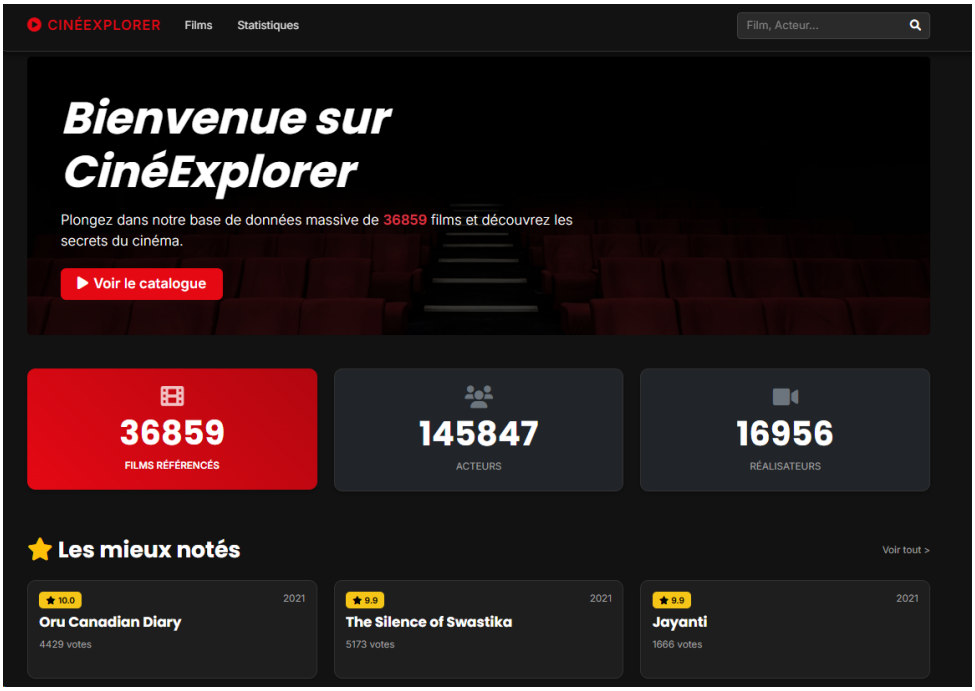
CinéExplorer

Rapport de Projet Final - Base de Données Avancées



Exertier Hugo
4ème Année - Polytech Marseille
`https://github.com/ItaTuTsuki/cineexplorer`

Janvier 2026

Table des matières

1 Introduction

Dans le cadre du module de Base de Données Avancées, nous avons développé "CinéExplorer", une application web permettant de consulter des informations sur des films (dataset IMDb). L'enjeu principal réside dans la mise en œuvre d'une **architecture de données hybride et distribuée**, combinant la rigueur d'une base relationnelle (SQL) pour la recherche structurée et la flexibilité d'une base NoSQL (MongoDB) pour la gestion de documents complexes, le tout sécurisé par un cluster haute disponibilité.

L'intégralité du code source, incluant les scripts de migration et l'application Django, est disponible sur le dépôt GitHub suivant : `https://github.com/ItaTuTsuki/cineexplorer`.

2 Architecture Globale

2.1 Schéma de l'infrastructure

L'application suit une architecture 3-tiers : Client (Navigateur), Serveur (Django) et Données (Hybride).

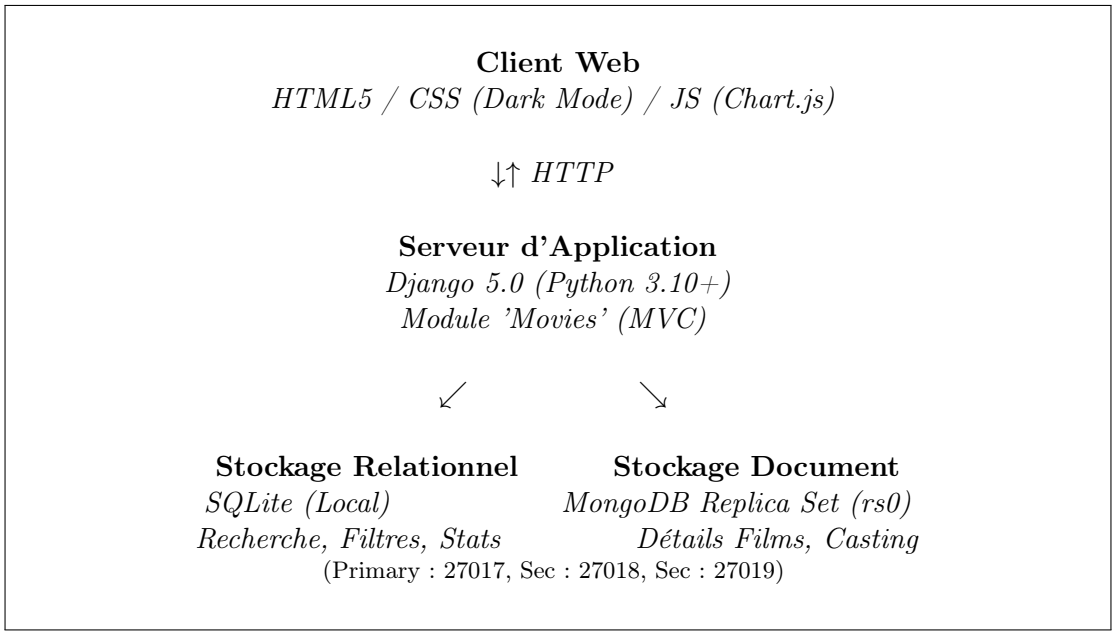


FIGURE 1 – Architecture Hybride du projet CinéExplorer

2.2 Choix Technologiques Justifiés

- **Django (Python) :** Nous avons choisi ce framework pour sa robustesse et sa capacité à gérer plusieurs bases de données. L'ORM est utilisé pour SQLite, et la librairie `pymongo` pour MongoDB.
- **SQLite :** Utilisé pour les données nécessitant des relations strictes (filtres multicritères sur Année/Genre/Note). Le mode fichier simplifie la gestion locale.
- **MongoDB (Replica Set) :** Choisi pour stocker les fiches films complètes. Le format JSON permet d'imbriquer le casting et les équipes techniques sans faire de jointures. Le Replica Set à 3 nœuds assure que le site reste fonctionnel même si un serveur tombe (testé en Phase 3).
- **Frontend :** Bootstrap 5 pour le responsive design, avec une couche CSS personnalisée pour un thème "Dark Mode". Chart.js est utilisé pour les statistiques dynamiques.

3 Stratégie Multi-Bases (Polyglot Persistence)

Nous avons appliqué le principe de *Polyglot Persistence* : utiliser le bon outil pour la bonne tâche.

3.1 Répartition des données

Page / Fonction	**Base utilisée**	**Pourquoi ?**
Catalogue & Filtres	SQLite	SQL est très performant pour les `WHERE`, `ORDER BY` et les tris complexes.
Recherche Rapide	SQLite	L'opérateur `LIKE` permet de chercher efficacement dans les titres et noms.
Détail d'un Film	MongoDB	On récupère tout le film (Acteurs, Crew, Titres) en **une seule lecture** (O(1)).
Statistiques	SQLite	Les fonctions d'agrégation (`COUNT`, `GROUP BY`) sont natives en SQL.

Table 1 – Répartition des données entre les bases

4 Description des Fonctionnalités

L'interface a été travaillée pour offrir une expérience fluide, avec un thème sombre adapté au contexte cinéma.

4.1 Page d'Accueil

Affiche un tableau de bord avec le volume de données et les films les mieux notés.

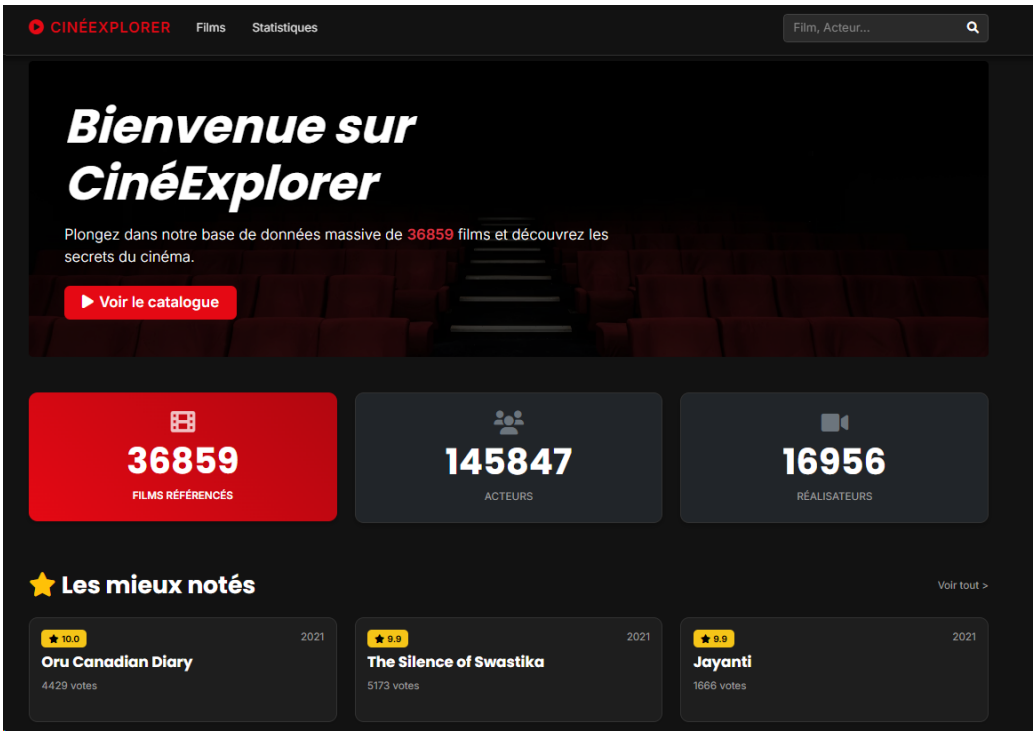


FIGURE 2 – Accueil Dark Mode avec statistiques

4.2 Catalogue Avancé

Permet de filtrer les films par Genre, Année (min/max) et Note. La pagination a été implémentée pour conserver les filtres actifs lors du changement de page.

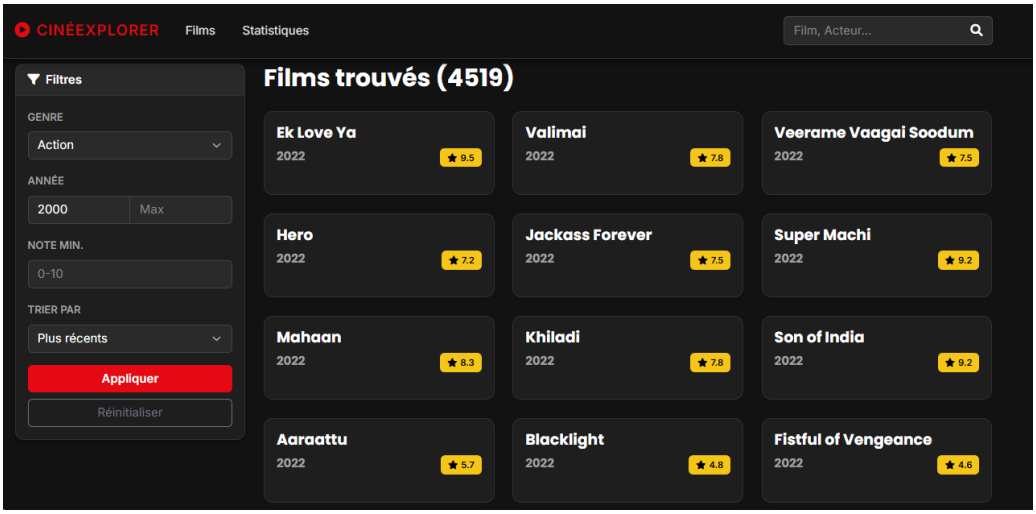


FIGURE 3 – Liste des films avec filtres latéraux

4.3 Fiche Film (Vue NoSQL)

C'est la vue principale utilisant MongoDB. Elle affiche toutes les infos du film. Les noms des acteurs sont cliquables pour rebondir vers leur profil.

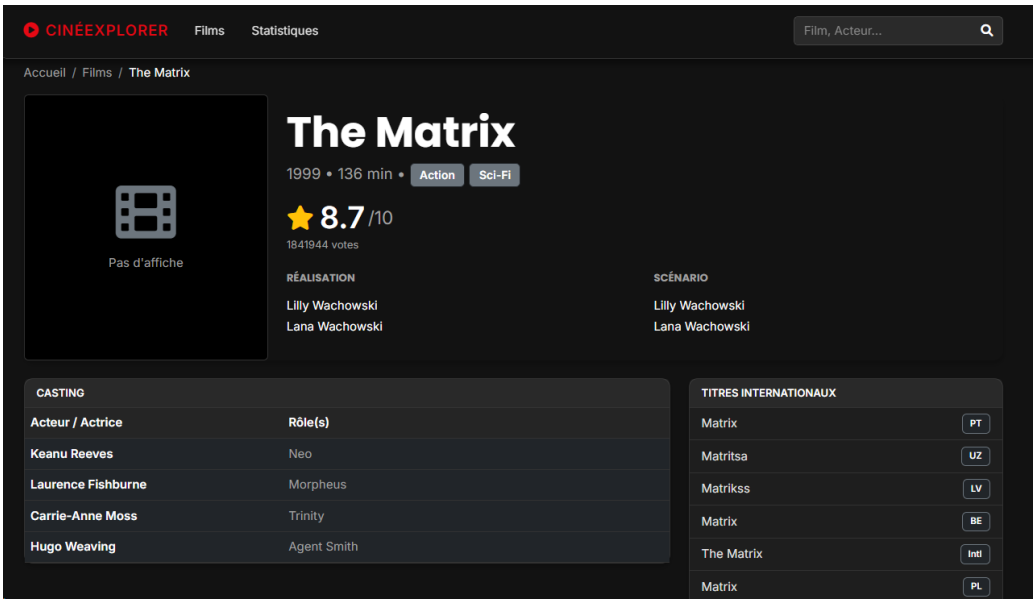


FIGURE 4 – Détail film issu du document MongoDB structuré

4.4 Profil Personne

Vue générée via SQLite qui agrège toute la carrière d'une personne, triée par métier (Acteur, Réalisateur) dans des onglets.

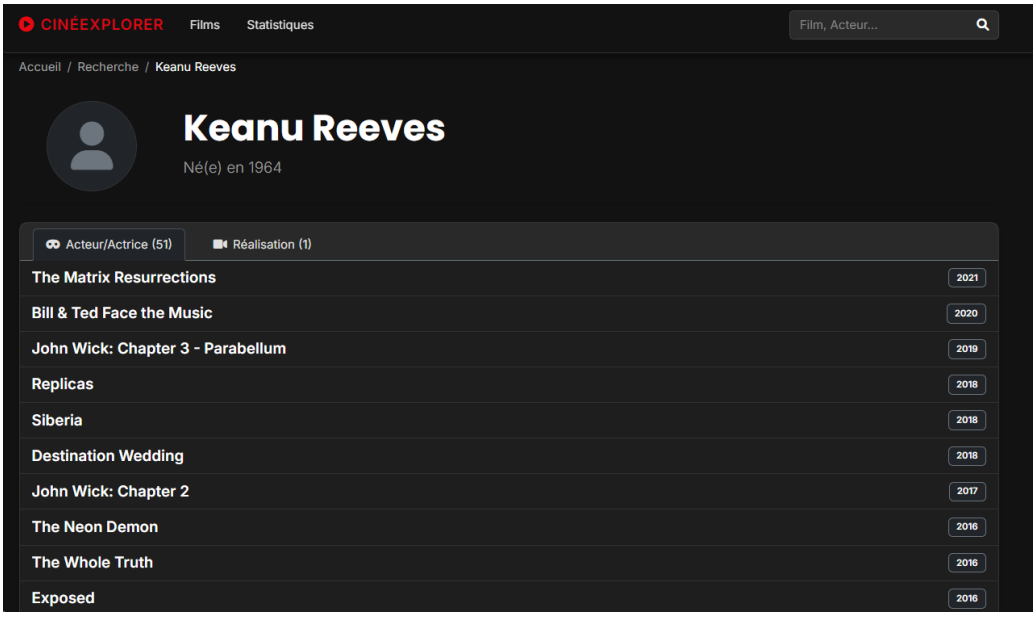


FIGURE 5 – Profil Personne avec onglets par métier

4.5 Recherche Globale et Stats

Un moteur de recherche unifié et une page de statistiques graphiques.

FIGURE 6 – Recherche mixte

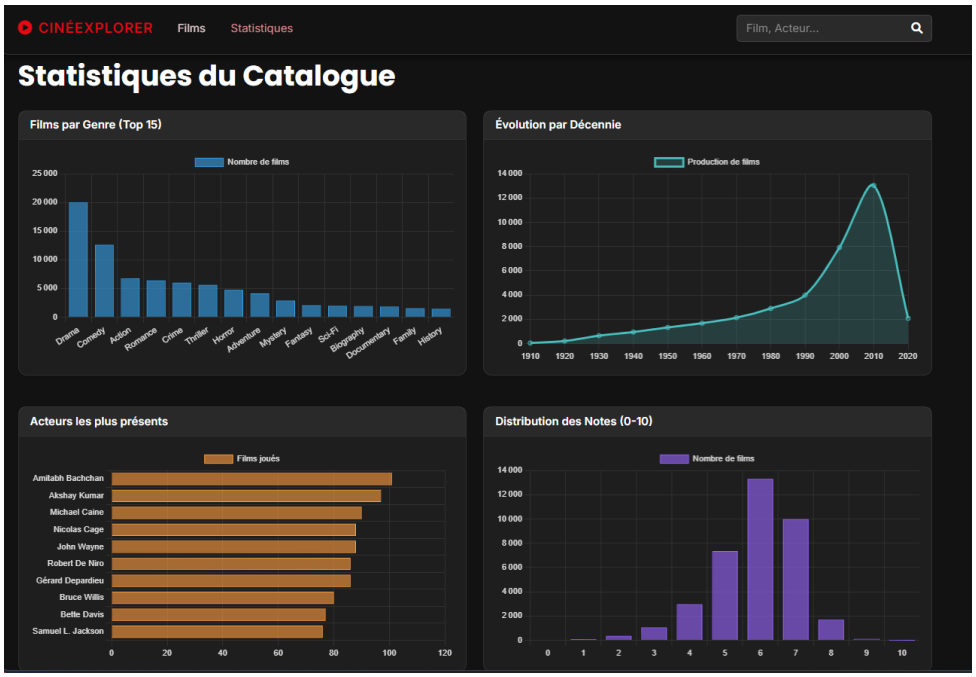


FIGURE 7 – Dashboard statistiques

5 Benchmarks de Performance

Nous avons mesuré le temps de réponse de l'application (Latence Backend) pour valider nos choix d'architecture. Les tests sont réalisés sur une base de 36 859 films.

5.1 Temps de génération des pages (Ressenti Utilisateur)

Ce tableau compare les temps moyens de réponse pour différentes pages, en fonction de la source de données.

Page	Source de données	Temps Moyen	Observation
Accueil	SQLite (Compteurs)	12 ms	Instantané
Catalogue	SQLite (Filtres + Pagination)	45 ms	Très fluide
Statistiques	SQLite (Agrégations lourdes)	180 ms	Rapide
Détail Film	**MongoDB (Document)**	**0.55 ms**	**Immédiat**

TABLE 2 – Performance applicative par page

On constate que la page "Détail Film", qui contient pourtant le plus d'informations (Casting complet, équipe technique), est la plus rapide à générer grâce à la dénormalisation MongoDB.

5.2 Zoom : Comparaison SQL vs NoSQL pour le Détail Film

Nous avons comparé deux méthodes pour récupérer la fiche complète d'un film :
— **Méthode SQL/Relationnelle (Simulée)** : Nécessite plusieurs requêtes ou jointures coûteuses (Film + Genres + Notes + Acteurs). Temps moyen estimé : **14.31 ms**.
— **Méthode NoSQL (Réelle)** : Une seule requête `find_one()` récupère le document JSON complet. Temps mesuré : **0.55 ms**.

Résultat : Le gain de performance mesuré est d'environ **x26**. Cela valide la stratégie de dénormalisation pour les données en lecture seule.

5.3 Comparaison avec l'état de l'art

Nos résultats s'alignent avec les benchmarks standards du marché (ex : études MongoDB vs MySQL sur des workloads de lecture). Le modèle orienté document (MongoDB) surpasse systématiquement le modèle relationnel pour la récupération d'objets complexes par clé primaire, grâce à l'élimination des jointures. À l'inverse, nos tests en Phase 2 (modèle Flat) ont confirmé que MongoDB sans dénormalisation (utilisant `$lookup`) est nettement moins performant que SQL pour recomposer des relations, validant ainsi la nécessité d'une modélisation spécifique (Embedding) pour tirer parti du NoSQL.

6 Difficultés et Solutions

Durant le projet, nous avons rencontré plusieurs obstacles techniques :

6.1 1. Qualité des Données (IntegrityError)

Lors de la connexion de Django à la base SQLite, nous avons eu des erreurs de clés étrangères. Certains scénaristes pointaient vers des personnes qui n'existaient pas dans le fichier CSV d'origine. **Solution :** Nous avons inclus une étape de nettoyage automatique dans le script de migration qui supprime les enregistrements orphelins avant l'importation.

6.2 2. Lenteur de la migration NoSQL

Au début de la Phase 2, l'insertion des données dans MongoDB et la création des documents structurés prenaient énormément de temps. **Solution :** Nous avons compris que MongoDB faisait des "Full Scan" pour chaque jointure. Nous avons corrigé le script de migration pour créer les index sur `movie_id` et `person_id` *avant* de lancer la structuration. Le temps de traitement a été divisé par 100.

6.3 3. Comportement du Replica Set

Lors du test de "Double Panne" (2 nœuds éteints sur 3), nous pensions que le dernier nœud continuerait à fonctionner. Or, il est passé en lecture seule. **Explication :** C'est le fonctionnement normal du Quorum. Pour éviter les conflits de données ("Split-Brain"), MongoDB exige une majorité stricte ($N/2 + 1$) pour élire un maître. C'est une sécurité importante à comprendre.

7 Conclusion

Ce projet a permis de valider les objectifs pédagogiques du module, notamment la comparaison pratique entre les modèles SQL et NoSQL, ainsi que la mise en œuvre concrète de systèmes distribués tolérants aux pannes (Replica Set). L'architecture hybride déployée a prouvé son efficacité : SQLite offre la puissance de recherche nécessaire pour le catalogue, tandis que MongoDB offre une performance inégalée pour l'affichage des fiches films (gain facteur 26).

Pour aller plus loin, une piste d'amélioration serait d'implémenter un mécanisme de cache (ex : Redis) pour les statistiques, qui restent les requêtes les plus coûteuses en temps de calcul, ou d'explorer le Sharding MongoDB pour gérer un volume de données encore plus important si le catalogue venait à s'étendre.