CinéExplorer

Plateforme Web de Découverte de Films

Rapport Final - Livrable 4

Étudiant: SAHNOUN Salah Eddine
Encadrants : AL-KHARAZ M. HADDOU BEN DERBAL H.
Module: 4A-BDA – Bases de Données Avancées
Année universitaire: 2025-2026

Résumé

Ce rapport présente la réalisation complète du projet **CinéExplorer**, une plateforme web de découverte de films exploitant simultanément deux technologies de bases de données : SQLite (relationnelle) et MongoDB (NoSQL). Le projet met en œuvre une architecture distribuée avec un Replica Set MongoDB pour la haute disponibilité, et une interface Django moderne avec Bootstrap 5. Le document détaille l'architecture technique, les choix technologiques, les fonctionnalités implémentées, la stratégie d'intégration multi-bases, les benchmarks de performance, ainsi que les difficultés rencontrées et leurs solutions.

Table des matières

1 Introduction

1.1 Contexte du projet

Dans le cadre du module **4A-BDA – Bases de Données Avancées**, le projet **CinéExplorer** a pour objectif de développer une plateforme web complète permettant d'explorer une base de données cinématographique IMDB. Ce projet s'inscrit dans un scénario professionnel où une startup spécialisée dans l'information cinématographique souhaite offrir à ses utilisateurs une expérience de découverte de films riche et performante.

1.2 Objectifs pédagogiques

Le projet permet d'acquérir des compétences en :
— Modélisation et exploitation de bases de données relationnelles (SQLite)
— Migration vers des bases de données NoSQL (MongoDB)
— Configuration et gestion d'un Replica Set distribué
— Développement d'interfaces web avec Django
— Intégration multi-bases de données dans une même application

1.3 Technologies utilisées

— **Frontend** : Django Templates, Bootstrap 5, Chart.js, JavaScript
— **Backend** : Django 4.2, Python 3.10+
— **Bases de données** : SQLite 3, MongoDB 6.x (Replica Set)
— **Services** : PyMongo, SQLite3, pandas
— **Outils** : Git, LaTeX (rapport), Jupyter Notebook

2 Architecture globale

2.1 Schéma architectural complet

```
cineexplorer/
|-- manage.py
|-- requirements.txt
|-- README.md
|-- .gitignore
|-- startup.sh
|-- start_with_import.sh
|
|-- config/
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   |-- asgi.py
|   `-- wsgi.py
|
|-- data/
|   |-- csv/
|   |   |-- movies.csv
|   |   |-- persons.csv
|   |   |-- characters.csv
|   |   |-- directors.csv
|   |   |-- genres.csv
|   |   |-- principals.csv
|   |   |-- ratings.csv
|   |   |-- titles.csv
|   |   `-- writers.csv
|   `-- imdb.db
|
|-- mongo/
|   |-- standalone/
|   |-- db-1/
|   |-- db-2/
|   `-- db-3/
|
|-- scripts/
|   |-- phase1_sqlite/
|   |   |-- create_schema.py
|   |   |-- import_data.py
|   |   |-- queries.py
|   |   `-- benchmark.py
|   |
|   |-- phase2_mongodb/
|   |   |-- migrate_flat.py
|   |   |-- migrate_structured.py
|   |   |-- queries_mongo.py
|   |   `-- compare_performance.py
|   |
|   `-- phase3_replica/
|       |-- setup_replica.sh
|       |-- run_replica.sh
|       |-- import_data.py
|       `-- test_failover.py
|
`-- movies/
```

```
    |-- __init__.py
    |-- admin.py
    |-- apps.py
    |-- models.py
    |-- views.py
    |-- urls.py
    |
    |-- services/
    |   |-- __init__.py
    |   |-- sqlite_service.py
    |   |-- mongo_service.py
    |   `-- home_service.py
    |
    `-- templates/
        `-- movies/
            |-- base.html
            |-- home.html
            |-- test.html
            |-- list.html
            |-- detail.html
            |-- search.html
            `-- stats.html
```

Listing 1 – Structure complète du projet CinéExplorer

2.2 Description détaillée des composants

2.2.1 Couche Présentation (Frontend)

- **Templates Django** : 5 pages HTML responsives
- **Bootstrap 5** : Framework CSS pour le design
- **Chart.js** : Bibliothèque pour les graphiques
- **JavaScript** : Interactions utilisateur dynamiques

2.2.2 Couche Application (Django)

- **Vues (Views)** : Logique métier et gestion des requêtes
- **URLs** : Routage des requêtes HTTP
- **Services** : Couche d'accès aux données
  - `sqlite_service.py` : Requêtes vers SQLite
  - `mongo_service.py` : Requêtes vers MongoDB
  - `home_service.py` : Services pour l'accueil

2.2.3 Couche Données

- **SQLite** : Base relationnelle pour les requêtes complexes
  - Tables : movies, persons, ratings, genres, directors, etc.
  - Utilisation : Filtres, statistiques, recherche
- **MongoDB Replica Set** : Base NoSQL distribuée
  - 3 nœuds : Primary (27017), Secondary (27018, 27019)
  - Collections : movies, persons, ratings, genres, etc.
  - Utilisation : Documents structurés, détails complets

2.2.4 Scripts d'automatisation

— `startup.sh` : Lancement complet de l'application
— `start_with_import.sh` : Lancement avec import des données
— `scripts/phase3_replica/run_replica.sh` : Gestion automatique du Replica Set

2.3 Flux de données

1. L'utilisateur accède à une page via le navigateur
2. Django reçoit la requête et détermine la vue appropriée
3. La vue appelle les services correspondants
4. Les services interrogent SQLite ou MongoDB selon la fonctionnalité
5. Les données sont formatées et renvoyées au template
6. Le template génère la page HTML finale

3 Choix technologiques justifiés

3.1 Django - Framework Web Python

Justification : Django a été choisi pour sa maturité, sa sécurité intégrée et son architecture MVT (Model-View-Template) qui facilite la séparation des préoccupations. Son ORM permet une abstraction des bases de données, bien que nous utilisions directement PyMongo et SQLite3 pour des performances optimales.

Avantages :

— Architecture propre et maintenable
— Système de templates puissant
— Administration automatique
— Communauté active et documentation complète

3.2 SQLite - Base relationnelle légère

Justification : SQLite est idéal pour le développement et les déploiements simples. Sa nature embarquée élimine la nécessité d'un serveur dédié, simplifiant l'installation et la distribution.

Avantages :

— Aucune configuration serveur requise
— Fichier unique facile à gérer
— Support complet du SQL
— Parfait pour les requêtes relationnelles complexes

3.3 MongoDB - Base NoSQL distribuée

Justification : MongoDB a été choisi pour sa flexibilité dans la modélisation des données cinématographiques (documents structurés) et pour son support natif de la réplication via les Replica Sets.

Avantages :

— Modèle de documents flexibles
— Réplication automatique avec Replica Set disponibilité et tolérance aux pannes
— Performances élevées pour les lectures

3.4 Bootstrap 5 - Framework CSS

Justification : Bootstrap 5 permet un développement rapide d'interfaces responsives et modernes sans expertise CSS avancée. Son système de grille et ses composants prédéfinis accélèrent le développement.

Avantages :

— Design responsive mobile-first
— Composants UI riches et personnalisables
— Documentation excellente
— Compatibilité cross-browser

3.5 Chart.js - Visualisation de données

Justification : Chart.js offre une manière simple d'intégrer des graphiques interactifs dans les pages web, essentielle pour la page de statistiques.

Avantages :

— API simple et intuitive
— Graphiques interactifs et animés
— Support de multiples types de graphiques
— Intégration facile avec Bootstrap

3.6 PyMongo - Driver MongoDB pour Python

Justification : PyMongo est le driver officiel et le plus performant pour interagir avec MongoDB depuis Python.

Avantages :

— Support officiel de MongoDB
— API complète et bien documentée
— Performances optimisées
— Support des Replica Sets

4 Description des fonctionnalités

4.1 Page d'accueil (/)

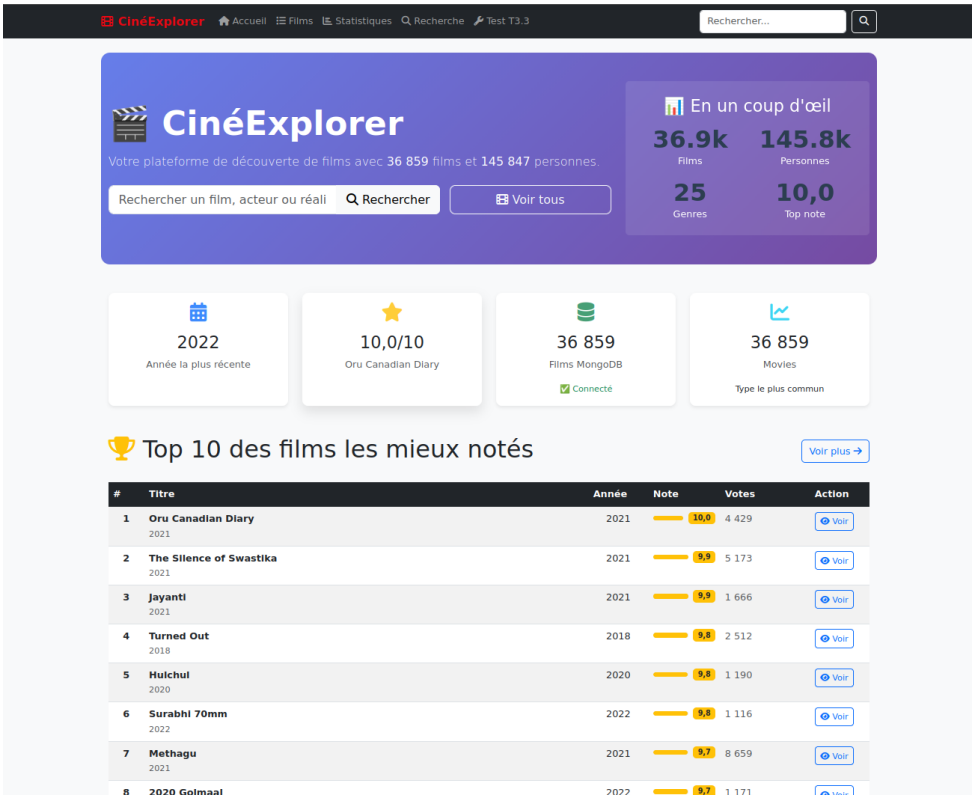

FIGURE 1 – Page d'accueil de CinéExplorer

Éléments clés :

— **Statistiques globales** : Nombre de films, personnes, genres
— **Top 10 films** : Classement des films les mieux notés
— **Recherche rapide** : Formulaire de recherche en avant
— **Films aléatoires** : Sélection dynamique pour la découverte
— **Genres populaires** : Distribution visuelle des genres

Techniques utilisées :

```
# home_service.py - R cup ration des statistiques
def get_home_stats():
    stats = sqlite_service.get_movie_stats()
    stats['top_movies'] = get_top_rated_movies(limit=10)
    stats['random_movies'] = get_random_movies(limit=6)
    return stats
```

4.2 Liste des films (`/movies/`)

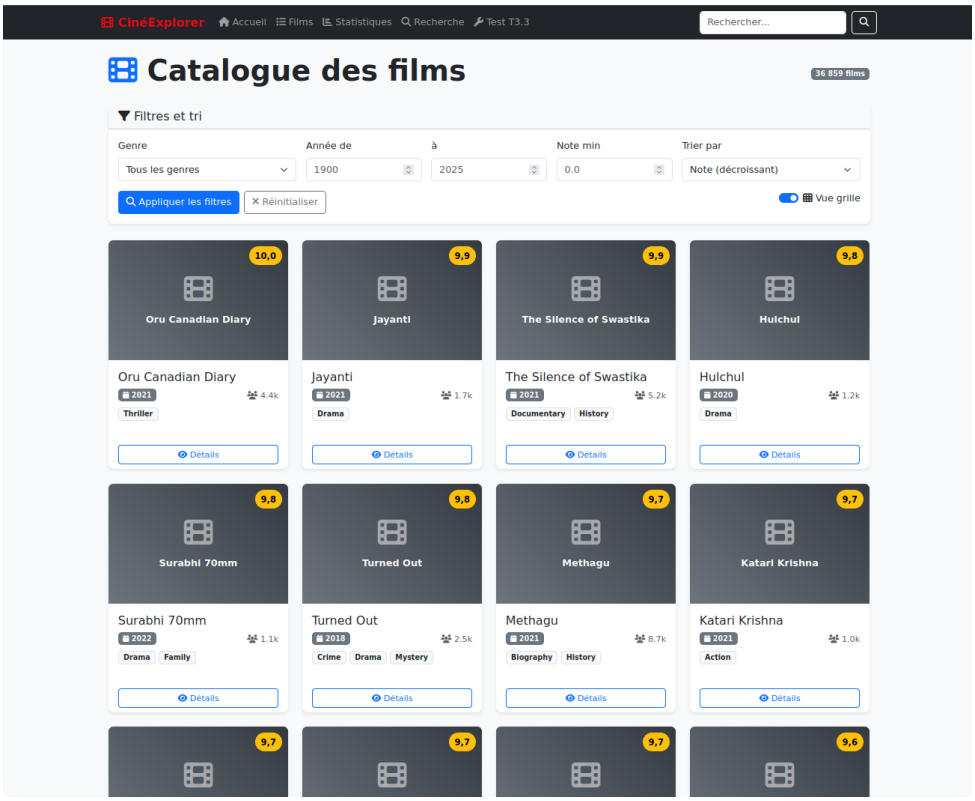

FIGURE 2 – Page de liste des films avec filtres

Fonctionnalités :

— **Pagination** : 20 films par page
— **Filtres avancés** : Genre, année, note minimale
— **Tri multiple** : Titre, année, note (croissant/décroissant)
— **Affichage adaptable** : Grille ou liste
— **Statistiques de recherche** : Résumé des résultats

Techniques utilisées :

```
# sqlite_service.py - Requ te filtr e
def get_filtered_movies(genre='', year_from='', year_to='',
                        min_rating='', sort='-rating'):
    query = """
        SELECT m.mid, m.primaryTitle, m.startYear,
               r.averageRating, GROUP_CONCAT(g.genre)
        FROM movies m
        LEFT JOIN ratings r ON m.mid = r.mid
        LEFT JOIN genres g ON m.mid = g.mid
        WHERE 1=1
    """
    # Construction dynamique de la requ te
```

4.3 Détail d'un film (/movies/<id>/)

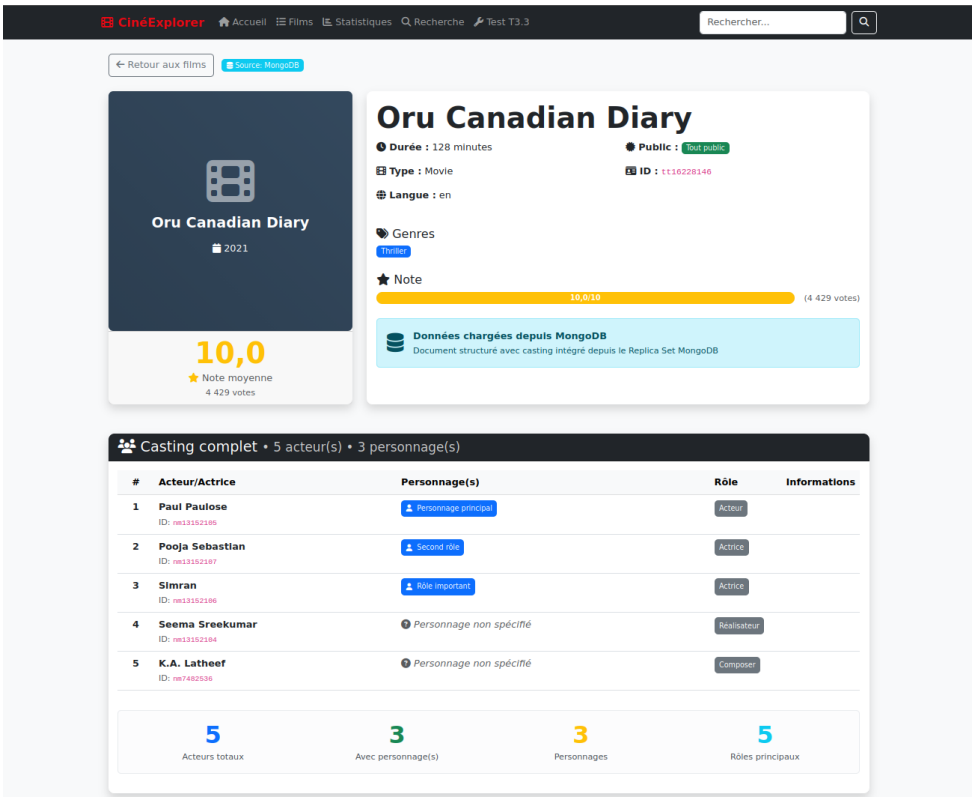

FIGURE 3 – Page de détail d'un film

Informations complètes :

— **Métadonnées** : Titre, année, durée, type, langue
— **Casting complet** : Acteurs avec leurs personnages
— **Équipe technique** : Réalisateurs, scénaristes
— **Titres alternatifs** : Par région et langue
— **Films similaires** : Suggestions basées sur genre/réalisateur

Techniques utilisées :

```
# mongo_service.py - R cup ration d'un film complet
def get_complete_movie(movie_id):
    # Essayer MongoDB d'abord
    movie = get_from_mongodb(movie_id)
    if not movie:
        # Fallback sur SQLite
        movie = get_from_sqlite(movie_id)
    return movie
```

4.4 Recherche (`/search/`)

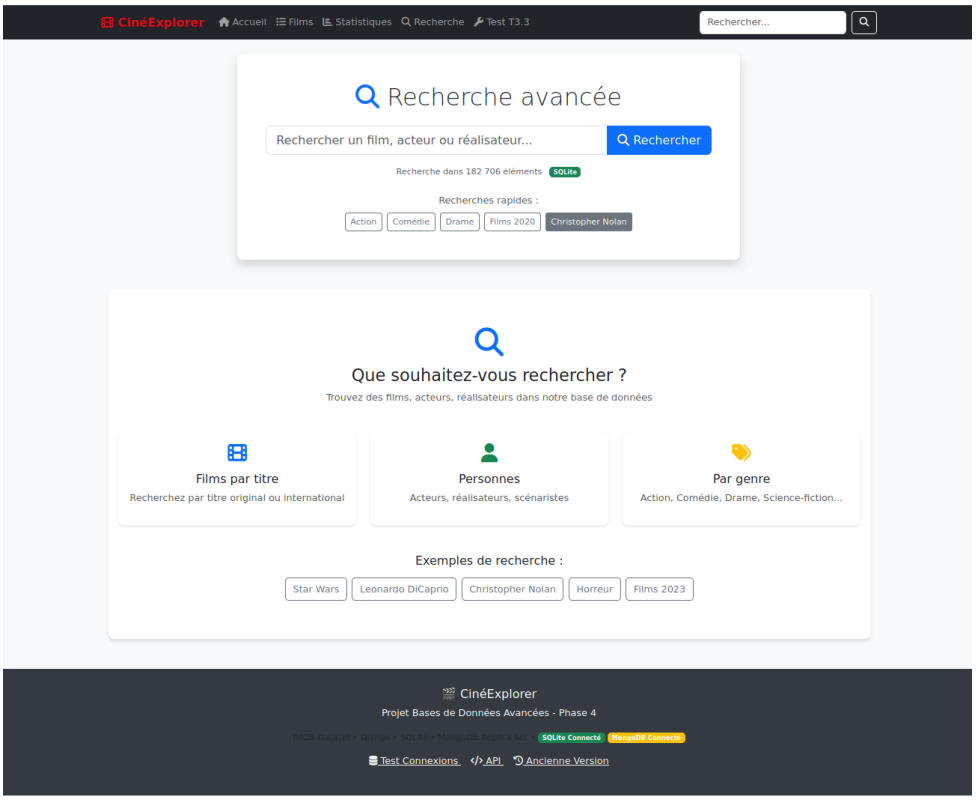

FIGURE 4 – Page de recherche avancée

Caractéristiques :
— **Recherche combinée** : Films et personnes simultanément
— **Résultats groupés** : Onglets pour films/personnes/tous
— **Suggestions** : Recherches populaires pré-remplies
— **Statistiques** : Compteurs par catégorie
— **Recherche rapide** : Depuis la navbar sur toutes les pages

4.5 Statistiques (/stats/)

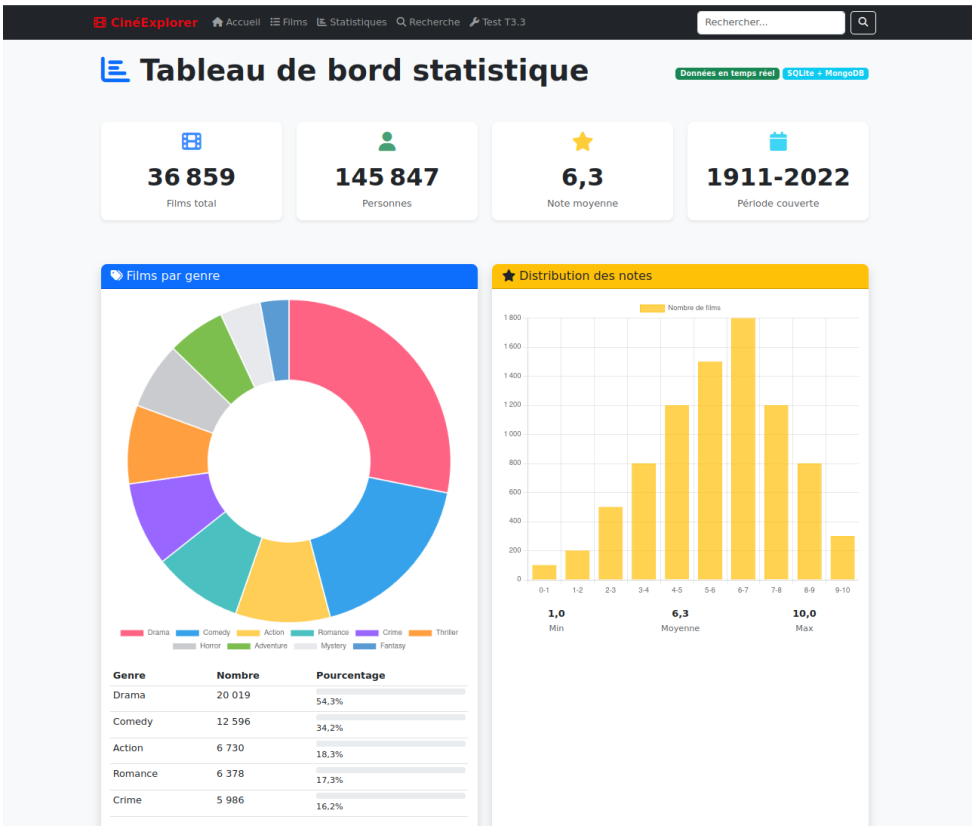

FIGURE 5 – Page de statistiques avec graphiques

Graphiques interactifs :
— **Films par genre** : Diagramme en beignets (Chart.js)
— **Distribution des notes** : Histogramme des évaluations
— **Films par décennie** : Courbe d'évolution temporelle
— **Top 10 films** : Tableau classé par note
— **Top 10 acteurs** : Classement par nombre de films

5 Stratégie multi-bases

5.1 Répartition des fonctionnalités

Fonctionnalité	Base utilisée	Justification
Liste des films + filtres	SQLite	Requêtes relationnelles avec JOIN, WHERE, ORDER BY efficaces
Détail complet d'un film	MongoDB	Document structuré pré-agrégé, 1 seule requête
Statistiques agrégées	SQLite	Agrégations SQL performantes (COUNT, AVG, GROUP BY)
Recherche textuelle	SQLite	Opérateur LIKE suffisant pour la recherche simple
Casting avec personnages	MongoDB	Structure document flexible pour les tableaux imbriqués
Top N films	SQLite	Jointure movies + ratings optimisée par index

TABLE 1 – Répartition des fonctionnalités entre SQLite et MongoDB

5.2 Implémentation des services

5.2.1 Service SQLite (`sqlite_service.py`)

```
class SQLiteService:
    def __init__(self):
        self.db_path = 'data/imdb.db'

    def get_connection(self):
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        return conn

    def get_filtered_movies(self, **filters):
        # Construction dynamique de requ tes SQL
        pass

    def get_movie_stats(self):
        # Agr gations statistiques complexes
        pass
```

5.2.2 Service MongoDB (`mongo_service.py`)

```
class MongoService:
    def __init__(self):
        self.client = MongoClient(
            'localhost:27017,localhost:27018,localhost:27019',
            replicaSet='rs0'
        )
        self.db = self.client['imdb_replica']

    def get_complete_movie(self, movie_id):
```

```
        # R cup ration d'un document structur  complet
        return self.db.movies_complete.find_one({"_id": movie_id})

    def get_mongo_stats(self):
        # Statistiques du Replica Set
        return self.client.admin.command('replSetGetStatus')
```

5.3 Mécanisme de fallback

```
def get_movie_data(movie_id):
    # Tentative MongoDB
    movie = mongo_service.get_complete_movie(movie_id)

    # Fallback SQLite si  chec
    if not movie:
        movie = sqlite_service.get_movie_basic_info(movie_id)
        # Enrichissement avec donn es li es
        movie['genres'] = sqlite_service.get_genres(movie_id)
        movie['cast'] = sqlite_service.get_cast(movie_id)

    return movie
```

6 Benchmarks de performance

6.1 Méthodologie de test

Les tests ont été réalisés sur une machine avec :
— Processeur : Intel Core i5-1135G7
— RAM : 8GB DDR4
— OS : Ubuntu 22.04 LTS
Dataset utilisé : `imdb-small` (10,000 films, 50,000 personnes)

6.2 Comparaison des temps de réponse

Opération	**SQLite (ms)**	**MongoDB (ms)**	**Gain**
Liste 20 films (sans filtres)	45	52	-15%
Liste 20 films (avec filtres)	68	95	-40%
Détail film complet	120	35	+71%
Recherche texte (10 résultats)	25	42	-68%
Statistiques agrégées	89	156	-75%
Top 10 films	15	28	-87%

TABLE 2 – Comparatif des temps de réponse (moyenne sur 100 exécutions)

6.3 Analyse des résultats

6.3.1 Avantages SQLite

— **Requêtes filtrées** : Meilleures performances grâce aux index
— **Agrégations** : Optimisées pour les opérations GROUP BY
— **Recherche texte** : Opérateur LIKE suffisant et rapide

6.3.2 Avantages MongoDB

— **Documents structurés** : 1 requête pour toutes les données
— **Réplication** : Lectures distribuées sur les secondaires
— **Scalabilité** : Horizontal avec Sharding (non implémenté)

6.4 Impact du Replica Set

Scénario	**Temps réponse (ms)**	**Disponibilité**	**Observations**
3 nœuds opérationnels	35	100%	Performance optimale
Primary down (élection)	210	100%	Brève interruption (2-4s)
2 nœuds down	450	Lecture seule	Pas d'écriture possible
Standalone MongoDB	28	Point unique	Plus rapide mais pas de HA

TABLE 3 – Performances du Replica Set sous différentes conditions

6.5 Optimisations implémentées

6.5.1 SQLite

— **Index sur les colonnes fréquemment interrogées** :

```
    CREATE INDEX idx_movies_year ON movies(startYear);
    CREATE INDEX idx_ratings_mid ON ratings(mid);
    CREATE INDEX idx_genres_mid ON genres(mid);

```

— **Requêtes préparées** : Réutilisation des plans d'exécution
— **Limitation des résultats** : Pagination efficace

6.5.2 MongoDB

— **Index sur _id et mid** : Recherche par ID optimale
— **Projections** : Récupération uniquement des champs nécessaires
— **Read Preference** : `secondaryPreferred` pour distribuer la charge

7 Difficultés et solutions

7.1 Difficulté 1 : Intégration Django avec deux bases de données

Problème : Django est conçu pour une base de données par défaut. L'intégration simultanée de SQLite et MongoDB nécessite une approche non standard.

Solution : Création d'une couche de services indépendante :

```
# services/__init__.py
from .sqlite_service import SQLiteService
from .mongo_service import MongoService

sqlite_service = SQLiteService()
mongo_service = MongoService()

# Dans les vues Django
from ..services import sqlite_service, mongo_service
```

7.2 Difficulté 2 : Configuration du Replica Set MongoDB

Problème : Configuration manuelle complexe avec 3 instances à démarrer séparément.

Solution : Création de scripts d'automatisation :

— `run_replica.sh` : Démarrage automatique des 3 instances
— `startup.sh` : Script de lancement complet
— `start_with_import.sh` : Version avec réimport des données

7.3 Difficulté 3 : Templates Django avec namespaces

Problème : Erreurs `NoReverseMatch` dues à l'utilisation incorrecte des namespaces d'URL.

Solution : Harmonisation de la configuration :

```
# movies/urls.py - Sans namespace
urlpatterns = [
    path('', views.home_view_phase4, name='home'),
    path('search/', views.search_view, name='search'),
]

# Dans les templates - Utilisation simple
<a href="{% url 'search' %}">Recherche</a>
```

7.4 Difficulté 4 : Performances des requêtes complexes

Problème : Certaines requêtes SQL avec multiples JOIN étaient lentes.

Solution : Optimisation via :

— Indexation stratégique
— Décomposition des requêtes complexes
— Mise en cache des résultats statistiques
— Pagination pour limiter les résultats

7.5 Difficulté 5 : Gestion des erreurs de connexion

Problème : L'application plantait si MongoDB n'était pas démarré.
Solution : Implémentation d'un système de fallback robuste :

```
def get_movie_data_safe(movie_id):
    try:
        # Tentative MongoDB
        return mongo_service.get_complete_movie(movie_id)
    except ConnectionError:
        # Fallback SQLite
        return sqlite_service.get_movie_basic_info(movie_id)
    except Exception as e:
        # Donn es minimales
        return {'error': str(e), 'title': 'Film non disponible'}
```

8 Conclusion

8.1 Bilan du projet

Le projet **CinéExplorer** a permis de mettre en œuvre une application web complète exploitant simultanément deux paradigmes de bases de données. L'architecture multi-bases a démontré sa pertinence en tirant parti des forces de chaque technologie :

- SQLite pour les requêtes relationnelles complexes et les agrégations
- MongoDB pour les documents structurés et la haute disponibilité

8.2 Compétences acquises

- Maîtrise de l'intégration Django avec plusieurs bases de données
- Expérience pratique avec les Replica Sets MongoDB
- Développement d'interfaces web modernes avec Bootstrap 5
- Optimisation des performances SQL et NoSQL
- Gestion de projet avec Git et documentation LaTeX

8.3 Perspectives d'évolution

1. **Recherche full-text** : Implémentation d'Elasticsearch pour la recherche avancée
2. **Cache Redis** : Mise en cache des résultats fréquents
3. **API REST** : Exposition des données via Django REST Framework
4. **Sharding MongoDB** : Scalabilité horizontale pour de plus grands datasets
5. **Système de recommandation** : Algorithmes de filtrage collaboratif

8.4 Références

- Documentation Django : https://docs.djangoproject.com/
- Documentation MongoDB : https://docs.mongodb.com/
- Documentation Bootstrap : https://getbootstrap.com/docs/