RAPPORT BASES DE DONNÉES AVANCÉES : SQL OR NOT SQL

Melvin CURINIER et Anass EZZINE

Sommaire

Sommaire *1*

1. Introduction *2*

2. SQLite *3*

a. **Insertion des données** 3

b. **Requêtes SQL sur la base** 4

c. **Performances** 6

3. MongoDB *8*

a. **Insertion de données plates** 8

b. **Insertion de données structurées** 9

c. **Résistance aux pannes** 9

d. **Synchronisation bidirectionnelle** 10

1. Introduction

Ce document a pour objectif d'expliquer le cheminement de la transition d'une base relationnelle et d'une base NoSQL et de comparer leurs performances.

Une entreprise a une base de données de films, d'acteurs, etc..., conservée depuis longtemps dans une base relationnelle poussiéreuse. On doit mettre en place une base NoSQL flambant neuve et gérer les petits désagréments liés à la transition entre les deux bases. Voici ci-dessous un schéma de la base relationnelle de l'entreprise :

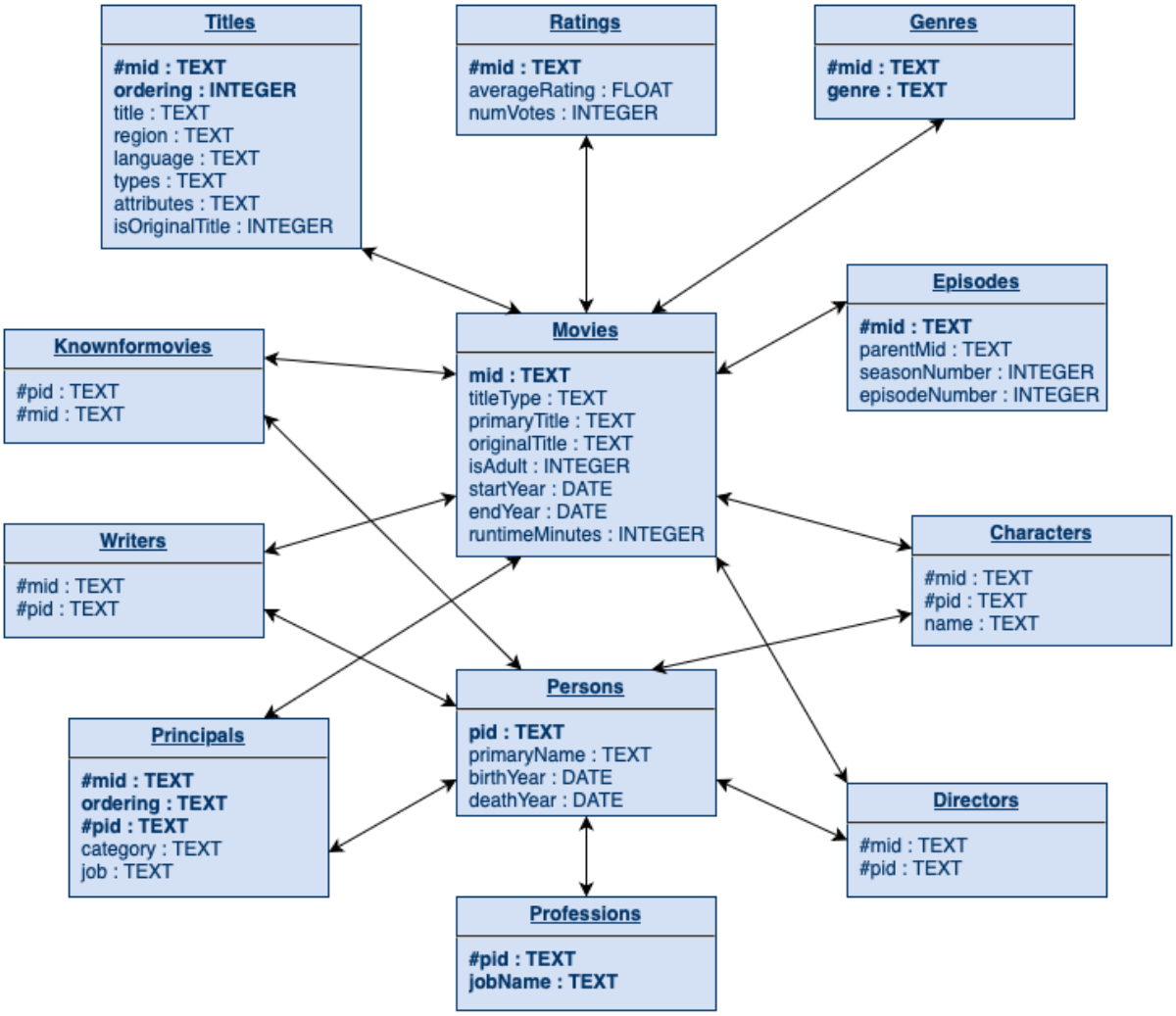

Figure 1 : Schéma de la base relationnelle

2. SQLite

a. Insertion des données

1. Création

On établit une connexion à la base de données SQLite « imdb.db » et crée un curseur associé à cette connexion, permettant ainsi d'exécuter des requêtes SQL et d'interagir avec la base de données à partir de votre programme Python.

```
con = sqlite3.connect("imdb.db")
cur = con.cursor()
```

Figure 2 : Se connecter à la base de données MySQL

On réinitialise la structure de la base de données en supprimant toutes les tables existantes grâce aux commandes ci-dessous, pour changer de data sets si on le souhaite.

```
cur.execute("DROP TABLE IF EXISTS movies")
cur.execute("DROP TABLE IF EXISTS episodes")
```

Figure 3 : Supprimer des tables de la base de données

On prend, pour exemple, la première table, nommée « Movies ». Les commandes de l'image ci-dessous définissent les colonnes de la table avec leurs types de données respectifs : la colonne **« mid »** qui est une clé primaire, et de type texte stocke l'identifiant du film, la colonne **« titleType »** de type texte stocke le type de titre, la colonne **« primaryTitle »** de type texte stocke le titre principal, la colonne **« originalTitle »** de type texte stocke le titre original, la colonne **« isAdult »** de type entier indique si le film est destiné à un public adulte, la colonne **« startYear »** de type date stocke l'année de début, la colonne **« endYear »** de type date stocke l'année de fin, la colonne **« runtimeMinutes »** de type entier stocke la durée du film en minutes.

```
cur.execute('''
    CREATE TABLE IF NOT EXISTS movies (
        mid TEXT,
        titleType TEXT,
        primaryTitle TEXT,
        originalTitle TEXT,
        isAdult INTEGER,
        startYear DATE,
        endYear DATE,
        runtimeMinutes INTEGER,
        PRIMARY KEY(mid)
```

Figure 4 : Créer une table Movies

2. Insertion

Pour chaque table, on ouvre son fichier CSV respectif situé dans le data set spécifié par la variable « **path** ». L'utilisation de la déclaration **« with open(...) as file: »** garantit que le fichier est fermé correctement, même en cas d'erreur lors de la lecture du fichier ou d'une exception. Ensuite, on ignore la première ligne qui est l'en-tête avec l'utilisation de la fonction **« next(...) »**, et on insère les données dans la table l'aide de la méthode **« executemany »**. La requête SQL utilise des paramètres de substitution (?) pour chaque colonne de la table. On prend dans l'image ci-dessous pour exemple la table « Movies ».

```
with open(path + "/movies.csv", "r") as file:
    content = csv.reader(file, delimiter=',')
    next(content)
    cur.executemany("INSERT INTO movies VALUES (?, ?, ?, ?, ?, ?, ?, ?)", content)
```

Figure 5 : Insérer les données dans une table Movies

A la fin, on utilise la fonction **« commit() »** pour valider tous les changements et les rendre permanents dans la base de données SQLite.

```
con.commit()
```

Figure 6 : Appliquer les ajouts à la base de données

b. Requêtes SQL sur la base

On écrit plusieurs requêtes mettant en oeuvre une ou plusieurs jointures sur l'ensemble des tables de la base. Les données insérées dans la base proviennent du data set « imdb-small ». Les réponses sont limitées à 5 ou moins d'éléments pour les afficher en évidence.

1. Première requête : Dans quels films a joué Jean Reno ?

```
cur.execute("CREATE INDEX IF NOT EXISTS idx_characters_pid ON characters(pid);")
cur.execute("CREATE INDEX IF NOT EXISTS idx_persons_primaryName ON persons(primaryName);")
res = cur.execute('''
                  EXPLAIN QUERY PLAN
                  SELECT m.primaryTitle FROM movies m
                  JOIN characters c ON m.mid = c.mid
                  JOIN persons p ON c.pid = p.pid
                  WHERE p.primaryName = 'Jean Reno'
```

Figure 7 : Requête SQL sur les films de Jean Reno

2. Deuxième requête : Quels sont les trois meilleurs films d'horreur des années 2000 au sens de la note moyenne par les utilisateurs ?

```
cur.execute("CREATE INDEX IF NOT EXISTS idx_genres_genre on genres(genre);")
res = cur.execute('''
                    SELECT m.primaryTitle, r.averageRating
                    FROM movies m
                    JOIN ratings r ON m.mid = r.mid
                    JOIN genres g ON m.mid = g.mid
                    WHERE g.genre = 'Horror'
                    AND m.startYear BETWEEN 2000 and 2009
                    ORDER BY r.averageRating DESC
                    LIMIT 3;
```

Figure 8 : Requête SQL sur les trois meilleurs films d'horreur des années 2000

3. Troisième requête : Quels sont les scénaristes qui ont écrit des films jamais diffusées en Espagne (région ES dans la table titles) ?

```
cur.execute("CREATE INDEX IF NOT EXISTS idx_region_titles ON titles(region);")
res = cur.execute('''
                    SELECT p.*
                      FROM persons p
                      WHERE p.pid IN (
                          SELECT w.pid
                          FROM writers w
                          WHERE w.mid NOT IN (
                              SELECT t.mid
                              FROM titles t
                              WHERE t.region = 'ES'
```

Figure 9 : Requête SQL sur les scénaristes qui n'ont jamais diffusés de films en Espagne

4. Quatrième requête : Quels acteurs ont joué le plus de rôles différents dans un même film ?

```
cur.execute("CREATE INDEX IF NOT EXISTS idx_characters_pid ON characters(pid);")
res = cur.execute('''
                    WITH max_roles_per_person_movie AS (
                      SELECT MAX(role_count) as max_roles
                        FROM (
                            SELECT COUNT(*) AS role_count
                            FROM persons p
                            JOIN characters c ON p.pid = c.pid
                            JOIN movies m ON c.mid = m.mid
                            GROUP BY c.pid, c.mid ))
                    SELECT p.primaryName, m.primaryTitle, COUNT(*) AS role_count
                    FROM persons p, max_roles_per_person_movie
                    JOIN characters c ON p.pid = c.pid
                    JOIN movies m ON c.mid = m.mid
                    GROUP BY c.pid, c.mid
                    HAVING role_count = max_roles
                    ORDER BY role_count DESC;
```

Figure 10 : Requête SQL sur les acteurs ayant le plus de rôles différents dans un même film

5. Cinquième requête : Quelles personnes ont vu leur carrière propulsée par Avatar et alors qu'elles n'étaient pas connues avant (ayant seulement participé à des films avec moins de 200000 votes avant Avatar), elles sont apparues par la suite dans un film à succès (nombre de votes > 200000) ?

```
cur.execute("CREATE INDEX IF NOT EXISTS idx_knownformovies_mid ON knownformovies(mid)")
cur.execute("CREATE INDEX IF NOT EXISTS idx_movies_primaryTitle ON movies(primaryTitle);")
cur.execute("CREATE INDEX IF NOT EXISTS index_movies_startYear ON movies(startYear);")
res = cur.execute('''
                SELECT DISTINCT persons.primaryName FROM persons
                INNER JOIN knownformovies ON persons.pid = knownformovies.pid
                INNER JOIN movies ON knownformovies.mid = movies.mid
                INNER JOIN ratings ON movies.mid = ratings.mid
                WHERE movies.startYear < (SELECT startYear FROM movies WHERE primaryTitle = 'Avatar')
                AND ratings.numVotes < 200000
                AND movies.mid NOT IN (
                  SELECT movies.mid
                  FROM movies
                  INNER JOIN ratings ON movies.mid = ratings.mid
                  WHERE movies.startYear >= (SELECT startYear FROM movies WHERE primaryTitle = 'Avatar')
                AND ratings.numVotes > 200000)
```

Figure 11 : Requête SQL sur les personnes moins connues avant Avatar, et apparues après dans un film à succès

c. Performances

Lorsque l'on utilise une base suffisamment grande, les requêtes avec plusieurs jointures peuvent durer longtemps. On crée des index dans SQLite pour accélérer le traitement de ces dernières, mais cela a un coût en taille de la base de données.

Tableau 1 : Taille de la base de données en fonction du data set et des indexes

Data set	Tiny	Small	Medium	Full
Taille sans indexes	11.21 MB	137.21 MB	683.76 MB	11719.80 MB
Taille avec indexes	**13.44 MB**	**160.35 MB**	**789.62 MB**	**13497.75 MB**

L'indexation entraîne une augmentation de la taille des bases de données, comme le montrent les tailles de bases de données "avec indexes" par rapport à celles "sans indexes". Les temps d'exécution des requêtes ont tendance à augmenter avec la taille de la base de données. Cela est principalement dû à l'augmentation du temps nécessaire pour parcourir et rechercher des données dans des ensembles de données plus vastes. Les requêtes sur des bases de données plus grandes prennent généralement plus de temps à s'exécuter que celles sur des bases de données plus petites, même lorsque des indexes sont utilisés pour optimiser les recherches.

On a alors chronométré la durée d'exécution des requêtes avec le module Python « time » sur chaque data set. On prend en compte l'affichage de l'intégralité des données récupérées dans le temps de la requête.

Tableau 2 : Performance en seconde de chaque requête dans chaque jeu de données

Requêtes	1	2	3	4	5
Tiny	0.0055s	0.0005s	0.0077s	0.0167s	0.0253s
Tiny avec indexes	**0.0027s**	**0.0042s**	**0.0039s**	**0.0184s**	**0.0112s**
Small	0.2032s	0.0077s	0.1848s	0.4664s	0.7035s
Small avec indexes	**0.0004s**	**0.0045s**	**0.1207s**	**0.6183s**	**0.5663s**
Medium	2.2557s	0.0608s	1.4059s	5.1817s	6.0820s
Medium avec indexes	**0.0056s**	**0.0407s**	**1.3289s**	**7.4247s**	**4.3741s**
Full	61.8598s	3.3360s	18.7354s	136.9553s	53.9861s
Full avec indexes	**0.0873s**	**6.2808s**	**20.4237s**	**471.8552s**	**12.5487s**

Globalement, l'utilisation d'index semble considérablement réduire les temps d'exécution des requêtes. Cela est particulièrement évident dans les catégories **"Small", "Medium", et "Full"**. Les temps d'exécution des requêtes sans index sont généralement plus longs que ceux avec index, suggérant que l'indexation améliore les performances des requêtes en réduisant le temps nécessaire pour rechercher des données, mais le tableau nous montre que les indexes peuvent aussi augmenter ce temps.

Les temps d'exécution des différentes requêtes varient en fonction de la nature de la recherche effectuée. On peut voir les étapes de la recherche grâce à la commande **« EXPLAIN QUERY PLAN »**. Par exemple, les requêtes impliquant des opérations de recherche sur des champs spécifiques de tables (**« SEARCH »**) semblent être plus rapides que celles nécessitant des balayages sur plusieurs tables (**« SCAN »**). De plus, le balayage est plus sensible à la taille de la base de données.

Alors, l'indexation joue un rôle dans l'amélioration des performances des requêtes, mais d'autres facteurs tels que la taille de la base de données et la nature de la requête peuvent également influencer les temps d'exécution. La taille de la base de données a un impact significatif sur ces performances et nécessite une gestion et une optimisation attentives pour garantir des opérations efficaces en fonction de la nature des requêtes, surtout lorsque la taille de la base de données devient importante.

3. MongoDB

a. Insertion de données plates

1. Création

On crée la base de données grâce à Mongo Shell avec la commande « **use** » et le nom de la base de données. Ensuite, on reproduit à l'identique la base SQL dans MongoDB, chaque table devenant une collection. On utilise alors la commande « **createCollection()** ».

```
test> use imdb
switched to db imdb
imdb> db
imdb
```

Figure 12 : Créer une base de données MongoDB

```
imdb> db.createCollection("movies")
{ ok: 1 }
imdb> show collections
movies
imdb> db.createCollection("persons")
{ ok: 1 }
imdb> show collections
movies
persons
```

Figure 13 : Créer les collections MongoDB

2. Insertion

On alimente la base MongoDB à l'aide de requêtes SQL sur la base SQLite avec la fonction « **export_sqliteTable_to_mongoCollection()** ». Tout d'abord, la fonction établit une connexion à la base de données SQLite spécifiée, récupère les noms de champs de la table donnée, puis établit une connexion à la base de données MongoDB à l'aide de l'URL fournie. Puis, elle supprime toutes les données existantes dans la collection MongoDB cible dans le cas où on change de data set. Ensuite, elle extrait toutes les lignes de la table SQLite et les insère dans la collection MongoDB sous forme de documents, en utilisant les noms de champs extraits précédemment pour créer des clés dans les documents MongoDB. Enfin, une fois le transfert terminé, les connexions à la base de données SQLite et à MongoDB sont fermées.

Ensuite, on recrée à l'aide de requêtes MongoDB et de scripts Python les requêtes SQL sur la base. Voici les erreurs qu'on a eu lors des tests : dans le set de données « Full », les requêtes 3 et 5 sont de type « **DocumentTooLarge** » et la requête 4 a une **très longue attente**, dans le set de données « Medium », la requête 4 a une **très longue attente** aussi.

Tableau 3 : Performance en seconde de chaque requête dans chaque jeu de données

Requête	1	2	3	4	5
Tiny	0.0042s	0.0054s	0.0267s	0.0542s	0.0249s
Small	0.0292s	0.0444s	0.2955s	5.6372s	0.5566s
Medium	0.2369s	0.3016s	2.8191s		4.3715s
Full	8.2028s	7.3015s			

b. Insertion de données structurées

Lorsqu'on compare les temps de récupération d'un film dans la nouvelle collection avec ceux des requêtes simulant des jointures entre les tables, on constate, lors des tests avec le jeu de données « tiny », une différence négligeable. Cependant, il est crucial de noter que nous n'avons pas inclus les jointures retours nécessaires pour récupérer les informations spécifiques à chaque acteur ou directeur, par exemple.

Cette observation nous conduit à conclure que les requêtes dans une collection structurée sont généralement plus rapides. En effet, elles évitent le coût de fusionner des documents à partir de différentes collections, une opération qui se révèle particulièrement coûteuse en termes de performances. Seulement, plus la base de données est grande, plus on met de temps pour créer cette collection structurée. On remarque une très grosse différence entre les jeux de données « tiny » et « small ». Respectivement, le temps pour insérer les informations est de 49.302 secondes alors que l'autre est indéterminé.

c. Résistance aux pannes

On crée deux instances : « **mongod –port 27017 –dbpath ./mongod/db1 –replSet rs0** » et « **mongod –port 27018 –dbpath ./mongod/db2 –replSet rs0** ».Puis, on se connecte dessus : « **mongosh –port 27017 & mongosh –port 27018** ». Sur la première instance, on effectue la commande « **rs.status()** », permettant de savoir si le replicaSet est initialisé, normalement non lors de la première fois. Ensuite, on intialise le replicaSet avec la commande « **rs.initiate()** ».

```
MongoServerError[NotYetInitialized]: no replset config has been received
```

Figure 14 : Pas de replicaSet initialisé

```
rs0 [direct: primary] test> rs.status()
{
  set: 'rs0',
```

Figure 15 : ReplicaSet initialisé et première instance de type primaire

Au départ, il y a qu'un membre établi dans le replicaSet, et c'est la première instance. Il faut donc, ajouter notre deuxième membre avec la commande « rs.add() ». Maintenant, notre replicaSet comporte deux membres. Lorsqu'on se connecte sur la deuxième instance, on remarque que le replicaSet est bien actif.

```
rs0 [direct: primary] test> rs.add("localhost:27018");
{
  ok: 1,
```

Figure 16 : Second membre du replicaSet

```
rs0 [direct: secondary] test>
```

Figure 17 : Deuxième instance de type secondaire

On tue l'instance primaire, et on remarque que l'instance secondaire notifie qu'il a perdu contact avec l'instance primaire. Alors, en se reconnectant à la deuxième instance, elle s'élit instance primaire. Puis, la première instance, tombé en panne, se reconnecte et on remarque qu'elle est devenue secondaire.

```
members: [
  {
    _id: 0,
    name: 'localhost:27017',
    health: 1,
    state: 2,
    stateStr: 'SECONDARY',
  {
    _id: 1,
    name: 'localhost:27018',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
```

Figure 18 : Première instance tombé en panne & remise en route

On peut conclure que la configuration en replicaSet permet de résister aux pannes. Elle redistribue la priorité à l'une des instances à jour lorsque l'instance primaire tombe en panne.

d. Synchronisation bidirectionnelle

1. Implémentation coté Sqlite

Pour implémenter la synchronisation entre les bases de données SQLite et MongoDB, nous avons procédé par plusieurs étapes. Tout d'abord, nous avons défini une fonction nommée « **sqlite_trigger_handler** » chargée de gérer la synchronisation des données en fonction de l'opération effectuée (insertion, mise à jour ou suppression), des données de la ligne modifiée et du nom de la table concernée. Nous avons établi une connexion à la base de données MongoDB, identifiant la base de données « **imdb** », puis avons créé un dictionnaire « **collections_mapping** » associant les tables SQLite aux collections MongoDB correspondantes. Les opérations de synchronisation ont été

effectuées en fonction de l'opération spécifiée, insérant, mettant à jour ou supprimant des documents dans les collections MongoDB appropriées. Des déclencheurs ont été configurés dans SQLite pour appeler la fonction « **sqlite_trigger_handler** » lorsqu'une opération est effectuée sur une table donnée.

2. Implémentation coté MongoDB

Initialement, on a instauré les connexions à la base de données SQLite et MongoDB à l'aide des fonctions dédiées « **init_sqlite_connection** » et « **init_mongodb_connection** ». Ensuite, pour surveiller en temps réel les modifications dans les collections MongoDB, on a utilisé les « changestreams » intégrés à MongoDB, offrant ainsi une détection instantanée des opérations d'insertion, de mise à jour et de suppression. Pour gérer ces opérations détectées, on a mis en place des fonctions spécifiques telles que « **handle_insert_operation** », « **handle_update_operation** », et « **handle_delete_operation** », chacune traitant respectivement les différentes actions de manière appropriée afin d'assurer la cohérence des données dans SQLite.