Bases de Données Avancées, Rapport Phase 1 : Exploration et base SQLite.

Guilhem ALBALADEJO
Info 4A
Polytech Marseille

TABLE DES MATIÈRES

Bases de Données Avancées, Rapport Phase 1 : Exploration et base SQLite.

I. Introduction

Ce rapport correspond au compte-rendu de la première phase du cours de bases de données avancées. Nous verrons au sein de celui-ci comment à partir de données brutes créer une base de données SQL entièrement exploitable.

II. Exploration des données

Afin de se familiariser avec les données de notre application, nous commençons tout d'abord par en effectuer une exploration préliminaire. Nous en détaillerons ici les principales explorations qui en ressortent. Le notebook complet est cependant disponible en annexe dans le fichier "exploration.ipynb".

A. Statistiques descriptives

Nos données brutes sont composées de 11 fichiers csv, correspondant à des tables distinctes.

Parmi ces tables, on note deux clés présentes en tant que clés primaires et étrangères : "pid" et "mid" correspondant respectivement aux identifiants de personnes et de films.

Voici une description de nos tables :

```
-------------characters-------------
mid     varchar
pid     varchar
name    varchar
-------------directors--------------
mid    varchar
pid    varchar
-------------genres-------------
mid      varchar
genre    varchar
-------------knownformovies-------------
pid    varchar
mid    varchar
-------------movies-------------
mid                varchar
titleType          varchar
primaryTitle       varchar
originalTitle      varchar
isAdult             int64
startYear           int64
endYear           float64
runtimeMinutes    float64
-------------persons-------------
pid             varchar
primaryName     varchar
birthYear      float64
deathYear      float64
-------------principals-------------
mid         varchar
ordering     int64
pid         varchar
category    varchar
job         varchar
-------------professions-------------
pid        varchar
jobName    varchar
-------------ratings-------------
mid               varchar
averageRating    float64
numVotes           int64
-------------titles-------------
mid                varchar
ordering            int64
title              varchar
region             varchar
language           varchar
types              varchar
attributes         varchar
isOriginalTitle     int64
-------------writers-------------
mid    varchar
pid    varchar
```

Des premières statistiques, on tire les informations suivantes :

— Le champ "endYear" de movies est entièrement vide et pourrait être supprimé.
— La table movies contient également des informations sur le titre qui pourraient être contenues dans la table titles à la place.

B. Analyse exploratoire

On peut déjà, à partir de simple requêtes, révéler quelques tendances dans nos données.

Le nombre de films créé chaque année semble suivre une loi exponentielle :

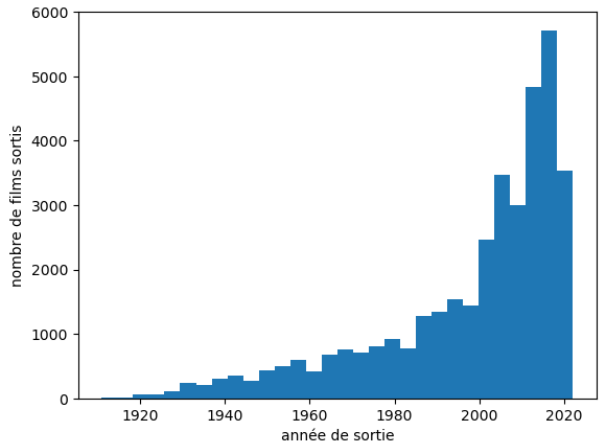

FIGURE 1. Nombre de films produit par année

On voit également une chute de la production de film pendant la période de confinement.

On peut également observer que la distribution des notes moyennes semble suivre une loi gaussienne :

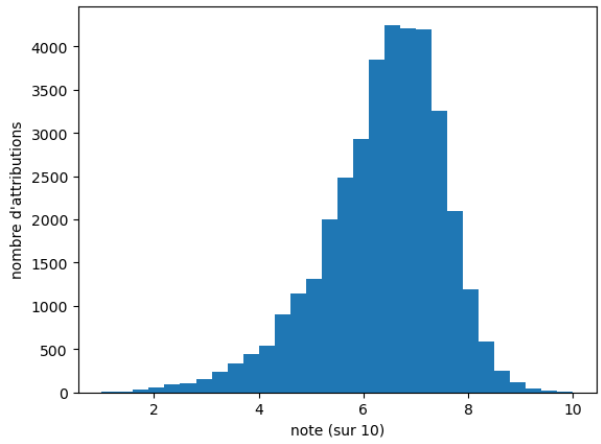

FIGURE 2. Distribution de notes attribuées aux films

Ce genre de tendances statistique est un indicateur que notre jeu de données est effectivement qualitatif et exploitable.

C. Relations entre tables

Comme dit plus tôt, notre jeu de données utilise deux clés : pid, clé principale de persons et mid, clé principale de movies.

La clé mid n'a de valeur orpheline dans aucune table, en revanche les tables characters, directors, genres et writers supposées contenir au moins une entrée pour chaque film n'ont pas d'entrée pour toutes les valeurs de mid.

La clé pid, elle, possède des valeurs orphelines : 5 valeurs existant dans characters et principals et 2 existant dans writers n'existent pas dans persons. Il existe également des valeurs manquantes, mais pas dans des tables devant contenir une entrée pour chaque personne.

III. NORMALISATION ET MODÉLISATION RELATIONNELLE

A. Normalisation

Les tables données par les fichiers csv importés sont quasiment déjà normalisées, cependant, il faut tout de même apporter quelques modifications pour que la base de données soit sous troisième forme normale.

Tout d'abord, la table principals et titles possèdent toutes les deux un champ ordering, il faut penser à l'inclure dans la clé primaire composite : en effet, une même personne peut avoir plusieurs rôles principaux dans un même film et un seul film peut avoir plusieurs titres (localisation etc...). Maintenant que l'on utilise ordering comme clé primaire dans principals, il faut également faire attention à ne pas utiliser la clé étrangère person_id dans la clé composite : en effet, l'information est maintenant trouvable seulement avec le film et l'ordre.

De plus, on note beaucoup de redondances entre les champs de la table movies et ceux de la table title dans nos données originale : on a dans movies un champ originalTitle et un champ primaryTitle, qui contiennent tous les deux des chaînes de charactères correspondant à un titre : ils devraient être stockés dans titles, d'autant plus que la table titles contient déjà un champ "isOriginal". Un autre problème important est que certains champ dans title correspondent à plusieurs associations (film, titre), d'où l'existance du champ ordering. Afin de régler au mieux ces problèmes de design dans la relation film <> titre, on utilisera le schéma suivant :

- La table movies ne contient plus d'information sur les titres.
- On crée une table titles avec une clé primaire title_id associée à un titre unique.
- on crée une table d'association (film, titre) qui contient en plus les informations indiquant si le titre est primaire et / ou original pour ce film, via des booléens.
- Finalement une dernière table title_ordering qui référence la clé primaire composite (movie_id, title_id) de la table movie_title et la combine avec un ordering pour créer sa propre clé primaire. Elle contiendra les informations qui ne peuvent être déterminées uniquement par l'association (titre, film), à savoir : région, langue, types, attributs.

Les tables characters, genres et professions nécessitent elles aussi une normalisation : une personne peut jouer plusieurs personnages dans un même film, un film peut avoir plusieurs genres et une personne peut avoir de multiples professions. Pour chacune de ces tables, on les sépare en deux tables : une contenant la relation et une contenant la liste des champs possibles avec leurs identifiants.

B. Redondance

Notre base de données est maintenant correctement normalisée, il subsiste tout de même des informations redondantes entre nos tables, pour garantir une modélisation plus fiable, il vaut mieux éviter de dupliquer l'information.

- Le champ "category" dans principals serait à présent plus correctement exprimé par la clé étrangère "profession_id".

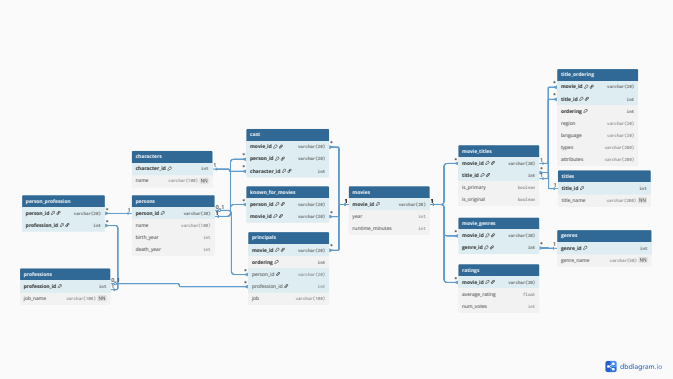

FIGURE 3. Diagramme Entité-Relation

— Les tables writers et directors encodent des relations déjà présentes dans la table principals.

Le diagramme final de notre base de données est visible dans la figure 3.

IV. CRÉATION ET MIGRATION DE DONNÉES

A. *Création du schéma*

À présent que notre base de données est entièrement modélisée sous troisième forme normale, nous pouvons la créer. Pour cela nous utiliserons un script python (`create_schema.py` en annexe), créant un fichier sqlite3 contenant chacune de nos table.

On définit les clés primaires et étrangères pour chaque table. Nous définissons également quelques contraintes d'intégrités pour empêcher l'insertion de données invalides :

— Le premier film à été créé en 1895, on s'assure donc que tous le champ year soit supérieur à 1895 dans movies.
— On s'assure que toutes les notes soient bien dans l'intervalle [0, 10].
— On s'assure qu'il n'y a pas de valeur négative pour le nombre de votes.
— On s'assure que chaque participant à un film soit né après 1800 et mort après sa naissance.
— On s'assure que les valeurs de characters, professions, genres et titles soient uniques.

B. *Insertion des valeurs*

On crée un script `insert_data.py` qui s'occupera de la migration de nos données brute dans les tables de notre base de données. Pour cela on crée une fonction par table, chargée de l'insertion des données dans la dite table. En combinant cela à des transactions, nous sommes sûr de ne jamais avoir de tables incomplètes dans notre base de données.

Nos contraintes d'intégrité définies précédemment vont s'occuper d'empêcher l'insertion de la plus des données invalides. Afin de pouvoir suivre nos statistiques d'import, on rajoute un compteur pour chaque ligne insérée avec succès et un pour chaque ligne dont l'insertion à été refusée par le SGBD (exception IntegrityError).

Notre nouveau schéma étant différent de celui de nos données CSV brutes, certaines de nos tables devront aller recouper des données entre plusieurs fichiers pour assurer la complétude de leurs valeurs. C'est le cas notamment des tables liées au titres, en particulier movie_titles qui possède une très longue fonction d'importation de données pour cette raison précise. En effet les tables titles et movies possèdent toutes les deux des données qui doivent être insérées ensemble dans cette table. On note tout de même que des données que l'on insère depuis la table movies, nos statistiques indiquent que seules 4 n'existaient pas déjà dans la table movie_titles, et seules 2 sont valides. Abandonner ces données pour un gain de performances pourrait donc être envisagé. Pour garantir, que chaque film possède bien un et un seul titre original et titre primaire, on nettoie et complète nos données après l'insertion de nos valeurs.

Afin de respecter nos contraintes d'intégrité sur les clés étrangères, nous devons respecter un certain ordre d'insertion : d'abord les tables sans clés étrangères, ensuite celles dont les clés étrangères ont été créées par la première étape etc...

V. REQUÊTES ET PERFORMANCE

A. *Présentation des requêtes*

Nous définissons les requêtes suivantes dans un fichier `queries.py` :

1) Filmographie d'un acteur:

```
SELECT t.title_name, m.year, c.name,
 ↪  r.average_rating
FROM movies m
JOIN principals p ON m.movie_id = p.movie_id
JOIN persons pe ON p.person_id = pe.person_id
JOIN movie_titles mt ON m.movie_id =
 ↪  mt.movie_id
JOIN titles t ON mt.title_id = t.title_id
LEFT JOIN cast ca ON pe.person_id =
 ↪  ca.person_id AND m.movie_id = ca.movie_id
LEFT JOIN characters c ON c.character_id =
 ↪  ca.character_id
LEFT JOIN ratings r ON m.movie_id =
 ↪  r.movie_id
WHERE pe.name LIKE ?
AND mt.is_primary = TRUE
ORDER BY m.year DESC
```

On récupère le titre, année, personnage joué et note moyenne de tous les films joué par l'acteur, en joignant les tables concernées et en filtrant uniquement les films où l'acteur apparait dans le casting. Finalement, on ordonne par l'année de sortie du film.

2) Top N films:

```
SELECT m.movie_id, t.title_name, m.year,
 ↪  r.average_rating,r.num_votes
FROM movies m
JOIN movie_titles mt ON m.movie_id =
 ↪  mt.movie_id
JOIN titles t ON mt.title_id = t.title_id
JOIN movie_genres mg ON m.movie_id =
 ↪  mg.movie_id
JOIN genres g ON mg.genre_id = g.genre_id
```

```
JOIN ratings r ON m.movie_id = r.movie_id
WHERE g.genre_name LIKE ?
AND m.year >= ?
AND m.year <= ?
AND mt.is_primary = TRUE
ORDER BY r.average_rating DESC
LIMIT ?
```

On joint les tables qui contiennent les informations que l'on veut récupérer (movie_id, titre, année, note, nombre de votants) et on filtre pour garder uniquement les films du genre qui nous intéresse sorti entre l'année de début et l'année de fin de la période considérée. Finalement on ordonne par note décroissante et on garde seulement les N premiers résultats pour avoir les top N films.

3) Acteurs multi-rôles:

```
SELECT pe.person_id, pe.name, m.movie_id,
 ↪  t.title_name,COUNT(ca.character_id)
FROM persons pe
JOIN cast ca ON pe.person_id = ca.person_id
JOIN movies m ON ca.movie_id = m.movie_id
JOIN movie_titles mt on m.movie_id =
 ↪  mt.movie_id
JOIN titles t ON mt.title_id = t.title_id
WHERE mt.is_primary = TRUE
GROUP BY ca.person_id, pe.name, m.movie_id,
 ↪  t.title_name
HAVING COUNT(ca.character_id) > 1
ORDER BY COUNT(ca.character_id) DESC
```

On joint les tables concernées, puis on regroupe les résultats par films et acteurs, pour garder uniquement les films dans lequel un même acteur est associé à plusieurs personnages, finalement on ordonne par nombre de personnages joués.

4) Collaborations:

```
SELECT pe.name, COUNT(pr.movie_id)
FROM persons pe
JOIN principals pr ON pe.person_id =
 ↪  pr.person_id
LEFT JOIN professions p ON pr.profession_id =
 ↪  p.profession_id
WHERE p.job_name LIKE 'director'
AND pr.movie_id IN (
    SELECT pr.movie_id
    FROM persons pe
    JOIN principals pr ON pe.person_id =
     ↪  pr.person_id
    LEFT JOIN professions p ON
     ↪  pr.profession_id = p.profession_id
    WHERE pe.name LIKE ?
    AND p.job_name LIKE 'actor'
)
GROUP BY pe.name
ORDER BY COUNT( DISTINCT(pr.movie_id)) DESC
```

On joint les tables concernées, puis on garde uniquement les personnes ayant travaillé comme directeurs dans des films où a aussi travaillé l'acteur considéré grâce à une sous-requête. Finalement on groupe par directeur et on ordonne par nombre de films effectués (avec l'acteur en question donc).

5) Genres populaires:

```
SELECT g.genre_name, AVG(r.average_rating),
 ↪  COUNT(mg.movie_id)
FROM genres g
JOIN movie_genres mg ON g.genre_id =
 ↪  mg.genre_id
JOIN movies m ON mg.movie_id = m.movie_id
JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY g.genre_id, g.genre_name
HAVING AVG(r.average_rating) > 7.0
AND COUNT(mg.movie_id) > 50
ORDER BY AVG(r.average_rating) DESC
```

On joint les tables concernées, puis on regroupe par genre, pour garder uniquement les genres ayant une note moyenne sur l'ensemble de leurs films supérieure à 7.0 et un nombre de films supérieurs à 50. Finalement on ordonne par note moyenne du genre.

6) Évolution de carrière:

```
WITH cte AS (
    SELECT movie_id, year - (year % 10) AS
     ↪  decade
    FROM movies
)
SELECT cte.decade,
COUNT(DISTINCT(cte.movie_id)),
AVG(r.average_rating)
FROM cte
JOIN cast ca ON cte.movie_id = ca.movie_id
JOIN persons pe ON ca.person_id =
 ↪  pe.person_id
LEFT JOIN ratings r ON cte.movie_id =
 ↪  r.movie_id
WHERE pe.name LIKE ?
GROUP BY cte.decade
ORDER BY cte.decade DESC
```

On commence par créer une expression de table commune contenant l'identifiant et la décade de chaque film de la table movies.

Puis on joint cette table aux autres contenant les informations qui nous intéressent, pour ne garder finalement que les films où à joué l'acteur qui nous intéresse, puis les grouper par décade. Finalement, on calcule sur ces groupes le nombre de films distincts et la note moyenne de tous les films.

7) Classement par genre:

```
WITH cte AS (
    SELECT g.genre_name, t.title_name, RANK()
     ↪  OVER (
        PARTITION BY g.genre_id
        ORDER BY r.average_rating DESC
    ) AS rank
    FROM genres g
    JOIN movie_genres mg ON g.genre_id =
     ↪  mg.genre_id
    JOIN movies m ON mg.movie_id = m.movie_id
    JOIN movie_titles mt ON m.movie_id =
     ↪  mt.movie_id
    JOIN titles t ON mt.title_id = t.title_id
    JOIN ratings r ON m.movie_id = r.movie_id
    WHERE mt.is_primary = TRUE
```

```
)
SELECT genre_name, title_name, rank
FROM cte
WHERE rank <= 3
ORDER BY genre_name
```

On commence par créer une expression de table commune contenant le genre, titre et rang de chaque film par genre, à l'aide d'une partition ordonnée par la note des films. Puis sur cette expression de table commune on garde uniquement les films du top 3 (pour chaque genre).

8) Carrière propulsée:

```
WITH first_hit AS (
    SELECT person_id, movie_id, year FROM (
        SELECT pe.person_id, m.movie_id,
        ↪  m.year, ROW_NUMBER() OVER(
            PARTITION BY pe.person_id
            ORDER BY m.year ASC
        ) as hit_num, r.num_votes
        FROM persons pe
        JOIN principals pr ON pe.person_id =
        ↪  pr.person_id
        JOIN movies m ON pr.movie_id =
        ↪  m.movie_id
        JOIN ratings r ON m.movie_id =
        ↪  r.movie_id
        WHERE r.num_votes > 200000
    ) WHERE hit_num = 1
), last_non_hit AS (
    SELECT person_id, movie_id, year FROM (
        SELECT pe.person_id, m.movie_id,
        ↪  m.year, ROW_NUMBER() OVER(
            PARTITION BY pe.person_id
            ORDER BY m.year DESC
        ) as nhit_num, r.num_votes
        FROM persons pe
        JOIN principals pr ON pe.person_id =
        ↪  pr.person_id
        JOIN movies m ON pr.movie_id =
        ↪  m.movie_id
        JOIN ratings r ON m.movie_id =
        ↪  r.movie_id
        WHERE r.num_votes < 200000
    ) WHERE nhit_num = 1)
SELECT pe.name, t.title_name, fh.year
FROM first_hit fh
LEFT JOIN last_non_hit lnh ON fh.person_id =
↪  lnh.person_id
JOIN persons pe ON fh.person_id =
↪  pe.person_id
JOIN movie_titles mt ON fh.movie_id =
↪  mt.movie_id
JOIN titles t ON mt.title_id = t.title_id
WHERE (lnh.year IS NULL OR fh.year >
↪  lnh.year)
AND mt.is_primary = TRUE
```

On commence par créer deux expressions de table commune : une qui contient le premier "hit" de la personne (plus de 200000 notes) et une contenant le dernier "flop" (moins de 200000 notes). On les crées à l'aide d'une partition par personne et d'un classement par année, sur une table où l'on filtre pour garder uniquement les "hits" ou "flops". Puis on garde uniquement les premiers.

Ensuite, à l'aide de ces deux CTE, on filtre les personnes n'ayant aucun "flop" après leur premier "hit" pour remplir la condition. On renvoie finalement ces personnes, le titre de leur premier "hit" et l'année de ce film.

9) Enfants star:

```
WITH cte AS (
    SELECT pe.name, t.title_name,
    ↪  r.num_votes, (m.year -pe.birth_year)
    ↪  as age
    FROM cast ca
    JOIN movies m ON ca.movie_id = m.movie_id
    JOIN movie_titles mt ON m.movie_id =
    ↪  mt.movie_id
    JOIN titles t ON mt.title_id = t.title_id
    JOIN ratings r ON m.movie_id = r.movie_id
    JOIN persons pe ON ca.person_id =
    ↪  pe.person_id
    WHERE r.num_votes > 200000
    AND mt.is_primary = TRUE
)
SELECT name, title_name, num_votes, age
FROM cte
WHERE age < 18
ORDER BY name, age ASC
```

On utilise une CTE pour calculer l'âge de chaque acteurs, quand ils ont tournés chaque film "populaire" (plus de 200000 notes). Puis sur cette CTE, on filtre par âge pour ne garder que ceux qui avaient moins de 18 ans quand ils ont tourné dans les dits films.

B. Temps d'exécution et optimisation

Certaines des requêtes présentées précédemment sont complexes et nécessitent des recherches et groupements sur de très grande tables. Afin d'améliorer les performances de nos requêtes on va donc créer des index. Afin de déterminer quels index créer, on utilise la fonction `EXPLAIN QUERY PLAN` de SQLite.

On détermine les index suivants :

- movie_titles(is_primary) utilisé dans beaucoup de WHERE à travers nos requêtes.
- principals(movie_id) pour éviter un scan complet de principals sur certains JOIN.
- principals(person_id, profession_id, movie_id) pour optimiser les JOIN sur principals.
- movies(year) pour optimiser les ordinations par année, présentes dans plusieurs de nos requêtes. La requête 2 utilise aussi deux filtres par année et bénéficiera de cet index.
- persons(name) pour optimiser certains filtrages.
- cast(person_id, movie_id) pour optimiser les JOIN sur cast.

Requête	Sans index (ms)	Avec index (ms)	Gain (%)
Q1 - Filmographie	1904	1850	2
Q2 - Top N films	83	72	15
Q3 - Multi-rôles	952	931	2
Q4 - Collaborations	1613	204	690
Q5 - Genres populaires	140	111	26
Q6 - Évolution de carrière	614	108	468
Q7 - Classement par genre	273	260	5
Q8 - Carrières propulsées	3492	1765	98
Q9 - Enfants star	90	78	15

TABLE I. TABLEAU DE BENCHMARK DES REQUÊTES

On remarque que si toutes les requêtes bénéficient d'un gain de performance grâce aux index, pour beaucoup d'entre elles ce gain est négligeable. Cela s'explique de par le schéma de notre base de données déterminé dans la partie III, qui visait à privilégier des tables avec moins de colonnes mais moins de redondance inter tables. En conséquences, nos requêtes profitent très bien des index automatiques générés par SQLite et de l'indexation sur les clés primaires.

Sur d'autres on observe des gains de performances modérés mais non-négligeables (entre 15-30 %) ce sont les requêtes qui n'ont plus besoin de scanner des tables entières grâce aux index que l'on a créés, permettant ainsi de réduire les calculs nécessaires.

Il existe aussi des requêtes utilisant des tables d'association plus complexes qui voient des gains massifs de performances, justifiant entièrement à elles seules l'utilisation d'index personnalisés.

Avant indexation, la taille de notre fichier imdb.db est de : 181 564 Ko. Après la création d'index, sa taille a augmenté à 247 076 Ko. Cela représente une augmentation conséquente de la taille de la base donnée liée aux index (36 %), et ce même si l'on a créé assez peu d'index de part l'architecture de notre base de données. Pour une base plus grande, cela pourrait remettre en cause l'utilisation d'index apportant de faibles gains de performances.

VI. CONCLUSION

Nous avons pu, au cours de cette première phase comprendre nos données brutes et les adapter pour les rendre exploitables dans le contexte d'une application. Pour cela, nous avons vu comment les normaliser, et les insérer dans une base de données. Puis nous avons vu comment exploiter notre base SQL, et comment optimiser les performances de nos requêtes, ce qui est très important pour la scalabilité de l'application qui utilisera nos données.