

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	Exemple de requêtes et de jointures	10
3.2	Comparaison des Performances	10
4	Discussion	12
4.1	Comparaison des Performances	12
4.2	Limites et Améliorations Possibles	12
5	Conclusion	13

A Annexes	14
A.1 Configuration Docker pour Redis	14
A.2 Configuration Docker pour MongoDB	14
A.3 Configuration Docker pour Python	15
A.4 Création des Documents JSON	16
A.5 Insertion des documents dans Redis	18
A.5.1 Insertion des documents dans MongoDB	19
A.6 Profiling des Requêtes	20
A.7 Dépôt Git	20

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 performances avec les données d'origine et avec un grand volume de données . .	11

Liste des tableaux

4.1 Comparaison des performances entre MongoDB, MONGO_JSON et REDIS_JSON	12
4.2 Comparaison des performances avec un grand volume de données	12

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.

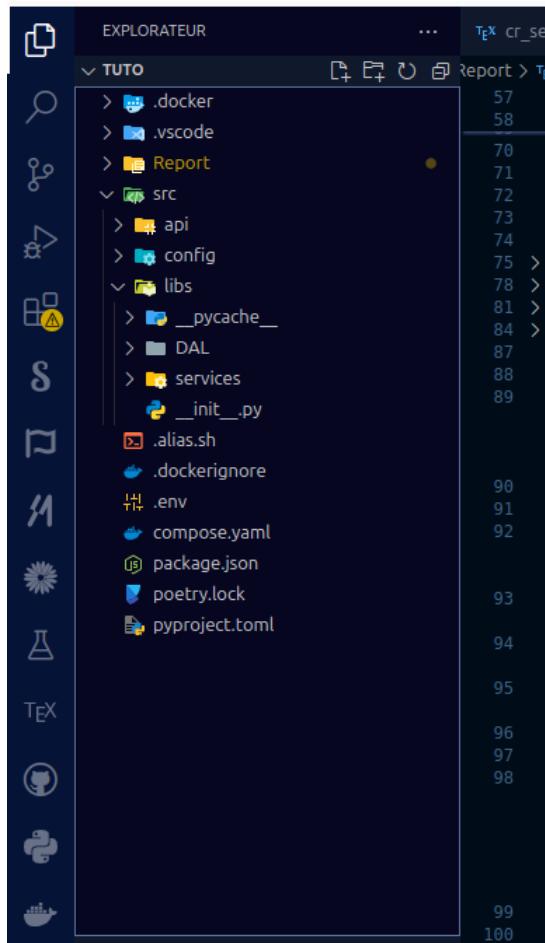


FIGURE 2.1 – Architecture du projet avec Docker et bases NoSQL

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 :

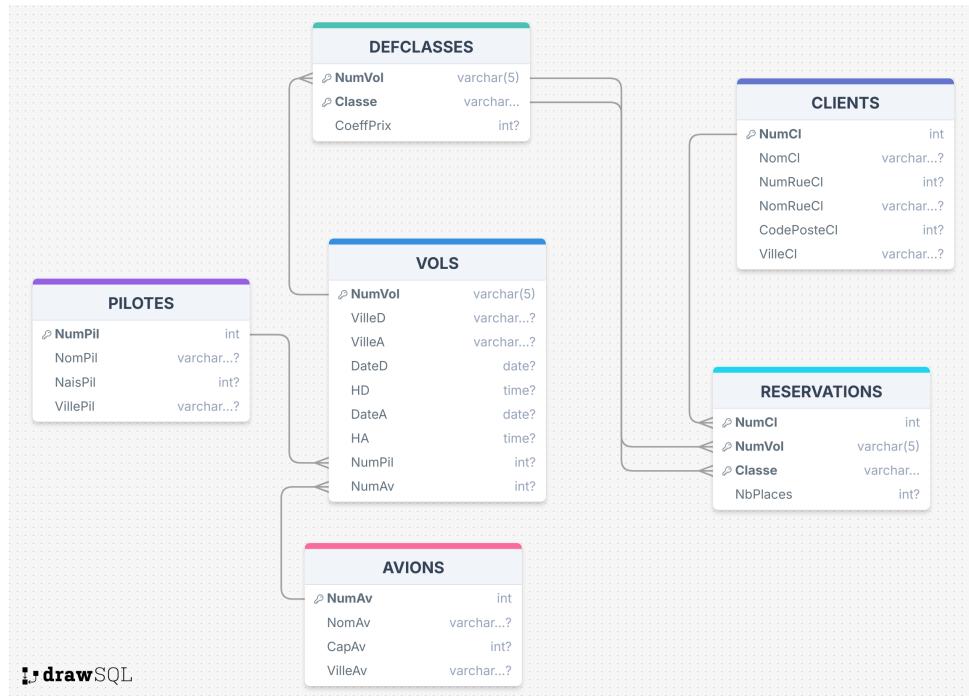


FIGURE 2.2 – Schéma de la base SQL

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.4 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.5 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.

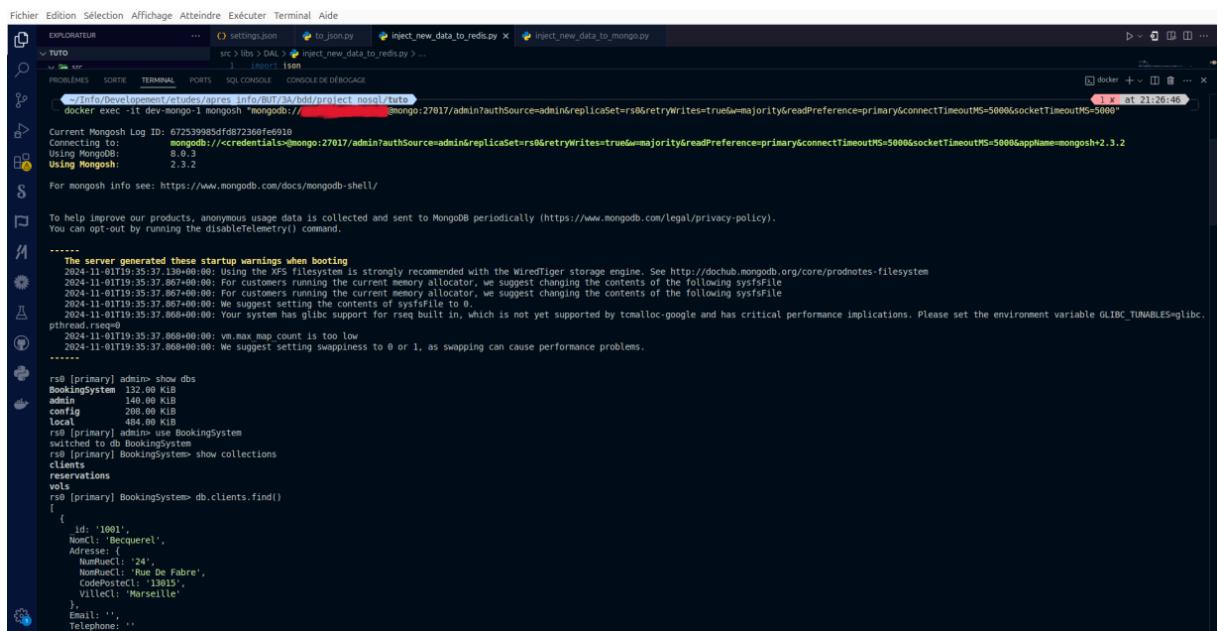
```
Fichier Édition Sélection Affichage Atteindre Exécuter Terminal Aide
FICHIER EXPLORATEUR ... settings.json to_json.py inject_new_data_to_redis.py inject_new_data_to_mongo.py
src > libs > DAL > inject_new_data_to_redis.py ...
1 import json
PROBLÈMES SORTIE TERMINAL PORTS SQL CONSOLE CONSOLE DE DÉBAGAGE
127.0.0.1:6379> keys *
ec1: 127.0.0.1:6379> keys *
1) "vol:v663"
2) "vol:v211"
3) "reservation:R025"
4) "vol:v220"
5) "vol:v209"
6) "vol:v208"
7) "vol:v889"
8) "vol:v622"
9) "vol:v625"
10) "vol:v191"
11) "reservation:R043"
12) "vol:v518"
13) "client:1080"
14) "vol:v97"
15) "vol:v690"
16) "vol:v614"
17) "client:1001"
18) "vol:v620"
19) "vol:v623"
20) "vol:v969"
21) "vol:v885"
22) "vol:v1007"
23) "client:1025"
24) "vol:v966"
25) "client:1028"
26) "vol:v624"
27) "reservation:R047"
28) "vol:v013"
29) "vol:v626"
30) "vol:v628"
31) "vol:v180"
32) "vol:v146"
33) "vol:v153"
34) "vol:v154"
35) "reservation:R004"
36) "reservation:R013"
37) "vol:v621"
38) "vol:v213"
39) "vol:v627"
40) "vol:v866"
41) "vol:v627"
42) "reservation:R017"
43) "vol:v150"
44) "vol:v157"
45) "vol:v662"
```

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.6 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 sysfsFile settings. Below the logs, the user runs commands to switch to the 'BookingSystem' database and view the contents of the 'clients' collection. One document is shown in detail:

```

rs0 [primary] admin> show dbs
BookingSystem 132.00 kB
admin 140.00 kB
empty 200.00 kB
local 484.00 kB
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", "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.

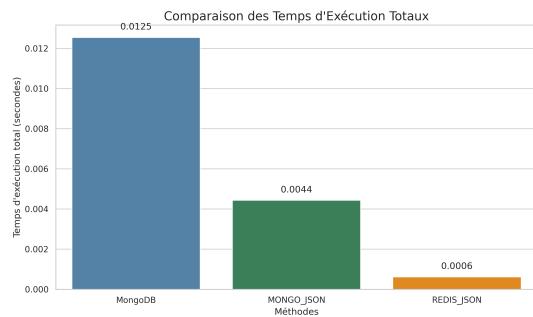
3.1 Exemple de requêtes et de jointures

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.

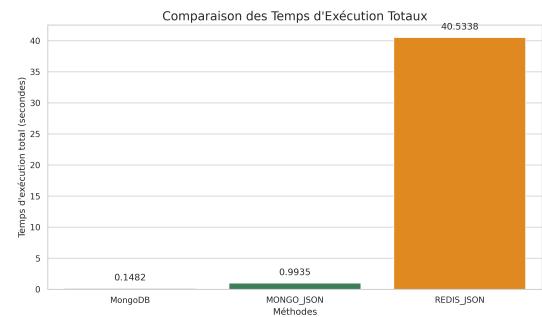
3.2 Comparaison des Performances

La figure 3.1a montre les performances des requêtes avec les données d'origine, tandis que la figure 3.1b 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 3.1 – 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.6).

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%

4.2 Limites et Améliorations Possibles

Bien que NoSQL ait permis une flexibilité accrue, certains cas d'utilisation nécessitent encore une réflexion pour l'optimisation des écritures massives.

Chapitre 5

Conclusion

La migration de SQL vers NoSQL, en utilisant Redis et MongoDB, a permis d'optimiser l'accès aux données tout en offrant une flexibilité accrue pour gérer des données semi-structurées. Les performances, analysées dans la section Discussion, montrent les avantages de chaque base de données selon les types de requêtes, ce qui confirme leur complémentarité pour des applications modernes.

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 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.3 – Fichier de configuration Docker pour Python

A.4 Crédit 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
22         # Vérifier si la première colonne est une clé unique
23         if fileName in ["DEFCLASSES.txt", "RESERVATIONS.txt"]:
24             # Stocker les données dans une liste pour ces fichiers
25             dictAllJson[fileName] = []
26             with open(file_path, 'r') as fh:
27                 for line in fh:
28                     description = list(line.strip().split("\t"))
29                     if len(description) < len(fields):
30                         # Gérer les lignes avec des colonnes manquantes
31                         continue
32                     data_entry = {}
33                     for i, categorie in enumerate(fields):
34                         data_entry[categorie] = description[i]
35                     dictAllJson[fileName].append(data_entry)
36
37         else:
38             # Utiliser un dictionnaire avec la première colonne comme clé
39             dictAllJson[fileName] = {}
40             with open(file_path, 'r') as fh:
41                 for line in fh:
42                     description = list(line.strip().split("\t"))
43                     if len(description) < len(fields):
44                         # Gérer les lignes avec des colonnes manquantes
45                         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[fileNames] [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.4 – Script de création des documents JSON

A.5 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.5 – Script d'insertion des documents dans Redis

A.5.1 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 db = client.get_database('BookingSystem')
22
23 # Créer ou obtenir les collections
24 vols_collection = db.get_collection('vols')
25 clients_collection = db.get_collection('clients')
26 reservations_collection = db.get_collection('reservations')
27
28 # Effacer les collections existantes pour éviter les doublons lors de ré-exécutions
29 vols_collection.delete_many({})
30 clients_collection.delete_many({})
31 reservations_collection.delete_many({})
32
33 # Insérer les documents JSON dans chaque collection respective
34 vols_collection.insert_many(vols)
35 clients_collection.insert_many(clients)
36 reservations_collection.insert_many(reservations)
37
38 print("Les données ont été insérées avec succès dans MongoDB.")
39
```

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

A.6 Profiling des Requêtes

```
# Détails de la performance des requêtes dans chaque base
```

A.7 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.