans un modèle NoSQL, certaines relations doivent être maintenues. Ici, nous utilisons des identifiants pour lier les réservations aux vols et aux clients sans dupliquer inutilement les données.
- **Économie de stockage** : En stockant uniquement les références aux documents ‘Vol’ et ‘Client’, nous évitons de dupliquer les informations des vols et des clients dans chaque réservation.

2.3 Conversion des données TXT en JSON

Le script A.5 montre la création de 3 documents json à partir de nos 6 tables, ces tables peuvent être intégrées facilement à Redis en ayant comme index le nom de la collection et indice (id) et pour mongo cela peut être représenté en trois collections chacune avec ses documents. et chaque document a un id unique.

2.3.1 Insertion des documents dans Redis

Pour insérer les documents dans Redis nous utiliserons le script A.6 qui charge les données JSON, les insère dans Redis et affiche un message de succès.

la figure 2.3 représente la sortie de la commande `redis-cli` qui affiche les données insérées dans Redis.

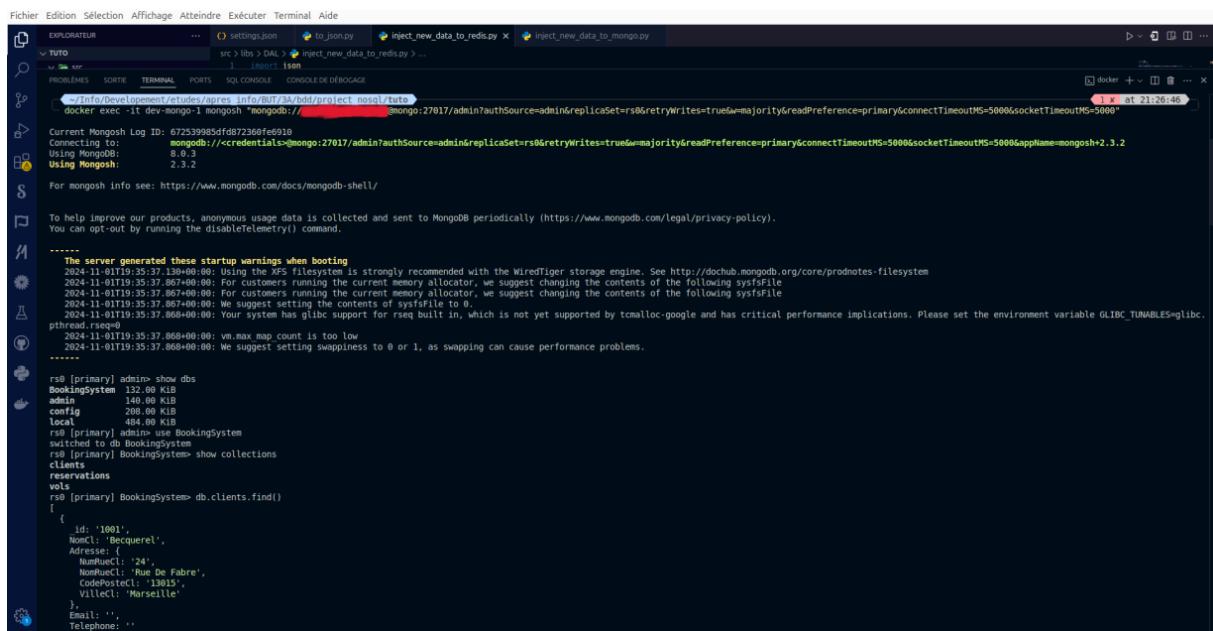
```
Fichier Edition Sélection Affichage Atteindre Exécuter Terminal Aide
EXPLORATEUR ... settings.json to_json.py inject_new_data_to_redis.py x inject_new_data_to_mongo.py
src > lib3 > DAL > inject_new_data_to_redis.py ...
TUTO
PROBLÈMES SORTIE TERMINAL PORTS SOI CONSOLE CONSOLDE DÉBOUTAGE
Vill eClt: docker exec -it dev-redis-1 redis-cli at 21:33:43
127.0.0.1:6379> AUTH [REDACTED]
OK
'Nant es' } 127.0.0.1:6379> keys *
1) "vol:V037"
2) "vol:V211"
3) "reservation:R025"
4) "vol:V220"
5) "vol:V919"
6) "vol:V38"
7) "vol:V869"
8) "vol:V622"
9) "vol:V688"
10) "vol:V131"
11) "reservation:R043"
12) "vol:V518"
13) "client:1030"
14) "vol:V927"
15) "vol:V609"
16) "vol:V614"
17) "client:1001"
18) "vol:V920"
19) "vol:V623"
20) "vol:V599"
21) "vol:V805"
22) "client:1007"
23) "client:1025"
24) "vol:V605"
25) "client:1028"
26) "vol:V214"
27) "reservation:R047"
28) "vol:V625"
29) "vol:V606"
30) "vol:V606"
NomR 31) "vol:V180"
usClt: 32) "vol:V146"
Res 33) "vol:V535"
De M 34) "vol:V511"
onaco 35) "reservation:R004"
', 36) "reservation:R013"
Code 37) "vol:V621"
Poste 38) "vol:V613"
Cl: 39) "vol:V925"
34000 40) "vol:V686"
41) "vol:V627"
42) "reservation:R017"
43) "reservation:R030"
Vill 44) "vol:V157"
eClt: 45) "vol:V602"
```

FIGURE 2.3 – Insertion des documents dans Redis

2.3.2 Insertion des documents dans MongoDB

Pour insérer les documents dans MongoDB nous utiliserons le script A.7 qui charge les données JSON, les insère dans MongoDB et affiche un message de succès.

la figure 2.4 représente la sortie de la commande mongo qui affiche les données insérées dans MongoDB.



The screenshot shows a terminal window with several tabs open. The active tab displays MongoDB shell commands and logs. The logs at the top show a connection from a Docker container to a MongoDB instance at port 27017. The logs include startup warnings about memory allocation and sysfs support. Below the logs, the user runs commands to switch to the 'BookingSystem' database and list collections ('clients', 'reservations', 'vols'). Finally, a query is run against the 'clients' collection, returning a single document.

```

Current Mongosh Log ID: 672539905d5f672360ffef210
Connecting to: mongodb://<credentials>@mongo:27017/admin?authSource=admin&replicaSet=rs0&retryWrites=true&w=majority&readPreference=primary&connectTimeoutMS=5000&socketTimeoutMS=5000
Using Mongosh: 2.3.2
Using Mongosh: 2.3.2

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry command.

-----
The server generated these startup warnings when booting
2024-11-01T19:35:37.867+00:00: It is strongly recommended to use the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem
2024-11-01T19:35:37.867+00:00: For customers running the current memory allocator, we suggest changing the contents of the following sysfsFile
2024-11-01T19:35:37.867+00:00: For customers running the current memory allocator, we suggest changing the contents of the following sysfsFile
2024-11-01T19:35:37.867+00:00: We suggest setting the contents of sysfsFile to 0.
2024-11-01T19:35:37.868+00:00: Your system has glibc support for rseq built in, which is not yet supported by tcmalloc-google and has critical performance implications. Please set the environment variable GLIBC_TUNABLES=glibc.
2024-11-01T19:35:37.868+00:00: We suggest setting swappiness to 0 or 1, as swapping can cause performance problems.

-----
rs0 [primary] admin> show dbs
BookingSystem 132.00 KIB
admin 140.00 KIB
local 200.00 KIB
local 484.00 KIB
rs0 [primary] admin> use BookingSystem
switched to db BookingSystem
rs0 [primary] BookingSystem> show collections
clients
reservations
vols
rs0 [primary] BookingSystem> db.clients.find()
{
    "_id": "1001",
    "NomCle": "Becquerel",
    "Adresse": "24 Rue de la République",
    "NumRueCle": "24",
    "NomRueCle": "Rue De Fabre",
    "CodePostalCle": "13015",
    "VilleCle": "Marseille",
    "Email": "",
    "Telephone": ""
}

```

FIGURE 2.4 – Insertion des documents dans MongoDB

Chapitre 3

Résultats

Les résultats de la migration sont présentés sous forme de documents JSON pour chaque entité, illustrant les avantages de la dénormalisation et les structures simplifiées obtenues. On peut visualiser les résultats en effectuant des requêtes vers la base de données NoSQL et donc au JSON car le format JSON est utilisé dans redis et mongo en tant que document.

Pour illustrer les avantages de la dénormalisation, nous avons utilisé des exemples de requêtes et de jointures pour montrer comment les données peuvent être manipulées et agrégées de manière plus efficace. Ces exemples sont présentés dans la section suivante.

3.1 RéPLICATION DES RÉSULTATS

Avec MongoDB, nous avons mis en place un système de réPLICATION entre trois serveurs pour garantir un accès continu à nos données. Cela permet aussi de réduire les temps de chargement des données, car les données sont chargées de manière asynchrone sur les serveurs. On peut voir dans A.2 la configuration des serveurs MongoDB et dans A.3 le script de réPLICATION qui montre comment on ajoute un serveur supplémentaire à la réPLICATION en tant que membres.

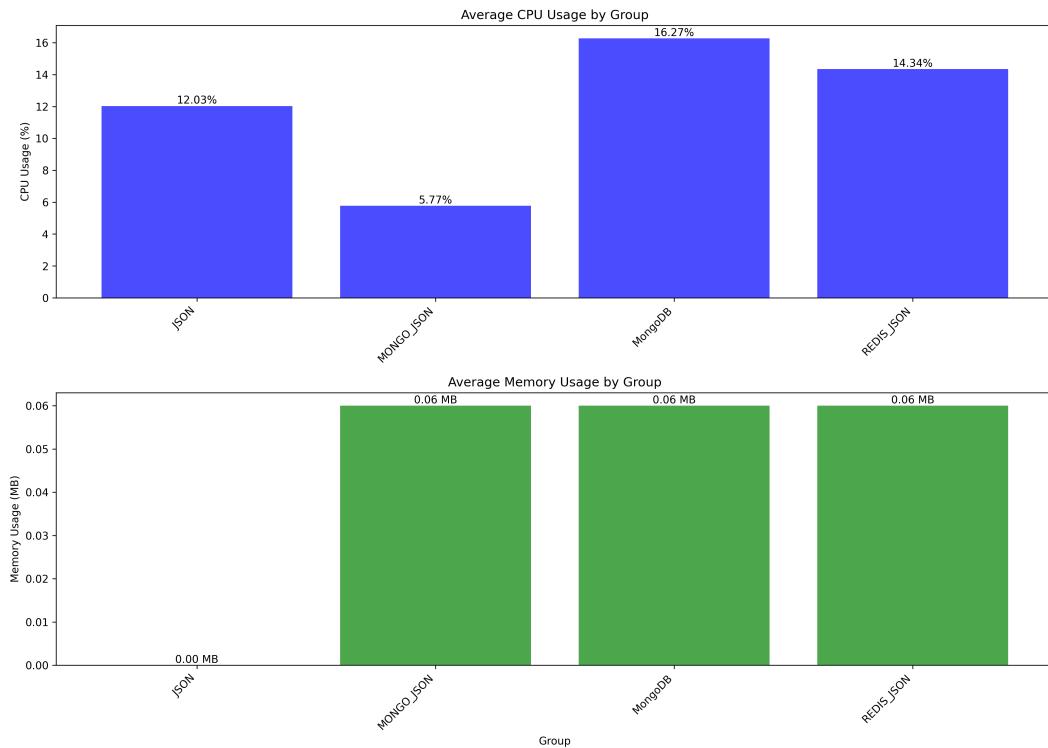
3.2 EXEMPLE DE RÉQUÊTES ET DE JOINTURES

En ayant convertit nos tables SQL en JSON, nous pouvons normaliser les requêtes et les jointures pour manipuler les données de manière plus efficace que les données viennent d'un fichier JSON, récupéré par redis ou par mongo.

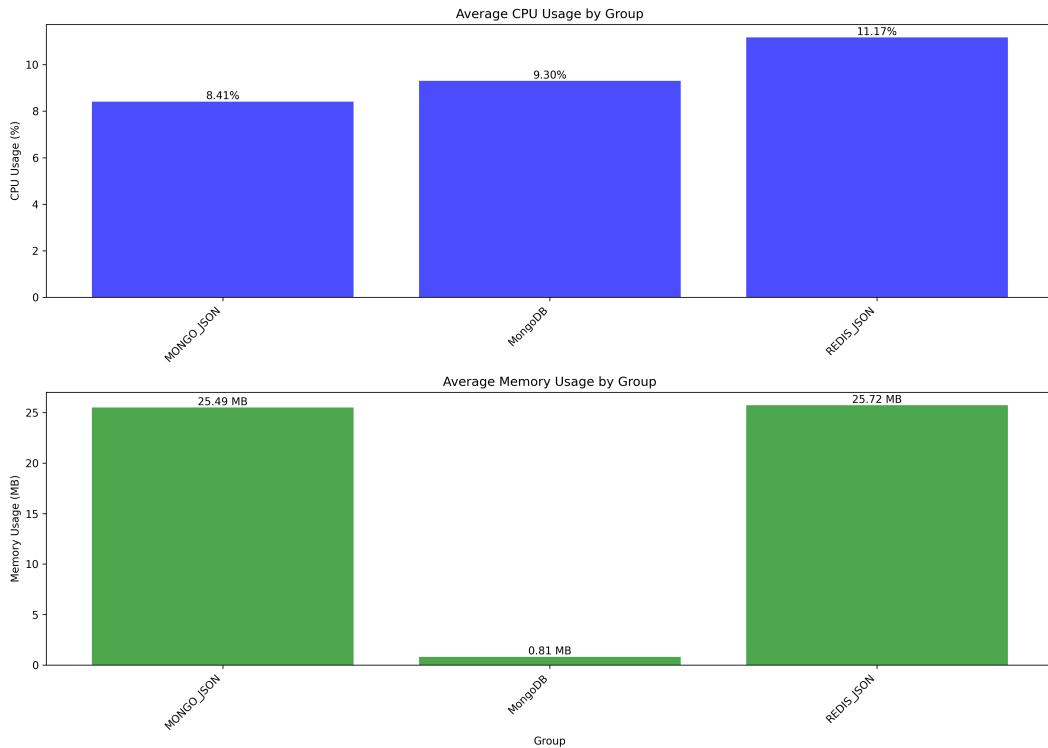
On peut voir dans A.8 les résultats des requêtes avec les données d'origine et le résultat des requêtes est le même pour Redis, MongoDB et MONGO_JSON.

3.3 Utilisation des ressources matérielles

Voici les ressources matérielles utilisées pour le développement de ce projet :



(a) Comparaison des ressources matérielles avec les données d'origine



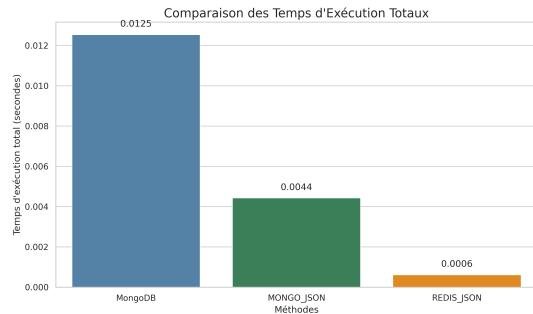
(b) Comparaison des ressources matérielles avec un grand volume de données

FIGURE 3.1 – Comparaison des ressources matérielles utilisées avec les données d'origine et avec un grand volume de données

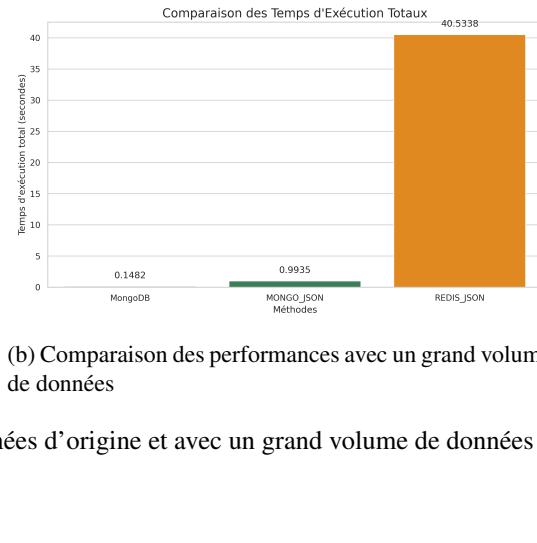
3.4 Comparaison des Performances

La figure 3.2a montre les performances des requêtes avec les données d'origine, tandis que la figure 3.2b montre les performances avec un grand volume de données.

On peut voir que pour un nombre de données faibles, le temps de requêtes avec Redis en important en JSON est plus faible que avec MongoDB en JSON et avec des recherches MongoDB (Pipelines). Cependant, avec un volume de données plus important, les performances sont plus élevées avec MongoDB (Pipelines) et en stocker les données Mongo en JSON.



(a) Comparaison des performances avec les données d'origine



(b) Comparaison des performances avec un grand volume de données

FIGURE 3.2 – Comparaison des performances avec les données d'origine et avec un grand volume de données

Chapitre 4

Discussion

Cette section analyse comment les résultats de la migration répondent aux objectifs fixés et aux besoins d'optimisation des performances. Nous avons mesuré les performances de chaque base de données à travers plusieurs requêtes (profiling détaillé en annexe A.8).

4.1 Comparaison des Performances

Les résultats montrent que Redis offre des temps de réponse plus rapides pour les opérations de lecture, tandis que MongoDB est plus adapté pour les requêtes complexes grâce à son système de stockage JSON natif.

TABLE 4.1 – Comparaison des performances entre MongoDB, MONGO_JSON et REDIS_JSON

Fonction	Temps MongoDB	Temps MONGO_JSON	Temps REDIS_JSON	Diff MONGO_JSON	Diff REDIS_JSON	% Diff MONGO_JSON	% Diff REDIS_JSON
get_flights_by_pilot	0.001110	0.000065	0.000002	-0.001046	-0.001109	-94.17%	-99.84%
get_reservations_by_client	0.001186	0.000015	0.000003	-0.001171	-0.001183	-98.75%	-99.74%
get_clients_by_flight	0.000767	0.000010	0.000003	-0.000757	-0.000764	-98.68%	-99.66%
get_vols_by_departure_city	0.001531	0.000080	0.000003	-0.001451	-0.001529	-94.76%	-99.81%
load_data_from_mongo	N/A	0.003973	N/A	N/A	N/A	N/A	N/A
get_clients_by_pilot	0.005947	0.000136	0.000021	-0.005811	-0.005927	-97.71%	-99.65%
load_data_from_redis	N/A	N/A	0.000578	N/A	N/A	N/A	N/A
get_arrival_cities	0.000921	0.000015	0.000002	-0.000906	-0.000919	-98.38%	-99.81%
get_pilotes	0.000541	0.000103	0.000005	-0.000437	-0.000535	-80.88%	-99.01%
get_arrival_city_by_id	0.000541	0.000034	0.000002	-0.000508	-0.000540	-93.73%	-99.64%
Total	0.012545	0.004432	0.000618	-0.008113	-0.011927	-64.67%	-95.08%

TABLE 4.2 – Comparaison des performances avec un grand volume de données

Fonction	Temps MongoDB	Temps MONGO_JSON	Temps REDIS_JSON	Diff MONGO_JSON	Diff REDIS_JSON	% Diff MONGO_JSON	% Diff REDIS_JSON
get_reservations_by_client	0.000784	0.019903	0.019361	0.019119	0.018577	2439.85%	2370.68%
get_clients_by_flight	0.000775	0.018795	0.021336	0.018020	0.020561	2324.48%	2652.33%
get_vols_by_departure_city	0.000957	0.022559	0.021597	0.021602	0.020640	2257.32%	2156.82%
get_flights_by_pilot	0.002058	0.015452	0.015755	0.013394	0.013696	650.73%	665.44%
load_data_from_redis	N/A	N/A	40.389436	N/A	N/A	N/A	N/A
get_clients_by_pilot	0.043849	0.045735	0.052641	0.001886	0.008792	4.30%	20.05%
get_arrival_cities	0.099761	0.012911	0.013698	-0.086850	-0.086062	-87.06%	-86.27%
load_data_from_mongo	N/A	0.858132	N/A	N/A	N/A	N/A	N/A
Total	0.148184	0.993487	40.533824	0.845303	40.385640	570.44%	27253.76%

Les performances des bases de données **MongoDB**, **MONGO_JSON** et **REDIS_JSON** ont été évaluées en exécutant plusieurs fonctions clés du système. Les résultats sont présentés dans les Tableaux 4.1 et 4.2, et illustrés dans la Figure 3.2.

4.1.1 Performances avec le Jeu de Données d'Origine

Le Tableau 4.1 présente les temps d'exécution des différentes fonctions avec le jeu de données initial, qui contient un volume de données relativement faible. On constate que **REDIS_JSON** offre les temps de réponse les plus rapides pour la majorité des fonctions. Par exemple, la fonction `get_flights_by_pilot` est exécutée en 0.000002 secondes avec **REDIS_JSON**, contre 0.001110 secondes avec **MongoDB**, ce qui représente une amélioration de 99.84%.

De même, **MONGO_JSON** montre des améliorations significatives par rapport à **MongoDB** standard, avec des gains de performance allant jusqu'à 98%. Ces améliorations sont dues à l'utilisation d'un format JSON dénormalisé, qui réduit le besoin de jointures et permet un accès plus rapide aux données.

4.1.2 Performances avec un Grand Volume de Données

Le Tableau 4.2 illustre les performances des mêmes fonctions, mais cette fois avec un grand volume de données. Dans ce scénario, les performances de **REDIS_JSON** se dégradent considérablement pour certaines fonctions. Par exemple, la fonction `load_data_from_redis` prend 40.389436 secondes, ce qui est nettement supérieur aux temps observés avec **MongoDB**.

En revanche, **MongoDB** démontre une meilleure scalabilité. Les temps d'exécution restent relativement stables malgré l'augmentation du volume de données. La fonction `get_clients_by_pilot` est exécutée en 0.043849 secondes avec **MongoDB**, contre 0.052641 secondes avec **REDIS_JSON**, indiquant une meilleure performance pour les requêtes complexes sur de grandes bases de données.

4.1.3 Interprétation des Résultats

Ces résultats mettent en évidence plusieurs points clés :

- **REDIS_JSON** est extrêmement performant pour les opérations de lecture simples et les petits volumes de données grâce à son stockage en mémoire. Cependant, il montre des limitations en termes de scalabilité lorsque le volume de données augmente.
- **MongoDB**, bien que légèrement moins performant sur de petites requêtes, offre une meilleure gestion des grands volumes de données et des requêtes complexes, grâce à sa structure de stockage optimisée et sa capacité à gérer des index efficaces.
- **MONGO_JSON** combine les avantages de MongoDB avec une structure de données dénormalisée, offrant de bonnes performances sur des volumes de données moyens, mais peut être limité par rapport à MongoDB standard pour des données massives.

La Figure 3.2 illustre visuellement ces tendances, montrant comment les temps d'exécution évoluent en fonction du volume de données pour chaque base de données. On y observe que si **REDIS_JSON** est le plus rapide sur des petits volumes, **MongoDB** devient plus performant à mesure que le volume de données augmente.

4.1.4 Conclusion sur le Choix des Bases de Données

Le choix entre **MongoDB** et **REDIS_JSON** dépend fortement des besoins spécifiques du projet :

- Si l'application nécessite des temps de réponse ultra-rapides pour des opérations simples sur de petits volumes de données, **REDIS_JSON** est recommandé.
- Pour des applications manipulant de grands volumes de données avec des requêtes complexes, **MongoDB** offre une meilleure performance globale et une scalabilité accrue.

Il est donc essentiel de prendre en compte le volume de données et la nature des requêtes lors de la sélection de la base de données appropriée pour une application donnée.

4.2 Comparaison des Ressources Matérielles

La figure 3.1 illustre les ressources matérielles utilisées pour le développement de ce projet, en comparant les performances des requêtes avec les données d'origine et avec un grand volume de données. On y observe que les ressources utilisées sont basses pour des requêtes avec mongo pour un volume de données faible et pour un volume de données important, les performances sont plus élevées avec des requêtes complexes et avec des données massives.

4.3 Limites et Améliorations Possibles

En implémentant Redis et mongo on peut voir que les deux ont certaines limites en fonction du nombre de données présents dans la base de données. Une application peut contenir les 2 types de base de données mais ne contenant pas les mêmes données.

Par exemple : Si on prend en exemple notre cas d'étude pour les vols... on peut mettre tout ça dans notre base mongo car on aura un nombre très important de données dans la base et si on imagine un système d'authentification à votre système de gestion de vols, on pourra stocker les utilisateurs dans mongo et les sessions d'authentification dans redis car on a un faible stockage de données stockées et redis permet de stocker et faire des requêtes pour une petite gestion de mémoire pour ne pas allourdir le système.

Bien que NoSQL ait permis une flexibilité accrue, certains cas d'utilisation nécessitent encore une réflexion pour l'optimisation des écritures massives. On remarque qu'actuellement entre le SQL et le NoSQL on trouve encore certaines limites, faut essayer de voir un nouveau type de base de données pour répondre à ces limites comme le graph database ou le NewSQL avec CockroachDB.

Chapitre 5

Conclusion

La migration de bases de données relationnelles SQL vers des solutions NoSQL, en utilisant notamment Redis et MongoDB, s'est avérée être une stratégie efficace pour répondre aux besoins croissants de flexibilité et de performance dans la gestion des données. Ce projet a mis en évidence les avantages significatifs qu'offrent ces technologies pour le stockage et la manipulation de données semi-structurées.

Les principaux résultats obtenus sont les suivants :

- **Optimisation des Performances :** Les tests de performance ont démontré que Redis offre des temps de réponse extrêmement rapides pour les opérations de lecture simples sur de petits volumes de données, grâce à son stockage en mémoire. MongoDB, quant à lui, assure une meilleure performance et une scalabilité accrue pour les requêtes complexes et les grands volumes de données, grâce à son système de gestion de documents JSON natif.
- **Flexibilité Accrue :** La transition vers des bases NoSQL a permis de dénormaliser les données et de les structurer de manière plus flexible. Cette approche facilite l'adaptation aux changements des modèles de données et réduit la complexité des opérations de manipulation des données.
- **Simplification des Requêtes :** En regroupant les informations auparavant réparties dans plusieurs tables SQL au sein de documents JSON unifiés, nous avons simplifié les requêtes nécessaires pour accéder aux données, réduisant ainsi le nombre de jointures et améliorant les temps de réponse.
- **Évolution de l'Architecture :** L'utilisation de Docker pour orchestrer les différents services a permis de mettre en place une architecture modulaire et facilement déployable. Cette approche facilite la gestion des environnements de développement et de production, tout en assurant une isolation des services.
- **Comparaison Objectivée des Bases de Données :** L'analyse comparative entre Redis et MongoDB a mis en lumière leurs forces et faiblesses respectives. Redis est idéal pour des applications nécessitant des accès ultra-rapides à des données volatiles, tandis que MongoDB est plus adapté pour des applications nécessitant une persistance des données et des requêtes complexes.

Ces résultats confirment que l'utilisation combinée de Redis et MongoDB peut offrir une solution robuste et performante pour des applications modernes, où la flexibilité et la performance sont essentielles.

Limites du Projet

Malgré les avantages constatés, certaines limitations ont été identifiées :

- **Gestion des Transactions** : Les bases NoSQL ne supportent pas les transactions complexes de la même manière que les bases SQL, ce qui peut poser des défis pour garantir l'intégrité des données dans certaines applications critiques.
- **Consistance des Données** : La flexibilité des schémas dans les bases NoSQL peut entraîner des incohérences si les données ne sont pas correctement validées au niveau de l'application.
- **Courbe d'Apprentissage** : La migration vers NoSQL nécessite une adaptation des compétences des développeurs et des administrateurs de bases de données, ainsi qu'une compréhension approfondie des nouveaux paradigmes de modélisation des données.

Perspectives Futures

Pour prolonger ce travail, plusieurs axes peuvent être explorés :

- **Mise en Place de Mécanismes de Validation** : Intégrer des schémas de validation au niveau de l'application ou utiliser des fonctionnalités offertes par les bases NoSQL pour assurer la cohérence des données.
- **Évaluation d'Autres Bases NoSQL** : Étudier l'intégration de bases telles que Cassandra, CouchDB ou Elasticsearch pour comparer leurs performances et leurs fonctionnalités avec celles de Redis et MongoDB.
- **Évaluation d'Autres Bases comme le NewSQL** :
- **Optimisation des Performances** : Mettre en œuvre des mécanismes de mise en cache avancés, comme Redis Cache, pour améliorer encore les temps de réponse des applications.
- **Sécurité et Gestion des Accès** : Explorer les options de sécurité offertes par les bases NoSQL, y compris l'authentification, l'autorisation et le chiffrement des données.
- **Scalabilité Horizontale** : Tester la mise en place de clusters Redis et MongoDB pour évaluer les performances en environnement distribué et la tolérance aux pannes.
- **Automatisation du Déploiement** : Utiliser des outils d'orchestration tels que Kubernetes pour automatiser le déploiement et la gestion des conteneurs Docker dans des environnements de production.

Mot de la Fin

Ce projet a permis de démontrer concrètement les bénéfices de la migration vers des bases de données NoSQL dans un contexte où la flexibilité et la performance sont primordiales. Les technologies étudiées offrent des perspectives prometteuses pour le développement d'applications modernes capables de gérer efficacement des volumes de données en constante augmentation.

En conclusion, le choix entre les bases de données relationnelles et les bases NoSQL doit être guidé par les besoins spécifiques de l'application, en tenant compte des contraintes de performance, de flexibilité et de scalabilité. La combinaison de plusieurs technologies, comme Redis et MongoDB, peut offrir une solution hybride adaptée aux exigences complexes des systèmes d'information actuels.

Annexe A

Annexes

A.1 Configuration Docker pour Redis

```
1 redis:
2   build:
3     context: ./docker/redis/.
4     dockerfile: Dockerfile.redis
5   ports:
6     - "${REDIS_PORT}:${REDIS_PORT}"
7   volumes:
8     - ./docker-data/redis/data:/data
9     - ./docker-data/redis/logs:/var/log/redis
10  restart: always
11  env_file:
12    - ./docker/redis/.env.redis
13  environment:
14    - REDIS_ARGS=--requirepass ${REDIS_PASSWORD}
15  healthcheck:
16    test: ["CMD", "redis-cli", "-a", "${REDIS_PASSWORD}", "ping"]
17    interval: 30s
18    timeout: 10s
19    retries: 5
```

Listing A.1 – Fichier de configuration Docker pour Redis

A.2 Configuration Docker pour MongoDB

```
1 # MongoDB Primary
2 mongo1:
3   build:
4     context: ./docker/mongo/
5     dockerfile: Dockerfile.mongo
6   ports:
7     - 27017:27017
8   restart: always
9   volumes:
10    - ./docker-data/mongo1/data:/data/db
```

```

11 env_file:
12   - ./docker/mongo/.env.mongo
13 healthcheck:
14   test: echo 'db.runCommand("ping").ok' | mongosh localhost:27017/test --quiet
15   interval: 30s
16   timeout: 10s
17   retries: 5
18   start_period: 20s
19
20 # MongoDB Replica Set
21 mongo2:
22   build:
23     context: ./docker/mongo/
24     dockerfile: Dockerfile.mongo
25   ports:
26     - 27018:27017
27   restart: always
28   volumes:
29     - ./docker-data/mongo2/data:/data/db
30   env_file:
31     - ./docker/mongo/.env.mongo
32   healthcheck:
33     test: echo 'db.runCommand("ping").ok' | mongosh localhost:27018/test --quiet
34     interval: 30s
35     timeout: 10s
36     retries: 5
37     start_period: 20s
38
39 # MongoDB Replica Set
40 mongo3:
41   build:
42     context: ./docker/mongo/
43     dockerfile: Dockerfile.mongo
44   ports:
45     - 27019:27017
46   restart: always
47   volumes:
48     - ./docker-data/mongo3/data:/data/db
49   env_file:
50     - ./docker/mongo/.env.mongo
51   healthcheck:
52     test: echo 'db.runCommand("ping").ok' | mongosh localhost:27019/test --quiet
53     interval: 30s
54     timeout: 10s
55     retries: 5
56     start_period: 20s

```

Listing A.2 – Fichier de configuration Docker pour MongoDB

A.3 Replica Set MongoDB

```

1 #!/bin/bash
2
3 echo 'Waiting for MongoDB to start...'
4 until mongosh --eval "db.adminCommand('ping')" --authenticationDatabase admin --username ${MONGO_INITDB_ROOT_USERNAME} --password ${MONGO_INITDB_ROOT_PASSWORD}; do

```

```

5  >&2 echo 'MongoDB is unavailable - sleeping'
6    sleep 5
7 done
8
9 echo 'Initializing replica set for Docker network...'
10 mongosh -u ${MONGO_INITDB_ROOT_USERNAME} -p ${MONGO_INITDB_ROOT_PASSWORD} --
11   authenticationDatabase admin --eval "rs.initiate({_id: 'rs0', members: [{ _id: 0, host: 'mongol:27017' }, { _id: 1, host: 'mongo2:27017' }, { _id: 2, host: 'mongo3:27017' }]})"
12 echo 'Replica set initialized.'

```

Listing A.3 – Replica Set MongoDB

A.4 Configuration Docker pour Python

```

1 poetry:
2   build:
3     context: ./docker/poetry/
4     dockerfile: Dockerfile.poetry
5     stdin_open: true
6     tty: true
7     volumes:
8       - .:/workspace
9     working_dir: /workspace
10    environment:
11      - POETRY_VIRTUALENVS_IN_PROJECT=true
12    command: [ "/bin/sh" ]

```

Listing A.4 – Fichier de configuration Docker pour Python

A.5 Crédation des Documents JSON

```

1 import json
2 import os
3
4 # Table de correspondance des colonnes
5 tableCorespondance = {
6   "AVIONS.txt": ["NumAv", "NomAv", "CapAv", "VilleAv"],
7   "CLIENTS.txt": ["NumCl", "NomCl", "NumRueCl", "NomRueCl", "CodePosteCl", "VilleCl"],
8   "DEFCLASSES.txt": ["NumVol", "Classe", "CoeffPrix"],
9   "PILOTES.txt": ["NumPil", "NomPil", "NaisPil", "VillePil"],
10  "RESERVATIONS.txt": ["NumCl", "NumVol", "Classe", "NbPlaces"],
11  "VOLS.txt": ["NumVol", "VilleD", "VilleA", "DateD", "HD", "DateA", "HA", "NumPil", "NumAv"]
12    ],
13 }
14
15 # Lire les fichiers .txt et les stocker dans un dictionnaire
16 dictAllJson = {}
17 for fileName in os.listdir("src/libs/db/txt"):
18   if fileName.endswith(".txt"):
19     file_path = os.path.join("src/libs/txt", fileName)
20     fields = tableCorespondance[fileName]

```

```

21     # Vérifier si la première colonne est une clé unique
22     if fileName in ["DEFCLASSES.txt", "RESERVATIONS.txt"]:
23         # Stocker les données dans une liste pour ces fichiers
24         dictAllJson[fileName] = []
25         with open(file_path, 'r') as fh:
26             for line in fh:
27                 description = list(line.strip().split("\t"))
28                 if len(description) < len(fields):
29                     # Gérer les lignes avec des colonnes manquantes
30                     continue
31                 data_entry = {}
32                 for i, categorie in enumerate(fields):
33                     data_entry[categorie] = description[i]
34                 dictAllJson[fileName].append(data_entry)
35             else:
36                 # Utiliser un dictionnaire avec la première colonne comme clé
37                 dictAllJson[fileName] = {}
38                 with open(file_path, 'r') as fh:
39                     for line in fh:
40                         description = list(line.strip().split("\t"))
41                         if len(description) < len(fields):
42                             # Gérer les lignes avec des colonnes manquantes
43                             continue
44                         key = description[0]
45                         data_entry = {}
46                         for i, categorie in enumerate(fields[1:], start=1):
47                             data_entry[categorie] = description[i]
48                         dictAllJson[fileName][key] = data_entry
49
50     # Construire la liste des vols
51     vols_list = []
52     for vol_id, vol_data in dictAllJson["VOLS.txt"].items():
53         vol_dict = {"_id": vol_id}
54
55         # Copier les champs du vol
56         vol_fields_mapping = {
57             "VilleD": "VilleD",
58             "VilleA": "VilleA",
59             "DateD": "DateD",
60             "HD": "HD",
61             "DateA": "DateA",
62             "HA": "HA",
63         }
64         for src_field, dest_field in vol_fields_mapping.items():
65             vol_dict[dest_field] = vol_data.get(src_field, "")
66
67         # Ajouter l'avion
68         num_av = vol_data.get("NumAv", "")
69         avion_data = dictAllJson["AVIONS.txt"].get(num_av, {})
70         vol_dict["Avion"] = {
71             "NumAv": num_av,
72             "NomAv": avion_data.get("NomAv", ""),
73             "CapAv": avion_data.get("CapAv", ""),
74             "VilleAv": avion_data.get("VilleAv", ""),
75         }
76
77         # Ajouter le pilote
78         num_pil = vol_data.get("NumPil", "")
79         pilote_data = dictAllJson["PILOTES.txt"].get(num_pil, {})

```

```

80     vol_dict["Pilote"] = {
81         "NumPil": num_pil,
82         "NomPil": pilote_data.get("NomPil", ""),
83         "NaisPil": pilote_data.get("NaisPil", ""),
84         "VillePil": pilote_data.get("VillePil", ""),
85     }
86
87     # Ajouter les classes
88     vol_dict["Classes"] = []
89     for classe in dictAllJson["DEFCLASSES.txt"]:
90         if classe.get("NumVol") == vol_id:
91             vol_dict["Classes"].append({
92                 "Classe": classe.get("Classe", ""),
93                 "CoeffPrix": classe.get("CoeffPrix", ""),
94             })
95
96     vols_list.append(vol_dict)
97
98     # Construire la liste des clients
99     clients_list = []
100    for client_id, client_data in dictAllJson["CLIENTS.txt"].items():
101        client_dict = {
102            "_id": client_id,
103            "NomCl": client_data.get("NomCl", ""),
104            "Adresse": {
105                "NumRueCl": client_data.get("NumRueCl", ""),
106                "NomRueCl": client_data.get("NomRueCl", ""),
107                "CodePosteCl": client_data.get("CodePosteCl", ""),
108                "VilleCl": client_data.get("VilleCl", ""),
109            },
110            "Email": "",           # Champs supplémentaires à remplir si nécessaire
111            "Telephone": "",      # Champs supplémentaires à remplir si nécessaire
112        }
113        clients_list.append(client_dict)
114
115    # Construire la liste des réservations
116    reservations_list = []
117    reservation_id_counter = 1
118    for reservation_data in dictAllJson["RESERVATIONS.txt"]:
119        reservation_dict = {
120            "_id": f"R{str(reservation_id_counter).zfill(3)}",
121            "VolId": reservation_data.get("NumVol", ""),
122            "ClientId": reservation_data.get("NumCl", ""),
123            "NbPlaces": reservation_data.get("NbPlaces", ""),
124            "Classe": reservation_data.get("Classe", ""),
125        }
126        reservations_list.append(reservation_dict)
127        reservation_id_counter += 1
128
129    # Écrire les données dans des fichiers JSON séparés
130    with open("src/libs/db/json/vols.json", "w") as f:
131        json.dump(vols_list, f, indent=2, ensure_ascii=False)
132
133    with open("src/libs/db/json/clients.json", "w") as f:
134        json.dump(clients_list, f, indent=2, ensure_ascii=False)
135
136    with open("src/libs/db/json/reservations.json", "w") as f:
137        json.dump(reservations_list, f, indent=2, ensure_ascii=False)
138

```

```
139     print("Données extraites avec succès.")
```

Listing A.5 – Script de création des documents JSON

A.6 Insertion des documents dans Redis

```
1 import json
2 import sys, os
3
4 # Définir le chemin
5 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))
6
7 # Définir le chemin vers le dossier contenant les fichiers JSON
8 SRC = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..'))
9 DATA_DIR = os.path.join(SRC, 'libs', 'db', 'json')
10
11 from config.connectionRedis import connect_redis as connect
12
13 # Charger les données JSON
14 with open(os.path.join(DATA_DIR, 'vols.json'), 'r', encoding='utf-8') as f:
15     vols = json.load(f)
16 with open(os.path.join(DATA_DIR, 'clients.json'), 'r', encoding='utf-8') as f:
17     clients = json.load(f)
18 with open(os.path.join(DATA_DIR, 'reservations.json'), 'r', encoding='utf-8') as f:
19     reservations = json.load(f)
20
21 r = connect()
22
23 def clear_redis_database():
24     r.flushdb()
25
26 clear_redis_database()
27
28 # Insérer les vols dans Redis
29 for vol in vols:
30     vol_id = vol['_id']
31     # Stocker le vol sous la clé 'vol:{vol_id}'
32     r.set(f'vol:{vol_id}', json.dumps(vol))
33
34 # Insérer les clients dans Redis
35 for client in clients:
36     client_id = client['_id']
37     # Stocker le client sous la clé 'client:{client_id}'
38     r.set(f'client:{client_id}', json.dumps(client))
39
40 # Insérer les réservations dans Redis
41 for reservation in reservations:
42     reservation_id = reservation['_id']
43     # Stocker la réservation sous la clé 'reservation:{reservation_id}'
44     r.set(f'reservation:{reservation_id}', json.dumps(reservation))
45
46 print("Données insérées avec succès dans Redis.")
```

Listing A.6 – Script d'insertion des documents dans Redis

A.6.1 Insertion des documents dans MongoDB

```

1 import sys, os, json
2
3 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))
4
5 from config.connectionMongo import connect_mongodb as connect
6
7 # Définir le chemin vers le dossier contenant les fichiers JSON
8 SRC = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..'))
9 DATA_DIR = os.path.join(SRC, 'libs', 'DAL', 'db', 'json')
10
11 # Charger les données JSON
12 with open(os.path.join(DATA_DIR, 'vols.json'), 'r', encoding='utf-8') as f:
13     vols = json.load(f)
14 with open(os.path.join(DATA_DIR, 'clients.json'), 'r', encoding='utf-8') as f:
15     clients = json.load(f)
16 with open(os.path.join(DATA_DIR, 'reservations.json'), 'r', encoding='utf-8') as f:
17     reservations = json.load(f)
18
19 # Se connecter à MongoDB
20 client = connect()
21
22 db = client.get_database('BookingSystem')
23
24 # Créer ou obtenir les collections
25 vols_collection = db.get_collection('vols')
26 clients_collection = db.get_collection('clients')
27 reservations_collection = db.get_collection('reservations')
28
29 # Effacer les collections existantes pour éviter les doublons lors de ré-exécutions
30 vols_collection.delete_many({})
31 clients_collection.delete_many({})
32 reservations_collection.delete_many({})
33
34 # Insérer les documents JSON dans chaque collection respective
35 vols_collection.insert_many(vols)
36 clients_collection.insert_many(clients)
37 reservations_collection.insert_many(reservations)
38
39 print("Les données ont été insérées avec succès dans MongoDB.")

```

Listing A.7 – Script d'insertion des documents dans MongoDB

A.7 Résultats des requêtes

```

1 [MongoDB] Function 'get_arrival_cities' executed in 0.024853 seconds
2 Liste des villes d'arrivée :
3 Marseille: 62
4 Amsterdam: 41
5 NewYork: 30
6 Nice: 29
7 Pekin: 28
8 Paris: 9
9
10 [MongoDB] Function 'get_arrival_city_by_id' executed in 0.002618 seconds

```

```
11 Ville d'arrivée du vol V519 : NewYork
12
13 Liste des pilotes :
14 [MongoDB] Function 'get_pilotes' executed in 0.002133 seconds
15 Delacroix
16 Delalande
17 Dubois
18 Dumas
19 Duparc
20 Duval
21 Leblanc
22 Ledru
23 Legrand
24 [MongoDB] Function 'get_pilotes' executed in 0.001526 seconds
25 Nombre de pilotes: 9
26 Vols au départ de Marseille :
27 [MongoDB] Function 'get_vols_by_departure_city' executed in 0.005160 seconds
28 V101 -> Amsterdam
29 V102 -> Amsterdam
30 V103 -> Amsterdam
31 V104 -> Amsterdam
32 V105 -> Amsterdam
33 V106 -> Amsterdam
34 V107 -> Amsterdam
35 V108 -> Amsterdam
36 V109 -> Amsterdam
37 V110 -> Amsterdam
38 V111 -> Amsterdam
39 V112 -> Amsterdam
40 V113 -> Amsterdam
41 V114 -> Amsterdam
42 V115 -> Amsterdam
43 V116 -> Amsterdam
44 V117 -> Amsterdam
45 V118 -> Amsterdam
46 V119 -> Amsterdam
47 V120 -> Amsterdam
48 V180 -> Amsterdam
49 V181 -> Amsterdam
50 V182 -> Amsterdam
51 V183 -> Amsterdam
52 V184 -> Amsterdam
53 V185 -> Amsterdam
54 V186 -> Amsterdam
55 V187 -> Amsterdam
56 V188 -> Amsterdam
57 V189 -> Amsterdam
58 V190 -> Amsterdam
59 V191 -> Amsterdam
60 V192 -> Amsterdam
61 V193 -> Amsterdam
62 V194 -> Amsterdam
63 V195 -> Amsterdam
64 V196 -> Amsterdam
65 V197 -> Amsterdam
66 V198 -> Amsterdam
67 V199 -> Amsterdam
68 V200 -> Amsterdam
69 V350 -> Paris
```

```
70 V351 -> Paris
71 V501 -> NewYork
72 V502 -> NewYork
73 V503 -> NewYork
74 V504 -> NewYork
75 V505 -> NewYork
76 V506 -> NewYork
77 V507 -> NewYork
78 V508 -> NewYork
79 V509 -> NewYork
80 V510 -> NewYork
81 V511 -> NewYork
82 V512 -> NewYork
83 V513 -> NewYork
84 V514 -> NewYork
85 V515 -> NewYork
86 V516 -> NewYork
87 V517 -> NewYork
88 V518 -> NewYork
89 V519 -> NewYork
90 V520 -> NewYork
91 V521 -> NewYork
92 V522 -> NewYork
93 V523 -> NewYork
94 V524 -> NewYork
95 V525 -> NewYork
96 V526 -> NewYork
97 V527 -> NewYork
98 V528 -> NewYork
99 V529 -> NewYork
100 V530 -> NewYork
101 V681 -> Pekin
102 V682 -> Pekin
103 V683 -> Pekin
104 V684 -> Pekin
105 V685 -> Pekin
106 V686 -> Pekin
107 V687 -> Pekin
108 V688 -> Pekin
109 V689 -> Pekin
110 V690 -> Pekin
111 V691 -> Pekin
112 V692 -> Pekin
113 V693 -> Pekin
114 V694 -> Pekin
115
116 [MongoDB] Function 'get_reservations_by_client' executed in 0.007681 seconds
117 Réservations pour le client 1001 :
118 Réservation ID: R001, Vol ID: V690, Classe: Business, NbPlaces: 3
119 Détails du vol: De Marseille à Pekin le 19/04/07
120
121 Réservation ID: R016, Vol ID: V141, Classe: Business, NbPlaces: 3
122 Détails du vol: De Amsterdam à Marseille le 1/04/07
123
124 Réservation ID: R021, Vol ID: V790, Classe: Touriste, NbPlaces: 1
125 Détails du vol: De Metz à Marseille le 20/04/07
126
127 Réservation ID: R022, Vol ID: V150, Classe: Touriste, NbPlaces: 1
128 Détails du vol: De Amsterdam à Marseille le 10/04/07
```

129
130 Réservation ID: R033, Vol ID: V601, Classe: Economique, NbPlaces: 6
131 Détails du vol: De Ajaccio à Marseille le 1/04/07
132
133 [MongoDB] Function '`get_clients_by_flight`' executed in 0.001317 seconds
134 Clients sur le vol V519 :
135 [MongoDB] Function '`get_flights_by_pilot`' executed in 0.002763 seconds
136 Vols opérés par le pilote Leblanc :
137 Vol ID: V101, De Marseille à Amsterdam le 1/04/07
138 Vol ID: V102, De Marseille à Amsterdam le 2/04/07
139 Vol ID: V103, De Marseille à Amsterdam le 3/04/07
140 Vol ID: V104, De Marseille à Amsterdam le 4/04/07
141 Vol ID: V105, De Marseille à Amsterdam le 5/04/07
142 Vol ID: V106, De Marseille à Amsterdam le 6/04/07
143 Vol ID: V107, De Marseille à Amsterdam le 7/04/07
144 Vol ID: V108, De Marseille à Amsterdam le 8/04/07
145 Vol ID: V109, De Marseille à Amsterdam le 9/04/07
146 Vol ID: V110, De Marseille à Amsterdam le 10/04/07
147 Vol ID: V111, De Marseille à Amsterdam le 11/04/07
148 Vol ID: V112, De Marseille à Amsterdam le 12/04/07
149 Vol ID: V113, De Marseille à Amsterdam le 13/04/07
150 Vol ID: V114, De Marseille à Amsterdam le 14/04/07
151 Vol ID: V115, De Marseille à Amsterdam le 15/04/07
152 Vol ID: V116, De Marseille à Amsterdam le 16/04/07
153 Vol ID: V117, De Marseille à Amsterdam le 17/04/07
154 Vol ID: V118, De Marseille à Amsterdam le 18/04/07
155 Vol ID: V119, De Marseille à Amsterdam le 19/04/07
156 Vol ID: V120, De Marseille à Amsterdam le 20/04/07
157 Vol ID: V141, De Amsterdam à Marseille le 1/04/07
158 Vol ID: V142, De Amsterdam à Marseille le 2/04/07
159 Vol ID: V143, De Amsterdam à Marseille le 3/04/07
160 Vol ID: V144, De Amsterdam à Marseille le 4/04/07
161 Vol ID: V145, De Amsterdam à Marseille le 5/04/07
162 Vol ID: V146, De Amsterdam à Marseille le 6/04/07
163 Vol ID: V147, De Amsterdam à Marseille le 7/04/07
164 Vol ID: V148, De Amsterdam à Marseille le 8/04/07
165 Vol ID: V149, De Amsterdam à Marseille le 9/04/07
166 Vol ID: V150, De Amsterdam à Marseille le 10/04/07
167 Vol ID: V151, De Amsterdam à Marseille le 11/04/07
168 Vol ID: V152, De Amsterdam à Marseille le 12/04/07
169 Vol ID: V153, De Amsterdam à Marseille le 13/04/07
170 Vol ID: V154, De Amsterdam à Marseille le 14/04/07
171 Vol ID: V155, De Amsterdam à Marseille le 15/04/07
172 Vol ID: V156, De Amsterdam à Marseille le 16/04/07
173 Vol ID: V157, De Amsterdam à Marseille le 17/04/07
174 Vol ID: V180, De Marseille à Amsterdam le 10/04/07
175 Vol ID: V181, De Marseille à Amsterdam le 11/04/07
176 Vol ID: V182, De Marseille à Amsterdam le 12/04/07
177 Vol ID: V183, De Marseille à Amsterdam le 13/04/07
178 Vol ID: V184, De Marseille à Amsterdam le 14/04/07
179 Vol ID: V185, De Marseille à Amsterdam le 15/04/07
180 Vol ID: V186, De Marseille à Amsterdam le 16/04/07
181 Vol ID: V187, De Marseille à Amsterdam le 17/04/07
182 Vol ID: V188, De Marseille à Amsterdam le 18/04/07
183 Vol ID: V189, De Marseille à Amsterdam le 19/04/07
184 Vol ID: V190, De Marseille à Amsterdam le 20/04/07
185 Vol ID: V191, De Marseille à Amsterdam le 21/04/07
186 Vol ID: V192, De Marseille à Amsterdam le 22/04/07
187 Vol ID: V193, De Marseille à Amsterdam le 23/04/07

```

188 Vol ID: V194, De Marseille à Amsterdam le 24/04/07
189 Vol ID: V195, De Marseille à Amsterdam le 25/04/07
190 Vol ID: V196, De Marseille à Amsterdam le 26/04/07
191 Vol ID: V197, De Marseille à Amsterdam le 27/04/07
192 Vol ID: V198, De Marseille à Amsterdam le 28/04/07
193 Vol ID: V199, De Marseille à Amsterdam le 29/04/07
194 Vol ID: V200, De Marseille à Amsterdam le 30/04/07
195
196 [MongoDB] Function 'get_clients_by_pilot' executed in 0.011204 seconds
197 Clients ayant des réservations sur les vols du pilote Leblanc :
198 Client ID: 1031, Nom: Bohr, Vol ID: V101, NbPlaces: 2, Classe: Business
199
200 Client ID: 1027, Nom: Lorentz, Vol ID: V101, NbPlaces: 2, Classe: Business
201
202 Client ID: 1031, Nom: Bohr, Vol ID: V101, NbPlaces: 5, Classe: Touriste
203
204 Client ID: 1033, Nom: Dirac, Vol ID: V101, NbPlaces: 7, Classe: Touriste
205
206 Client ID: 1028, Nom: Lenard, Vol ID: V101, NbPlaces: 6, Classe: Touriste
207
208 Client ID: 1021, Nom: Perse, Vol ID: V101, NbPlaces: 6, Classe: Touriste
209
210 Client ID: 1002, Nom: Leblanc, Vol ID: V101, NbPlaces: 5, Classe: Touriste
211
212 Client ID: 1029, Nom: Planck, Vol ID: V101, NbPlaces: 7, Classe: Economique
213
214 Client ID: 1015, Nom: Rolland, Vol ID: V101, NbPlaces: 1, Classe: Economique
215
216 Client ID: 1006, Nom: Grignard, Vol ID: V101, NbPlaces: 4, Classe: Economique
217
218 Client ID: 1001, Nom: Becquerel, Vol ID: V141, NbPlaces: 3, Classe: Business
219
220 Client ID: 1008, Nom: Joliot, Vol ID: V141, NbPlaces: 2, Classe: Touriste
221
222 Client ID: 1011, Nom: Richet, Vol ID: V141, NbPlaces: 8, Classe: Economique
223
224 Client ID: 1020, Nom: Gide, Vol ID: V150, NbPlaces: 1, Classe: Business
225
226 Client ID: 1027, Nom: Lorentz, Vol ID: V150, NbPlaces: 1, Classe: Business
227
228 Client ID: 1001, Nom: Becquerel, Vol ID: V150, NbPlaces: 1, Classe: Touriste
229
230 Client ID: 1029, Nom: Planck, Vol ID: V150, NbPlaces: 7, Classe: Economique

```

Listing A.8 – Résultats des requêtes

```

1 import time
2 from functools import wraps
3 import csv
4
5 # Global list to store performance results
6 performance_results = []
7
8 def timing_decorator(method_group = None):

```

A.8 Profiling des Requêtes

```

9      """
10     A decorator to measure execution time of methods and log the results.
11
12     Args:
13         method_group (str): The group or class name (e.g., 'MongoDB' or 'JSON').
14
15     Returns:
16         function: The decorated function with added timing code.
17     """
18
19     def decorator(func):
20         @wraps(func)
21         def wrapper(*args, **kwargs):
22             nonlocal method_group
23             if method_group is None and args:
24                 self = args[0]
25                 method_group = getattr(self, 'method_group', 'Undefined')
26
27             start_time = time.perf_counter()
28
29             result = func(*args, **kwargs)
30
31             end_time = time.perf_counter()
32
33             execution_time = end_time - start_time
34
35             performance_results.append({
36                 'method_group': method_group,
37                 'function_name': func.__name__,
38                 'execution_time': execution_time
39             })
40             # Print the result
41             print(f"[{method_group}] Function '{func.__name__}' executed in {execution_time:.6f}
42                 seconds")
43             # Return the original function's result
44             return result
45         return wrapper
46     return decorator

```

Listing A.9 – Profiling des requêtes

A.9 Dépôt Git

Vous trouverez le dépôt Git de ce projet sur GitHub : GitHub - BookingSystem-app.

Pour pouvoir tester le projet vous pourrez suivre les étapes mises dans le fichier README.md.