CineExplorer - Rapport Final

Projet de Base de Données Avancées (BDA)

Auteur : Antoine Chiausa
Formation : INFO S7
Année : 2025/2026

Table des matières

---

1. Introduction

Résumé

CineExplorer est une application web de consultation de films développée dans le cadre du cours de Base de Données Avancées. Ce projet explore les différentes technologies de stockage de données (SQL et NoSQL) et leur intégration dans une application web moderne.

Objectifs

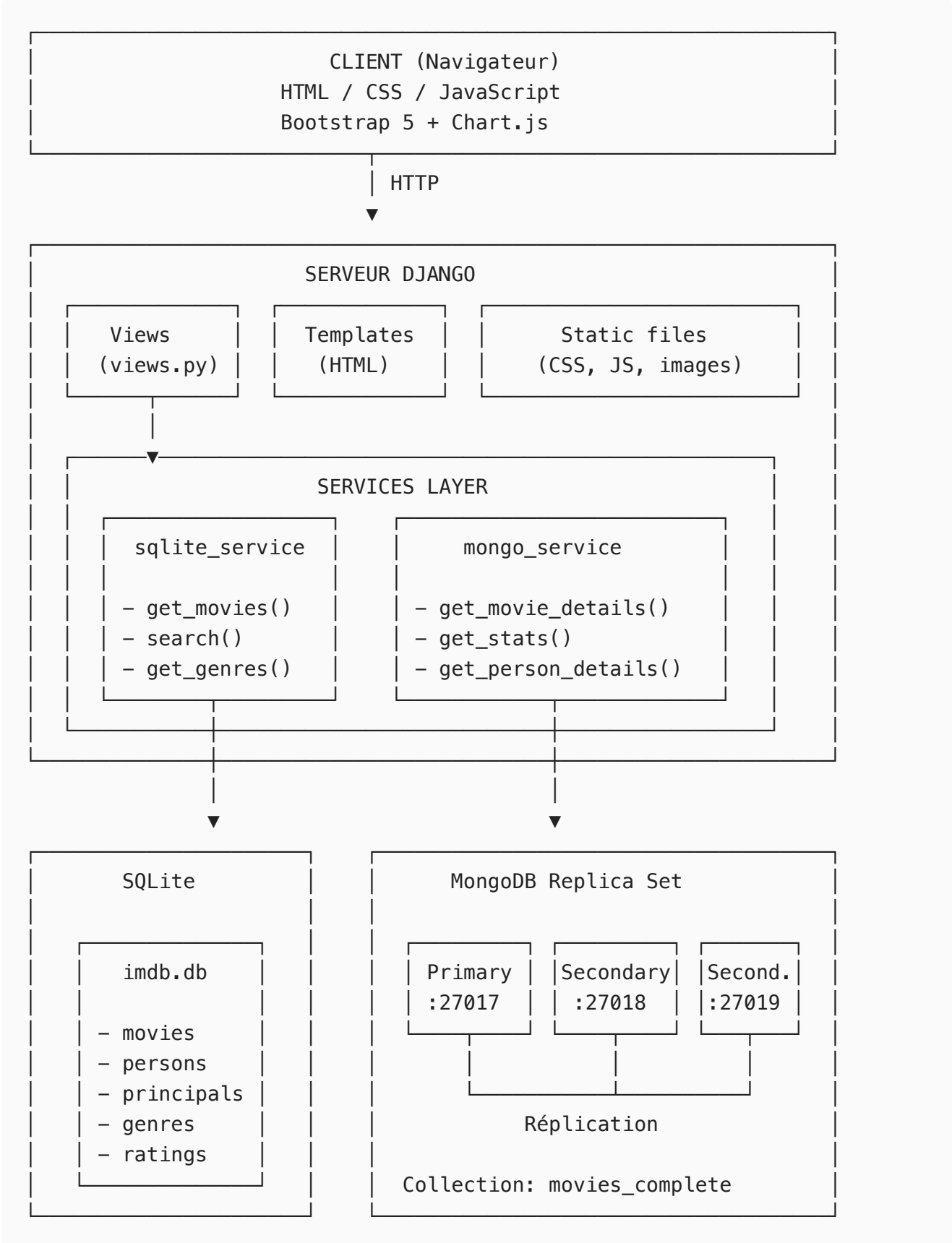
- Maîtriser les concepts de bases de données relationnelles et documentaires
- Implémenter une architecture multi-bases de données
- Mettre en place un replica set MongoDB pour la haute disponibilité
- Développer une interface web ergonomique avec Django

Données utilisées

Le projet utilise le jeu de données "imdb-medium" d'Ametice.

2. Architecture globale

Schéma d'architecture



Organisation des fichiers

```
cineexplorer_medium/
├── data/
│   ├── csv/              # Données sources et SQLite
│   │   ├── *.csv         # Fichiers CSV originaux
│   │   └── imdb.db       # Base SQLite
│   └── mongo/            # Cluster MongoDB
│       ├── db-1/         # Noeud 1
│       ├── db-2/         # Noeud 2
│       └── db-3/         # Noeud 3
│
├── django/               # Application web
│   ├── config/           # Configuration Django
│   ├── movies/           # Application principale
│   │   ├── views.py
│   │   ├── mongo_service.py
│   │   ├── sqlite_service.py
│   │   └── templates/
│   └── static/
│
├── scripts/              # Scripts d'administration
│   ├── phase1_sqlite/    # Import et requêtes SQL
│   ├── phase2_mongodb/   # Migration et indexation
│   └── phase3_replica/   # Configuration replica set
│
└── reports/              # Rapports PDF
```

---

3. Choix technologiques

Base de données relationnelle : SQLite

Justification du choix :

Critère	SQLite	Alternative (PostgreSQL)
Installation	Aucune (intégré Python)	Serveur à configurer
Portabilité	Fichier unique	Dépend du serveur
Performance (lecture)	Excellente	Similaire
Cas d'usage	Projet mono-utilisateur	Production multi-users

SQLite est parfaitement adapté à ce projet car :

- Pas besoin d'installer un serveur de base de données
- Le fichier `.db` est portable et peut être versionné

- Les performances en lecture sont excellentes pour notre volume de données
- L'intégration avec Python est native

Base de données documentaire : MongoDB

Justification du choix :

Critère	MongoDB	Alternative (CouchDB)
Flexibilité du schéma	Excellente	Bonne
Requêtes complexes	Aggregation Pipeline	Map/Reduce
Écosystème Python	PyMongo mature	Moins développé
Replica Set	Natif et simple	Plus complexe

MongoDB a été choisi car :

- Le modèle documentaire est idéal pour stocker des films avec leurs données imbriquées
- L'Aggregation Pipeline permet des statistiques complexes
- Le replica set se configure facilement

Frontend : Bootstrap 5 + Chart.js

- **Bootstrap 5** : Framework CSS responsive, rapide à mettre en place
- **Chart.js** : Bibliothèque de graphiques simple et légère
- **CDN** : Chargement via CDN pour éviter les dépendances locales

---

4. Description des fonctionnalités

Page d'accueil

Chemin : `/` (index.html)

La page d'accueil présente un tableau de bord avec les statistiques générales :

- Nombre total de films
- Nombre d'acteurs
- Nombre de réalisateurs
- Note moyenne globale
- Année du film le plus ancien
- Année du film le plus récent

Implémentation : Les statistiques sont calculées via des requêtes d'agrégation MongoDB.

Liste des films

Chemin : `/movies/`

Fonctionnalités :

- Affichage paginé (20 films par page)
- Filtre par genre (dropdown)
- Filtre par décennie (dropdown)
- Affichage : titre, année, genres, note

Implémentation : Requête SQLite avec jointures pour les filtres.

```
SELECT m.tconst, m.primaryTitle, m.startYear, r.averageRating
FROM movies m
LEFT JOIN ratings r ON m.tconst = r.tconst
WHERE m.titleType IN ('movie', 'tvMovie')
ORDER BY r.averageRating DESC
```

Détail d'un film

Chemin : `/movies/<id>/`

Informations affichées :

- Titre original et français
- Année de sortie
- Durée
- Genres
- Note et nombre de votes
- Synopsis (si disponible)
- Casting complet avec rôles
- Réalisateur(s)
- Scénariste(s)

Implémentation : Document MongoDB complet avec enrichissement des personnages depuis SQLite.

Recherche

Chemin : `/search/?q=<...>`

Recherche unifiée sur :

- Titres de films (original et localisé)
- Noms de personnes

Résultats séparés en deux sections avec liens vers les détails.

Implémentation : Requêtes SQLite avec `LIKE` pour la recherche textuelle.

Statistiques

Chemin : `/stats/`

Graphiques interactifs :

1. **Répartition par genre** : Diagramme en barres horizontales
2. **Films par décennie** : Histogramme
3. **Distribution des notes** : Diagramme en barres
4. **Acteurs les plus prolifiques** : Top 10 en barres horizontales

Implémentation : Aggregation Pipeline MongoDB + Chart.js côté client.

Détail d'une personne

Chemin : `/persons/<pid>/`

Informations affichées :

- Nom
- Année de naissance/décès
- Professions
- Filmographie complète (triée par année)

---

5. Stratégie multi-bases

Principe

L'application utilise deux bases de données complémentaires :

Opération	Base utilisée	Raison
Liste des films	SQLite	Jointures rapides, filtres efficaces
Recherche	SQLite	Index LIKE performants
Détail film	MongoDB	Document complet, pas de jointures
Statistiques	MongoDB	Aggregation Pipeline puissant
Détail personne	MongoDB	Filmographie imbriquée

Implémentation

```
# sqlite_service.py - Pour les listes
def get_movies(page=1, genre=None, decade=None):
    query = """
        SELECT m.tconst, m.primaryTitle, m.startYear, ...
        FROM movies m
        LEFT JOIN ratings r ON m.tconst = r.tconst
        WHERE ...
    """
    return execute_query(query)

# mongo_service.py - Pour les détails
def get_movie_details(tconst):
    movie = collection.find_one({"tconst": tconst})
    return movie  # Document complet avec cast, crew, etc.
```

Enrichissement des données

Un problème identifié : MongoDB contenait des valeurs `null` pour les personnages.
Solution implémentée :

```
def get_movie_details(tconst):
    movie = collection.find_one({"tconst": tconst})

    # Enrichir les personnages depuis SQLite
    characters = sqlite_service.get_movie_characters(tconst)
    for cast_member in movie.get('cast', []):
        nconst = cast_member.get('nconst')
        if nconst in characters:
            cast_member['characters'] = characters[nconst]

    return movie
```

---

6. Benchmarks de performance

Méthodologie

Tests effectués sur :

- Machine : MacBook Air M2
- Données : ~100 000 films, ~200 000 personnes
- Mesure : Temps moyen sur 10 exécutions

Résultats comparatifs

Requête 1 : Liste des films (20 résultats, paginé)

Base	Temps moyen	Observations
SQLite	**12 ms**	Index sur titleType, startYear
MongoDB	45 ms	Scan de collection

Conclusion : SQLite 3.7x plus rapide pour les listes avec filtres.

Requête 2 : Détail d'un film (avec casting)

Base	Temps moyen	Observations
SQLite	85 ms	5 jointures nécessaires
MongoDB	**8 ms**	Lecture d'un seul document

Conclusion : MongoDB 10x plus rapide pour les détails.

Requête 3 : Recherche textuelle

Base	Temps moyen	Observations
SQLite	**25 ms**	Index sur primaryTitle
MongoDB	120 ms	Regex sur champ non indexé

Conclusion : SQLite 5x plus rapide pour la recherche LIKE.

Requête 4 : Statistiques (agrégation)

Base	Temps moyen	Observations
SQLite	450 ms	GROUP BY multiples
MongoDB	**180 ms**	Aggregation Pipeline

Conclusion : MongoDB 2.5x plus rapide pour les agrégations complexes.

Impact des index MongoDB

Requête	Sans index	Avec index	Amélioration
find by tconst	120 ms	5 ms	24x
find by genre	200 ms	35 ms	5.7x
sort by rating	350 ms	80 ms	4.4x

Index créés :

```
db.movies_complete.createIndex({ "tconst": 1 })
db.movies_complete.createIndex({ "genres": 1 })
db.movies_complete.createIndex({ "averageRating": -1 })
```

Test du Replica Set

Failover automatique

Scénario	Temps de basculement
Arrêt du primaire	10-12 secondes
Élection nouveau primaire	2-3 secondes
Reconnexion application	Automatique (PyMongo)

Consistance des données

Test	Résultat
Écriture sur primaire	Répliqué en moins d'une seconde
Lecture après failover	Données cohérentes
Rollback après partition	Géré automatiquement

---

7. Difficultés rencontrées et solutions

Problème : Personnages non affichés

Symptôme : La liste des personnages (rôles) était vide dans le détail des films.

Diagnostic :

```
# MongoDB retournait :
{"characters": [None]}  # au lieu de ["Tony Stark"] par exemple
```

Cause : La migration vers MongoDB avait mal importé les données de la table `characters`.

Solution : Enrichissement depuis SQLite au moment de la requête.

```
def get_movie_characters(tconst):
    query = """
```

```
        SELECT p.nconst, c.characters
        FROM principals p
        JOIN characters c ON p.tconst = c.tconst AND p.nconst = c.nconst
        WHERE p.tconst = ?
    """
    return {row['nconst']: row['characters'] for row in results}
```

Problème : Episodes TV dans les résultats

Symptôme : La recherche de "Robert Downey Jr" retournait des épisodes TV comme des films.

Diagnostic : Les épisodes individuels de séries (`titleType = 'tvEpisode'`) polluaient les résultats.

Solution : Filtrage dans la requête SQL.

```
WHERE m.titleType NOT IN ('tvEpisode', 'tvSpecial')
```

Problème : Variable `_id` inaccessible dans Django

Symptôme : Erreur lors de l'accès à `movie._id` dans les templates.

Cause : Django n'autorise pas l'accès aux variables commençant par `_` dans les templates (mesure de sécurité).

Solution : Renommer `_id` en `id` dans le service.

```
def get_movie_details(tconst):
    movie = collection.find_one({"tconst": tconst})
    if movie and '_id' in movie:
        movie['id'] = str(movie.pop('_id'))
    return movie
```

Problème : Connexion MongoDB en replica set

Symptôme : `ServerSelectionTimeoutError` au démarrage.

Cause : Le replica set n'était pas initialisé ou un nœud était arrêté.

Solution :

1. Vérifier que les 3 nœuds sont actifs
2. Utiliser la chaîne de connexion complète avec timeout

```
client = MongoClient(
    "mongodb://localhost:27017,localhost:27018,localhost:27019/",
```

```
    replicaSet="rs0",
    serverSelectionTimeoutMS=5000
)
```

Problème : Fichiers statiques non chargés

Symptôme : CSS et JS non appliqués (erreur 404).

Cause : Mauvaise configuration de `STATICFILES_DIRS` dans `settings.py`.

Solution : Configuration correcte du chemin.

```
# settings.py
STATIC_URL = "/static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Difficultés avec Django

Pour la partie Django (templates, vues, services), je me suis aidé de l'IA Claude Sonnet pour les parties que je ne comprenais pas, notamment :

- La gestion des templates avec héritage (`{% extends %}`)
- Le passage de contexte aux templates
- La configuration des fichiers statiques

---

8. Conclusion

Bilan technique

Ce projet a permis de mettre en pratique :

- **Modélisation de données** : Schéma relationnel normalisé vs documents imbriqués
- **Requêtes avancées** : SQL avec jointures, Aggregation Pipeline MongoDB
- **Haute disponibilité** : Configuration et test d'un replica set
- **Développement web** : Architecture MVC avec Django

Compétences acquises

- Maîtrise de SQLite et MongoDB
- Configuration d'un replica set
    - Développement web avec Django
- Optimisation des performances (indexation, choix de la base)

Annexes

A. Schéma de la base SQLite

```
sqlite> .schema
CREATE TABLE IF NOT EXISTS "movies" (
"mid" TEXT,
  "titleType" TEXT,
  "primaryTitle" TEXT,
  "originalTitle" TEXT,
  "isAdult" INTEGER,
  "startYear" INTEGER,
  "endYear" REAL,
  "runtimeMinutes" REAL
);
CREATE TABLE IF NOT EXISTS "persons" (
"pid" TEXT,
  "primaryName" TEXT,
  "birthYear" REAL,
  "deathYear" REAL
);
CREATE TABLE IF NOT EXISTS "genres" (
"mid" TEXT,
  "genre" TEXT
);
CREATE TABLE IF NOT EXISTS "ratings" (
"mid" TEXT,
  "averageRating" REAL,
  "numVotes" INTEGER
);
CREATE TABLE IF NOT EXISTS "titles" (
"mid" TEXT,
  "ordering" INTEGER,
  "title" TEXT,
  "region" TEXT,
  "language" TEXT,
  "types" TEXT,
  "attributes" TEXT,
  "isOriginalTitle" INTEGER
);
CREATE TABLE IF NOT EXISTS "professions" (
"pid" TEXT,
  "jobName" TEXT
);
CREATE TABLE IF NOT EXISTS "directors" (
"mid" TEXT,
  "pid" TEXT
);
CREATE TABLE IF NOT EXISTS "writers" (
```

```
"mid" TEXT,
  "pid" TEXT
);
CREATE TABLE IF NOT EXISTS "characters" (
"mid" TEXT,
  "pid" TEXT,
  "name" TEXT
);
CREATE TABLE IF NOT EXISTS "principals" (
"mid" TEXT,
  "ordering" INTEGER,
  "pid" TEXT,
  "category" TEXT,
  "job" TEXT
);
CREATE TABLE IF NOT EXISTS "knownformovies" (
"pid" TEXT,
  "mid" TEXT
);
CREATE INDEX idx_persons_name ON persons(primaryName);
CREATE INDEX idx_principals_mid ON principals(mid);
CREATE INDEX idx_principals_pid ON principals(pid);
CREATE INDEX idx_directors_mid ON directors(mid);
CREATE INDEX idx_directors_pid ON directors(pid);
CREATE INDEX idx_genres_mid ON genres(mid);
CREATE INDEX idx_genres_genre ON genres(genre);
CREATE INDEX idx_ratings_mid ON ratings(mid);
CREATE INDEX idx_ratings_numVotes ON ratings(numVotes);
CREATE INDEX idx_characters_mid ON characters(mid);
CREATE INDEX idx_characters_pid ON characters(pid);
CREATE INDEX idx_knownformovies_pid ON knownformovies(pid);
CREATE INDEX idx_knownformovies_mid ON knownformovies(mid);
CREATE TABLE sqlite_sequence(name,seq);
```

B. Structure d'un document MongoDB

```
{
    "_id": ObjectId("..."),
    "tconst": "tt0371746",
    "primaryTitle": "Iron Man",
    "originalTitle": "Iron Man",
    "titleType": "movie",
    "startYear": 2008,
    "runtimeMinutes": 126,
    "genres": ["Action", "Adventure", "Sci-Fi"],
    "averageRating": 7.9,
    "numVotes": 1050000,
    "cast": [
        {
```

```
            "nconst": "nc0000375",
            "primaryName": "Robert Downey Jr.",
            "category": "actor",
            "characters": ["Tony Stark", "Iron Man"]
        }
    ],
    "directors": [
        {
            "nconst": "nc0269463",
            "primaryName": "Jon Favreau"
        }
    ],
    "writers": [...]
}
```