Rapport TP Bases de données avancées

Nom et Prénom : BOUKHRIS Mohamed Lyazid

Encadré par : CHOUIKRI Khalil

Structure du projet (Hiérarchie des fichiers)

```
bottle/
├── static/                      # Fichiers statiques (CSS, images,
etc.)
├── views/                       # Templates HTML pour l'interface
web
└── app.py                       # Script principal de l'application
web Bottle

env/                             # Environnement virtuel Python

imdb_data/
├── imdb-medium/                 # Données IMDb (jeu de données
moyen)
├── imdb-small/                  # Données IMDb (jeu de données
petit)
└── imdb-tiny/                   # Données IMDb (jeu de données
réduit, pour tests rapides)

Replica_Set/
├── dataC/                       # Données pour l'instance MongoDB
(port 28017)
├── dataD/                       # Données pour l'instance MongoDB
(port 28018)
└── dataRS27020/                 # Données pour Replica Set MongoDB
(port 27020)

Results_Query/
├── mongo_result_query.txt       # Résultats des requêtes MongoDB
└── sqlite_result_Query.txt      # Résultats des requêtes SQLite

scripts/
├── __pycache__/                 # Fichiers compilés Python
(automatiques)
├── createBD_and_Insert.py       # Création et insertion des données
dans SQLite
├── migrateToMongo.py            # Migration des données depuis
SQLite vers MongoDB
├── mongo_queries.py             # Les 5 requêtes principales en
MongoDB (avec index)
├── mongo_sync.py                # Synchronisation MongoDB → SQLite
```

```
(via Change Streams)
├── Query_and_Performance.py       # Requêtes SQL et mesure des
performances avec/sans index
├── sqlite_sync.py                 # Synchronisation SQLite → MongoDB
(via sqlite3_update_hook)
├── structure_insert_and_query.py  # Création d'un document structuré
MongoDB + comparaison
└── test.py                        # Fichier de test local

eda_notebook.ipynb                 # Notebook d'analyse exploratoire
(pandas)
Rapport.ipynb                      # Rapport complet en Markdown + code
Python
image.png                          # Capture d'écran de la page
d'accueil (interface web)
image2.png                         # Capture d'écran de la recherche
(interface web)
```

NB : Les fichiers CSV ont déjà été ajoutés dans le dossier attendu (imdb_data/). Il suffit maintenant d'exécuter le script createBD_and_Insert.py pour générer automatiquement les bases de données SQLite (dans le dossier data/), qui pourront ensuite être utilisées pour les différentes étapes de test du projet.

Partie 1 — Base de données relationnelle avec SQLite

Objectif

L'objectif de cette première phase est de mettre en place une base relationnelle SQLite locale à partir des fichiers CSV issus du jeu de données IMDb.

Environnement de travail

L'environnement de développement a été initialisé avec un environnement virtuel Python :

```
python -m venv env
source env/bin/activate
pip install pandas sqlite3
```

Définition : Un environnement virtuel Python est un espace isolé qui permet d'installer des bibliothèques spécifiques à un projet, sans affecter les autres projets ni l'installation globale de Python.

Cela permet de garantir :

- une meilleure **portabilité du projet**,
- une **gestion indépendante des dépendances**,
- une **séparation propre entre les projets**.

Étape 1 — Étude des données (analyse exploratoire)

Avant de créer les schémas des tables, une **analyse exploratoire** a été menée à l'aide de la bibliothèque `pandas`, afin de mieux comprendre la structure des fichiers CSV.

Par exemple, pour la table `writers.csv` :

```
import pandas as pd

data =
pd.read_csv(r'C:\Users\mbouk\Desktop\FouilleDonnée\Tp1\imdb_data\imdb-tiny\writers.csv')

data.head()
data.info()
data.describe()
```

Cette étape a permis d'identifier :

- Les **types de données** à utiliser dans SQLite
  (ex. : `INTEGER`, `TEXT`, `BOOLEAN`)
- Les **contraintes d'intégrité** à prévoir
  (clés primaires, valeurs nulles, relations entre tables)
- Le **nombre de lignes**, les **valeurs manquantes** et les **valeurs uniques**

Étape 2 — Création des schémas SQLite

Les schémas des tables ont été définis dans un script Python, avec une attention particulière portée aux éléments suivants :

- La définition de **clés primaires** sur chaque table pour garantir l'unicité des enregistrements.
- L'ajout de **contraintes d'intégrité** à l'aide de **clés étrangères** (`FOREIGN KEY`) afin de représenter correctement les relations entre les entités.
- L'**activation explicite des clés étrangères** dans SQLite avec la commande suivante, car elles sont désactivées par défaut :

```
cursor.execute("PRAGMA foreign_keys = ON;")
```

Étape 3 — Insertion des données

Les données ont été insérées automatiquement à partir des fichiers `.csv` grâce au module `csv.reader` de Python.

Avant chaque insertion :

- Les valeurs manquantes représentées par `\N` ou des champs vides ont été converties en `NULL` pour respecter le format SQLite.

- Un **filtrage** a été appliqué pour garantir la cohérence des **clés étrangères** (notamment sur les tables de relation comme `characters`, `directors`, `writers`, etc.).

Exemple de filtrage

Pour la table `characters`, le script vérifie que :

- Le champ `mid` existe dans la table `movies`
- Le champ `pid` existe dans la table `persons`

Si l'une de ces deux clés est absente, la ligne est ignorée avant l'insertion.

Ce filtrage est réalisé à l'aide de deux ensembles (`set`) construits à partir des identifiants valides des tables principales. Cela permet d'éviter les erreurs d'intégrité lors de l'insertion et d'assurer une base propre dès le départ.

Ce traitement a été appliqué pour les tables suivantes :

- `characters`
- `directors`
- `writers`
- `principals`
- `knownForMovies`
- `episodes`

Parce que toutes ces tables dépendent de **clés étrangères** vers les tables principales suivantes :

- `movies`
- `persons`

Enfin, les insertions sont réalisées en lot avec `executemany()` pour optimiser les performances d'insertion.

Le script gère également les erreurs d'intégrité avec un bloc `try/except`, ce qui permet d'ignorer proprement les lignes posant problème sans interrompre l'ensemble du processus.

Visualisation des données

Pour faciliter l'inspection des données et des relations entre les tables, l'outil **DB Browser for SQLite** a été utilisé.

Cet outil offre une interface graphique permettant de :

- Visualiser le contenu des tables (lignes et colonnes)
- Parcourir les relations entre les tables via les clés étrangères
- Exécuter manuellement des requêtes SQL
- Vérifier la structure des schémas créés

L'utilisation de cet outil a été particulièrement utile pour valider la cohérence des insertions, la qualité des données importées et l'efficacité des requêtes SQL développées.

Étape 4 — Requêtes SQL et performances

Un script Python a été utilisé pour exécuter plusieurs requêtes SQL sur la base de données `imdb3.db`. L'objectif était d'évaluer la **performance d'exécution** de ces requêtes avec et sans index, comme demandé dans l'énoncé.

Script utilisé

Le script suit la logique suivante pour chaque requête :

1. **Exécuter une première fois la requête sans index** et mesurer le temps avec `time.time()`.
2. **Créer les index nécessaires** à la main pour optimiser la requête.
3. **Réexécuter la même requête** et mesurer à nouveau le temps d'exécution.
4. **Comparer les performances** avant/après indexation.
5. **Mesurer la taille du fichier** `.db` avant et après l'indexation pour étudier l'impact mémoire.

Ce processus a été appliqué à **cinq requêtes SQL** représentatives, incluant des jointures, des conditions de filtre et des tris (`ORDER BY`), ce qui permet de simuler des requêtes réelles sur une base de données volumineuse.

Utilisation de `time.time()`

Pour chronométrer l'exécution d'une requête, on utilise :

```
start_time = time.time()
cursor.execute(query)
results = cursor.fetchall()
elapsed_time = time.time() - start_time
```

Comparatif des performances des requêtes SQL sur la BD medium :

Requête	Temps sans index (s)	Temps avec index (s)	Taille avant (KB)	Taille après (KB)
1	18.780	11.431	903356	903356
2	3.212	0.218	903356	903356
3	20.622	20.372	903356	903356
4	30.290	27.346	903356	903356
5	100.707	75.743	903356	903356

Observation

L'ajout d'index a permis de réduire significativement le temps d'exécution des requêtes 1, 2 et 5.
Pour les requêtes 3 et 4, la différence est plus faible car elles sont soit déjà optimisées, soit concernent des structures où l'indexation a moins d'impact.

La taille du fichier `imdb3.db` reste identique après indexation, ce qui montre que l'empreinte mémoire des index est minime dans ce cas précis.

Partie 2 — Base NoSQL avec MongoDB

3.1 Installation et mise en place

MongoDB a été installé via le fichier MSI. Un dossier mongo-db a été créé pour stocker les données. Le serveur MongoDB a ensuite été lancé avec la commande mongod --dbpath, et le client mongosh a été utilisé pour se connecter. Un test simple a été effectué en insérant un document dans une collection afin de vérifier le bon fonctionnement de l'environnement.

3.2 Migration des données (Insertion plate) :

Les données ont été migrées depuis SQLite vers MongoDB à l'aide d'un script Python. Chaque table relationnelle est devenue une collection MongoDB.

- Les fichiers CSV **n'ont pas été utilisés** à cette étape.
- Les **requêtes SQL** ont été utilisées pour extraire les données depuis SQLite, puis insérées directement dans MongoDB avec `insert_many`.

Réécriture des requêtes MongoDB

Les 5 requêtes précédemment formulées en SQL ont été réécrites avec les opérateurs Mongo suivants :

- `find()`
- `aggregate()`
- `lookup`, `unwind`, `match`, `project`, `group`

Voici un tableau récapitulatif de ces opérateurs :

Opérateur	Rôle principal	Équivalent SQL
`find()`	Rechercher	`SELECT ... WHERE ...`
`aggregate()`	Chaîne de traitement	Plusieurs requêtes
`$match`	Filtrage	`WHERE`

Opérateur	Rôle principal	Équivalent SQL
`$lookup`	Jointure	`JOIN`
`$unwind`	Aplatir les tableaux	Exploser les lignes
`$group`	Regrouper + agrégation	`GROUP BY`
`$project`	Sélectionner, renommer champs	`SELECT`, alias

Les **temps d'exécution** ont été mesurés via la fonction `time.time()`.

Comparaison des performances sur la BD medium :

Requête	SQLite (indexé)	MongoDB (avant)	MongoDB (après indexation)	Remarques
1. Films avec Jean Reno	~27.86 sec	~0.03 sec	~0.02 sec	Très rapide avec ou sans index Mongo
2. Top 3 films d'horreur des années 2000	~0.25 sec	~5.90 sec	~0.71 sec	Fort gain après indexation des champs utilisés
3. Scénaristes sans films en Espagne	~17.62 sec	~164.73 sec	~8.45 sec	Gain énorme grâce à l'index sur titles(region, mid)
4. Acteurs avec le plus de rôles dans un film	~21.65 sec	~14.42 sec	~6.21 sec	Bonne amélioration, pipeline plus fluide
5. Personnes révélées par Avatar	~72.58 sec	~30.98 sec	~5.14 sec	Index sur startYear et numVotes très bénéfique

Représentation ASCII des performances sur la BD medium :

```
Temps (en secondes)  | SQLite (■) | MongoDB avant (●) | MongoDB après (▲)

1. Jean Reno          ■■■■■■■■■■■■■■■■■■■■■■■■■■■■          ●
▲
2. Films Horreur      ■                                      ●●●●●
▲▲
3. Pas diffusés ES    ■■■■■■■■■■■■■■■■■
●●●●●●●●●●●●●●●●●●  ▲▲▲▲▲▲▲▲
4. Acteurs rôles      ■■■■■■■■■■■■■■■■                       ●●●●●●●●●●
▲▲▲▲▲▲
5. Révélés par Avatar ■■■■■■■■■■■■■■■■■■■■■■■■■■■■          ●●●●●●●●●
▲▲▲▲▲
```

Observation :

MongoDB bsans index peut être très lent sur des requêtes complexes notamment avec des agrégations ou des regroupements croisés.

L’indexation sur les bons champs améliore les performances. On passe par exemple de 164 sec à 8 sec pour la requête 3.

SQLite reste performant sur certaines requêtes simples grâce à l’optimisation SQL + index, mais devient lent sur les volumes (exple : Jean Reno).

MongoDB est plus efficace après indexation, surtout sur des requêtes analytiques ou basées sur des structures flexibles (comme les rôles ou la combinaison de conditions).

3.3 Insertion de données structurées (MongoDB)

L’objectif de cette étape est de démontrer la puissance de MongoDB pour modéliser les données de manière **structurée** dans un format JSON imbriqué, ce qui permet de regrouper toutes les informations pertinentes autour d’un film dans un **seul document**.

Objectif de la tâche

Créer une **nouvelle collection MongoDB** contenant des documents structurés de films, avec les informations suivantes :

- Métadonnées principales du film (titre, année, genres, note, etc.)
- Liste des acteurs et autres personnels
- Titres disponibles dans différentes langues
- Épisodes (s’il s’agit d’une série)

Résultat des performances observées

Type de requête	Temps d'exécution
Récupération classique (`flat_query()`)	~6.98 secondes
Récupération structurée (`structured_query()`)	~0.03 secondes

Analyse des performances

Grâce à l'utilisation d'une **collection structurée**, chaque document MongoDB regroupe toutes les informations liées à un film (acteurs, titres, épisodes, métadonnées), ce qui permet une **récupération rapide et directe** via une seule requête `find_one()` .

En comparaison, la version "plate" nécessite :

- Plusieurs requêtes MongoDB (sur `movies` , `principals` , `persons`)
- Des jointures simulées en Python entre collections
- Plus de temps de traitement

Le **gain de performance** est donc **très significatif**, surtout pour des cas d’usage type "affichage d’un film complet" comme sur un site type IMDb.

Conclusion

Ce test met en évidence la **puissance de MongoDB pour les applications orientées documents**. Une structure hiérarchique bien pensée permet :

- Une réduction drastique des temps de réponse
- Une structure de données plus naturelle pour certaines applications
- Une meilleure maintenabilité du code côté client

3.4 Résistance aux pannes (Replica Set MongoDB)

L'objectif de cette étape est de configurer un système MongoDB **tolérant aux pannes** en utilisant un **Replica Set**. Cela permet de garantir la disponibilité des données même en cas d'arrêt d'un nœud.

Étapes réalisées

1. **Création de deux répertoires** pour stocker les données de deux instances MongoDB :
   - `dataC` pour l'instance sur le port `28017`
   - `dataD` pour l'instance sur le port `28018`
2. **Lancement des deux instances avec `--replSet` activé** :

```
mongod --replSet rs0 --port 28017 --dbpath "C:\...\dataC"
mongod --replSet rs0 --port 28018 --dbpath "C:\...\dataD"
```

Connexion et configuration du Replica Set

Connexion via le shell `mongosh` sur le port `28017`

```
mongosh --port 28017
```

Initialisation du Replica Set :

rs.initiate({ _id: "rs0", members: [{ _id: 0, host: "localhost:28017" }, { _id: 1, host: "localhost:28018" }] })

Vérification de l'état du Replica Set :

rs.status()

Test de tolérance aux pannes

Après avoir arrêté l'instance MongoDB sur le port **28017**, le système détecte automatiquement l'indisponibilité du **nœud principal (PRIMARY)**.

MongoDB procède alors à une **élection automatique**, et le **SECONDARY** (situé sur `localhost:28018`) devient **PRIMARY** sans aucune intervention manuelle.

Résultat :
La **continuité du service** est assurée automatiquement, démontrant la **résilience** et la

haute disponibilité offertes par la configuration en **Replica Set**.

Ce test valide le bon fonctionnement du mécanisme de tolérance aux pannes dans MongoDB via Replica Set.

3.5 Synchronisation bidirectionnelle (SQLite et MongoDB)

Objectif

Assurer une **synchronisation en temps réel** entre la base SQLite (locale) et MongoDB (orientée documents) durant une période de transition, afin de **maintenir la cohérence des données** pour les clients n'ayant pas encore migré.

Mise en œuvre

De SQLite vers MongoDB

- Utilisation de `sqlite3_update_hook` via la **librairie C native** `sqlite3.dll`.
- Intégration en Python avec `ctypes` pour surveiller les opérations `INSERT` et `UPDATE`.
- À chaque modification dans SQLite :
  - Le `rowid` est intercepté,
  - Une requête SELECT récupère la ligne modifiée,
  - Elle est **insérée ou mise à jour dans MongoDB**.

Toutes les tables détectées dynamiquement.

De MongoDB vers SQLite

- Utilisation des **Change Streams** (nécessite un Replica Set qu'on a vu dans l'exercice précédent).
- Un thread est lancé **par collection** pour écouter les événements `insert` et `update`.
- À chaque changement :
  - Le document modifié est propagé vers SQLite,
  - Soit inséré, soit mis à jour selon qu'il existe déjà.

Fonctionne avec toutes les collections SQL : `movies`, `persons`, `genres`, etc.

Tests réalisés

Test 1 : SQLite vers MongoDB

- Ajout d'un film fictif dans SQLite :
  ```
  INSERT INTO movies VALUES ("ttTESTSYNC999", "movie", "Test Hook",
  "Test Hook Original", 0, 2035, NULL, NULL);
  ```
- Résultat dans MongoDB :

Je tape :

```
db.movies.find({ mid: "ttTESTSYNC999" }).pretty()
```

Le film est bien présent dans MongoDB : [{ _id: ObjectId('67ebad2a66466543d98d3839'), mid: 'ttTESTSYNC999', titleType: 'movie', primaryTitle: 'Test Hook', originalTitle: 'Test Hook Original', isAdult: 0, startYear: 2035, endYear: null, runtimeMinutes: null }]

Test 2 : MongoDB vers SQLite

Insertion d'un document dans MongoDB :

```
db.movies.insertOne({
  titleType: "movie",
  primaryTitle: "The Last Recipe",
  originalTitle: "The Last Recipe",
  isAdult: false,
  startYear: 2023,
  endYear: null,
  runtimeMinutes: 110
})
```

Requête dans SQLite :

sqlite> SELECT * FROM movies WHERE primaryTitle = 'The Last Recipe';
67ebb53bb7ce253291b71237|movie|The Last Recipe|The Last Recipe|0|2023||110 sqlite>

Conclusion

La mise en place d'un mécanisme de **synchronisation bidirectionnelle entre SQLite et MongoDB** a été un succès.

Les tests effectués dans les deux sens (de SQLite vers MongoDB et inversement) ont confirmé que :

- Les **modifications locales** dans la base relationnelle sont bien propagées en temps réel vers MongoDB grâce à `sqlite3_update_hook` et un traitement Python via `ctypes`.
- Les **changements dans MongoDB**, en mode Replica Set, sont capturés via `Change Streams` et correctement insérés ou mis à jour dans SQLite.

Cette solution permet de **garantir la cohérence des données pendant la transition** d'une architecture relationnelle vers un modèle orienté documents.

Elle est **modulaire**, **extensible**, et surtout **adaptée à une migration progressive sans coupure de service**.

Interface Web

Objectif

Proposer une **interface web simple** permettant de :

- Visualiser les **derniers films synchronisés** depuis SQLite,
- Rechercher un film par titre,
- Afficher automatiquement les **posters des films** grâce à l'API d'OMDb (http://www.omdbapi.com/).

Outils et technologies utilisés

- **Framework** : [Bottle],
- **Base de données** : SQLite (`imdb3.db`),
- **API externe** : OMDb API pour récupérer les images de films,
- **HTML/CSS** avec des templates `.tpl` de Bottle pour l'affichage.

Clé API OMDb

Une **clé API gratuite** a été récupérée et activée sur le site https://www.omdbapi.com/.
La clé utilisée est : `e1456522` .

Fonctionnalités implémentées

🔹 Accueil (`/`)

- Affiche les **10 derniers films** insérés dans la base `imdb3.db` .
- Pour chaque film, la route tente de récupérer son **poster** via OMDb API.
- Résultat affiché sous forme de **cartes film** avec image et titre.

🔹 Recherche (`/search`)

- Champ de recherche pour taper un titre partiel.
- Affiche les résultats avec posters correspondants.
- Message affiché si aucun résultat.

Exemple d'interface (capture d'écran)

localhost:8080

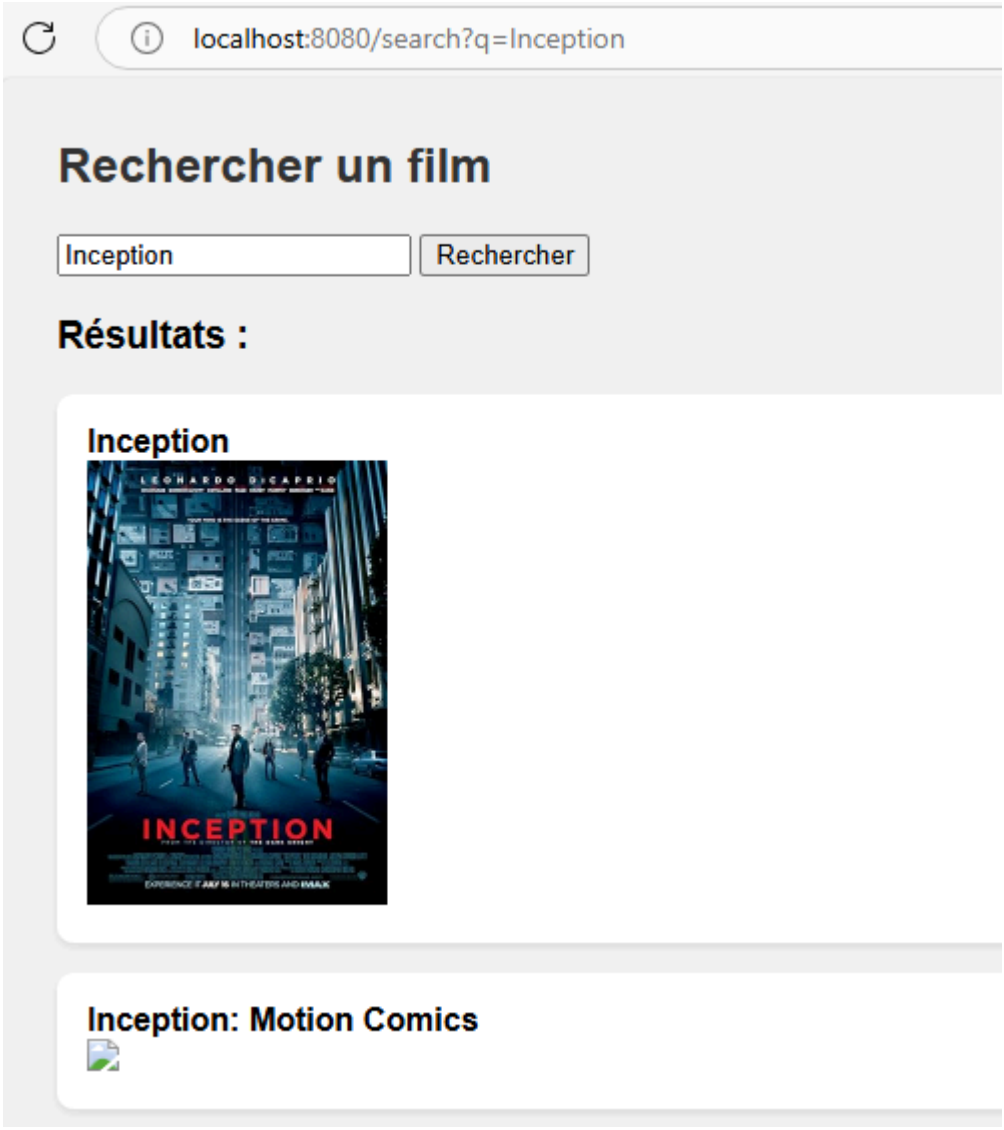
Derniers films synchronisés

The Last Recipe

Test Hook
Poster non disponible

Test Hook
Poster non disponible

Film test
Poster non disponible



Conclusion général :

Ce projet m'a permis de mieux comprendre comment fonctionnent deux types de bases de données : **relationnelles (SQLite)** et **non relationnelles (MongoDB)**.

Partie SQLite

Dans la première partie, on a utilisé **SQLite** pour :

- créer une base de données à partir de fichiers CSV,
- définir des **clés primaires et étrangères**,
- insérer des données en respectant les liens entre les tables,
- exécuter des requêtes SQL avec et sans index,
- mesurer les **performances** des requêtes.

On a vu que les **index** améliorent beaucoup la vitesse, surtout pour les requêtes complexes.

Partie MongoDB

Dans la deuxième partie, on a utilisé **MongoDB**, une base de données NoSQL. On a :

- **migré les données** depuis SQLite vers MongoDB,
- refait les mêmes requêtes avec MongoDB, en utilisant des opérateurs comme `$match`, `$group`, `$lookup`, etc.,
- mesuré les performances avant et après avoir ajouté des **index**,
- construit une nouvelle collection **structurée** (documents imbriqués),
- mis en place un **Replica Set** pour assurer la disponibilité même si un serveur tombe en panne,
- créé une **synchronisation en temps réel** entre SQLite et MongoDB,
- développé une **interface web** pour afficher les films et leurs posters avec l'API OMDb.

Ce qu'on a appris

Ce TP a montré les **forces et limites** de chaque base de données :

Point fort	SQLite	MongoDB
Structure	Très organisée, stricte	Flexible, plus naturelle pour certains cas
Requêtes	SQL classique	Pipeline d'opérations (`aggregate`)
Performances	Bonnes sur données simples	Très bonnes après indexation
Résilience	Non répartie (un seul fichier)	Réplication possible (Replica Set)
Évolution	Plus rigide	Très évolutif et adapté aux changements

Bilan

Ce TP m'a permis de :

- mieux comprendre **comment structurer et interroger une base relationnelle**,
- découvrir la **puissance de MongoDB** pour des données plus complexes,
- apprendre à **optimiser les performances** avec des index,
- **connecter et synchroniser** deux systèmes de base de données,
- **créer une interface web simple** pour interagir avec la base de données.

En résumé, ce projet montre qu'on peut utiliser **SQLite et MongoDB ensemble**, selon les besoins.
MongoDB est plus adapté si on veut des données flexibles et rapides à consulter.
SQLite est plus simple pour des structures bien définies.