Compte Rendu Technique - Livrable 1

DRAVET Timothée - FISA INFO 4A

Phase 1 : Exploration et base SQLite

Résumé

Ce rapport détaille la réalisation des tâches de la Phase 1 du projet CinéExplorer, axée sur l'analyse exploratoire des données IMDB et la mise en place d'une base de données SQLite normalisée. Nous présentons la conception du schéma relationnel, l'implémentation des requêtes SQL avancées, et les résultats d'un benchmark de performance avant et après l'ajout d'index.

1 Introduction et Objectifs Atteints

La Phase 1 a couvert la mise en place de l'infrastructure de données relationnelle du projet en utilisant **SQLite**. Les objectifs principaux comprenaient la compréhension des données, la conception d'un schéma relationnel normalisé (**3NF**), l'implémentation de requêtes complexes, et l'optimisation des performances par indexation.

2 Conception et Mise en Place du Schéma Relationnel (T1.1, T1.2)

2.1 Diagramme Entité-Relation (ER)

Le schéma relationnel a été conçu en 3ème Forme Normale (3NF) pour éviter la redondance et garantir l'intégrité des données. La structure s'articule autour de deux entités principales, `Movies` (films) et `Persons` (personnes), et de nombreuses tables d'association (N-M) pour modéliser les relations complexes (casting, réalisateurs, genres, etc.).

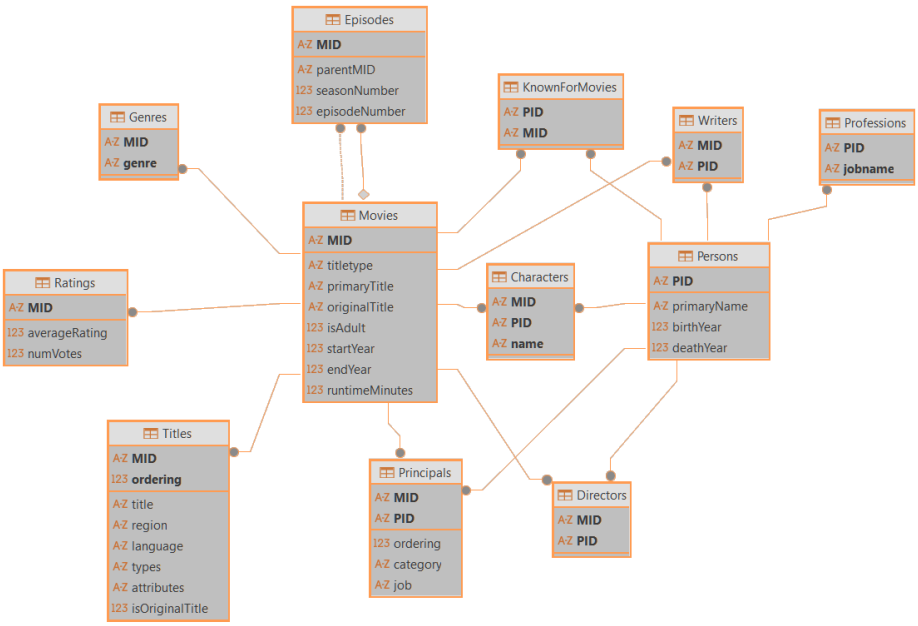

FIGURE 1 – Diagramme Entité-Relation (ER) de la base de données IMDB SQLite.

L'implémentation du schéma a été réalisée à l'aide du logiciel `DBeaver`, utilisant les contraintes de clés primaires et étrangères (`FOREIGN KEY`) avec `ON DELETE CASCADE` pour maintenir l'intégrité

référentielle. Le script `create_schema.py` a créé les tables de la base de données et `import_data.py` a géré l'insertion des données depuis les fichiers CSV, en respectant l'ordre d'insertion (*tables parentes avant enfants*) et en utilisant des transactions pour améliorer la performance.

3 Implémentation des Requêtes SQL (T1.3)

Les 9 requêtes spécifiées ont été implémentées dans le script `queries.py`. Elles utilisent des fonctionnalités SQL avancées, telles que les jointures multiples, l'agrégation (`GROUP BY` avec `HAVING`), les expressions de table communes (`WITH` / CTE), et les fonctions de fenêtrage (`RANK()`).

3.1 Exemple 1 : Évolution de Carrière (Requête 6)

Cette requête utilise une CTE (`ActorMovies`) pour isoler les films de l'acteur, puis regroupe par décennie et calcule le nombre de films et la note moyenne.

```
def query_evolution_career(conn, actor_name: str) -> list:
    # Pour un acteur donn , nombre de films par d cennie avec note moyenne (
    utiliser CTE - WITH
    sql = """
        WITH ActorMovies AS (
            SELECT m.startYear, r.averageRating
            FROM persons pe
            JOIN characters c ON pe.pid = c.pid
            JOIN movies m ON c.mid = m.mid
            JOIN ratings r ON m.mid = r.mid
            WHERE pe.primaryName LIKE ?
        )
        SELECT (startYear / 10) * 10 AS decade, COUNT(*) AS film_count, AVG(
    averageRating) AS avg_rating
        FROM ActorMovies
        GROUP BY decade
        ORDER BY decade
    """
    cursor = conn.cursor()
    cursor.execute(sql, (f'%{actor_name}%',))
    return cursor.fetchall()
```

Listing 1 – Requête 6 : Évolution de carrière (`query_evolution_career`)

3.2 Exemple 2 : Acteurs Multi-Rôles (Requête 3)

Cette requête trouve les acteurs qui ont plusieurs rôles dans un même film, triés par nombre de rôles.

```
def query_actor_multi_roles(conn, n) -> list:
    # Cette requ te trouve les acteurs qui plusieurs roles dans un meme film tri s
    par nombre de roles
    sql = """
        SELECT pe.primaryName, m.primaryTitle, COUNT(c.pid) as role_count
        FROM persons pe
        JOIN characters c ON pe.pid = c.pid
        JOIN movies m ON c.mid = m.mid
        GROUP BY pe.pid, m.mid
        HAVING role_count > 1
        ORDER BY role_count DESC, pe.primaryName
        LIMIT ?
    """
    cursor = conn.cursor()
    cursor.execute(sql, (n,))
    return cursor.fetchall()
```

Listing 2 – Requête 3 : Acteurs multi-rôles (`query_actor_multi_roles`)

4 Indexation et Benchmark de Performance (T1.4)

4.1 Stratégie d'Indexation

Les index ont été créés dans le script `benchmark.py` en analysant les besoins des requêtes SQL (filtres, tri, jointures). La stratégie s'est concentrée sur :

- **Index sur les colonnes de filtre et de tri :** `primaryName` (pour les recherches `LIKE`), `startYear` (pour les filtres temporels), et un index composite sur `Ratings(averageRating DESC, numVotes DESC)` pour les classements.
- **Index de clés étrangères :** Index simples sur les colonnes `PID` et `MID` dans les tables de relation (e.g., `Characters`, `Principals`) pour accélérer les jointures.
- **Index Composites :** Par exemple, `idx_genres_genre_mid` pour optimiser la recherche par genre et la jointure sur `MID`.

4.2 Tableau de Benchmark

Le script `queries.py` intègre la fonction `time_query` pour mesurer le temps d'exécution, permettant de calculer le gain en pourcentage.

TABLE 1 – Résultats du Benchmark des Requêtes SQL

Requête	Sans index (sec)	Avec index (sec)	Gain (%)
Filmography	18.0031	14.0658	21.87
Top_N_Films	0.3458	0.2320	32.92
Multi_Roles	20.7881	14.0349	32.49
Collaborations	33.1441	0.9505	97.13
Genre_Pop	1.5412	0.7850	49.07
Career_Evol	15.8127	0.5228	96.69
Rank_by_Genre	0.9259	0.3312	64.23
Career_Booster	42.0967	42.4677	0.88
Free_Form	7.2265	0.0005	99.99

NOTE : Temps mesurés sur un jeu de données de taille moyenne (`imdb-medium`).

4.3 Impact de l'Indexation sur la Taille de la Base

Conformément à l'attente d'une base de données indexée, l'ajout des index a entraîné une augmentation significative de la taille du fichier SQLite. Avant l'indexation, la taille de `imdb.db` était de **799 204 Ko**. Après la création des index simples et composites, cette taille est passée à **1 167 352 Ko**, soit une augmentation d'environ 46 %. Cette augmentation est nécessaire pour stocker les structures d'index qui permettent les gains de performance massifs observés, notamment sur les requêtes impliquant de multiples jointures et des recherches non exactes (`Collaborations`, `Career_Evol`).

4.4 Analyse des Gains

Les résultats ont montré des gains de performance notables, en particulier pour les requêtes impliquant des jointures complexes et des recherches par nom partiel (Requêtes 4, 8, 3). Ces gains confirment la pertinence des index créés sur les clés étrangères (`PID`, `MID`) et les index composites pour les regroupements.

5 Conclusion de la Phase 1 : Bilan SQLite et Transition

La Phase 1 est un succès, avec une base de données SQLite structurée et normalisée (3NF). La validation des 9 requêtes SQL avancées, impliquant des jointures complexes, des agrégations et des fonctions de fenêtrage, démontre la maîtrise de l'approche relationnelle. De plus, l'optimisation des

performances par une stratégie d'indexation ciblée a permis des gains significatifs, posant une base de données performante pour l'application Django.

Cependant, l'utilisation d'une structure fortement normalisée présente des inconvénients pour certaines tâches, notamment :

— **Complexité** des Requêtes de Récupération Complète : L'accès à l'information complète d'un film (acteurs, réalisateurs, notes, genres) nécessite des `JOIN` multiples, rendant les requêtes de détail d'objet verbeuses et plus complexes à maintenir.

— **Rigidité Schématique** : Le modèle relationnel impose une structure rigide, ce qui pourrait ralentir l'évolution si de nouvelles données devaient être ajoutées.

— **Coût d'Indexation** : L'obtention de performances optimales a entraîné une augmentation de la taille du fichier SQLite (passant de 799 Mo à 1167 Mo), un compromis entre la vitesse d'exécution et l'espace de stockage.

Cela justifie la transition vers la **Phase 2 : Migration MongoDB**. L'objectif de cette prochaine phase sera de tirer parti de la flexibilité du modèle orienté document (NoSQL) pour simplifier les requêtes de lecture complexes, notamment l'affichage du détail d'un film en utilisant des documents dénormalisés pré-agrégés.