Haute école de Gestion

PROJET NOSQL

Christophe Berger
Victor Hüni

23.12.2022

1 CONTEXTE

Nous sommes un e-commerce mondial spécialisé dans la distribution de produits culturels. Nous proposons à ce jour 4 types de produits différents, à savoir des films, des livres, des albums de musique et des jeux-vidéos.

Les objectifs que nous cherchons à atteindre dans ce projet sont :

- Centraliser les informations intrinsèques à nos produits
- Faciliter l'exploration de notre catalogue à nos clients
- Calculer des indicateurs pertinents pour nos équipes de marketing et de ventes

2 DONNÉES

Afin de simuler au mieux notre catalogue de produit, nous souhaitions obtenir une certaine variété de produit (ici 4) avec une certaine disparité dans leurs attributs mais en gardant cohérence quant à notre contexte. Nous sommes donc parties à la recherche de datasets de produits culturels sur Kaggle.com. Voici là liste de ces derniers :

- 1000 Movies from IMDB
  https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows

- 10000 Books from Goodreads
  https://www.kaggle.com/datasets/jealousleopard/goodreadsbooks

- 5000 Albums from Rollings Stones
  https://www.kaggle.com/datasets/notgibs/500-greatest-albums-of-all-time-rolling-stones

- ~200 Video Games
  https://www.kaggle.com/datasets/atharvaingle/video-games-dataset

Au final c'est un catalogue de plus de 17k produits différents (surtout des livres) que nous avons réussi à composer. Voici le dictionnaire de données de ces 4 datasets originaux.

Album

Album	int
Album	string
Artist Name	string
Release Date	date
Genres	list with ,
Descriptors	list with ,
Average Rating	double
Number of Ratings	long
Number of Reviews	long

Book

bookID	int
title	string
authors	list with /
average_rating	double
isbn	string
isbn13	string
language_code	string
num_pages	int
ratings_count	long
text_reviews_count	long
publication_date	date
publisher	string

Movie

Poster_Link	int
Series_Title	string
Released_Year	int
Certificate	string
Runtime	int
Genre	list with ,
IMDB_Rating	double
Overview	string
Meta_score	int
Director	string
Star1	string
Star2	string
Star3	string
Star4	string
No_of_Votes	long
Gross	long

Video Game

Rank	int
Name	string
Platform	string
Year	int
Genre	string
Publisher	string
NorthAmerica_Sales	double
EurpeanUnion_Sales	double
Japan_Sales	double
Other_Sales	double
Global_Sales	double

3 SÉLECTION DE LA BASE DE DONNÉES

Lors de notre recherche du type de base de données le plus pertinent pour répondre à notre besoin nous avons identifié quelques contraintes :

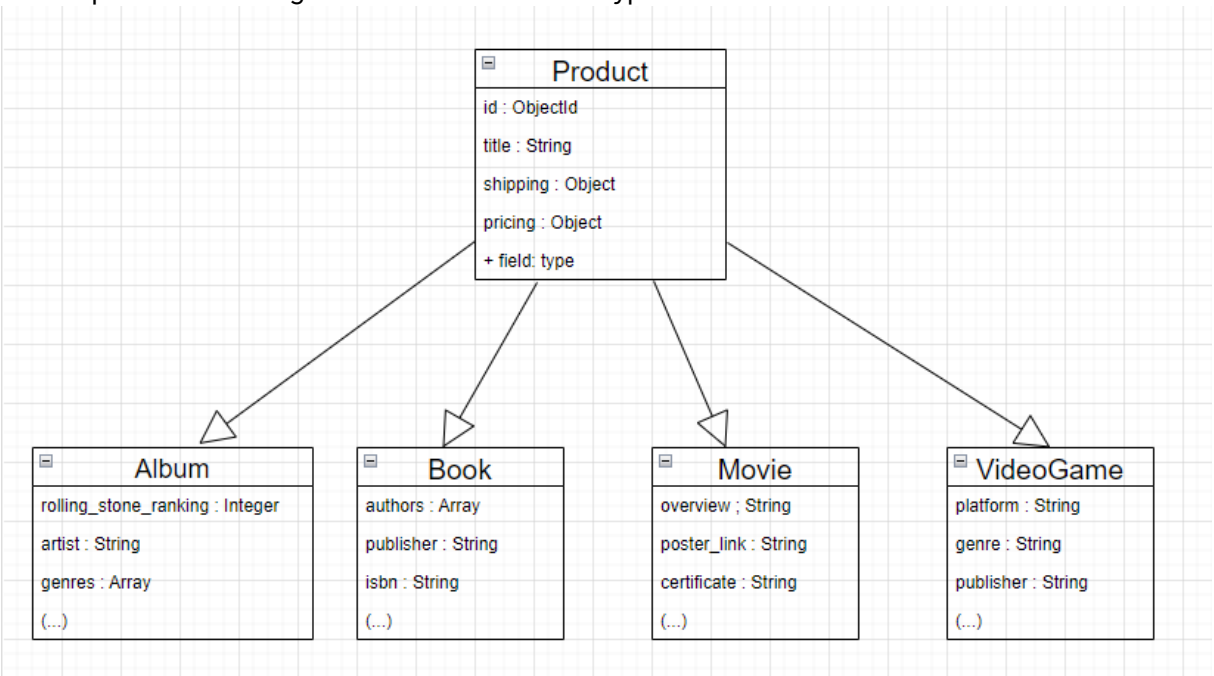
1. L'hétérogénéité de nos données et de leurs formats
2. La nécessité pour notre catalogue d'évolué dans le temps (ajout de nouveau produits, ajout de nouveau attributs...)
3. Notre catalogue doit être à jours en tout temps pour tous nos clients dans le monde entier
4. La disponibilité de donnée doit être assurée tant pour les recherches clients que pour nos calculs d'indicateurs

Les 2 premières contraintes nous amènent naturellement vers les bases de données NoSQL. En en effet, la gestion de données hétérogène et la nécessité de schéma évolutif nous éloigne grandement des schémas relationnels rigide. Afin de confirmer cela, nous allons nous attarder sur les potentiel implémentation en relationnel au prochain chapitre.

Les contraintes 2 et 4 sont respectivement des contraintes liées à la distributivité et à la disponibilité de nos données. En effet un magasin de e-commerce doit pouvoir assurer que son catalogue de produits reste accessible de manière concurrente et à jour 24h/24h dans le monde entier. La partition des données (distributivité) et la disponibilité au sens du théorème de CAP ne font donc aucun doute. Pour ce qui est de consistance des données, celle-ci n'a pas à être totale en tout temps mais le système doit tendre vers celle-ci. Ce constat nous dirige une fois de plus vers une base de données NoSQL. Mais laquelle ?

4 ANALYSE DE DIVERS ARCHITECTURES

Dans un monde relationnel et dans un paradigme objet, notre use case se désignerai la plus part du temps avec une diagramme de classe de ce type.



Une Classe abstraite Produit et une classe concrète pour chaque type de produit spécifique qui hérite des attributs de la classe Produit.

4.1 IMPLÉMENTATION EN RELATIONNEL

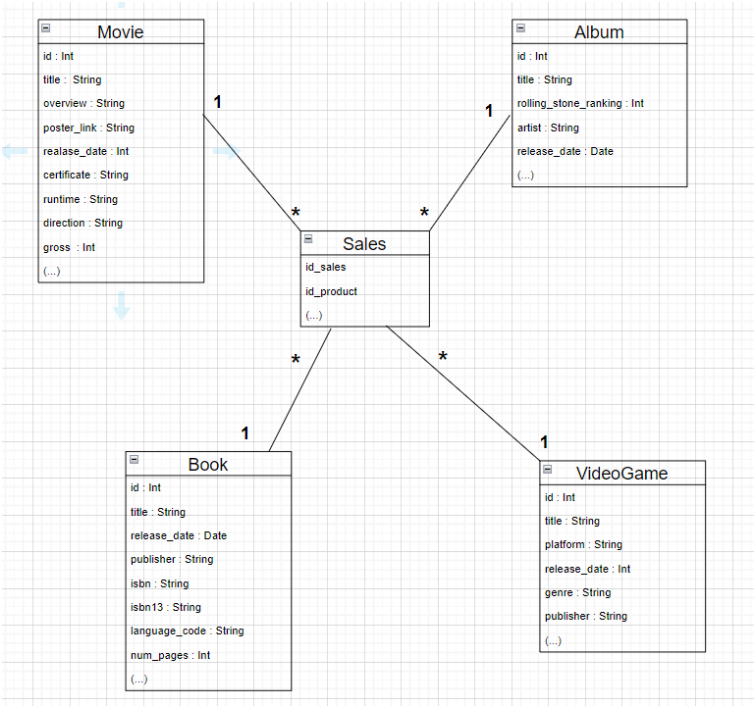
Avant de nos attarder sur notre choix de la bonne base NoSQL, faisons un zoom sur une potentielle implémentation d'un catalogue de produit en relationnel avec la modélisation présentée ci-dessus. Outre le fait que Disponibilité des données n'est pas au rendez-vous dans un système relationnel, nous avons tout de même fais l'exercice du design de l'implémentation de notre solution afin de bien saisir l'avantage comparatif du NoSQL. Ainsi, nous espérons pouvoir mettre en exergues les contraintes liés au développement et à l'exploitation d'un tel système de BDD dans notre cas.

Il existe 4 manières d'implémenter l'héritage dans un système de base de données relationnelle

- Concrete Table Inheritance
- Single Table Inheritance
- Multiple Table Inheritance
- Entry Attributes Values

4.1.1 CONCRETE TABLE INHERITANCE

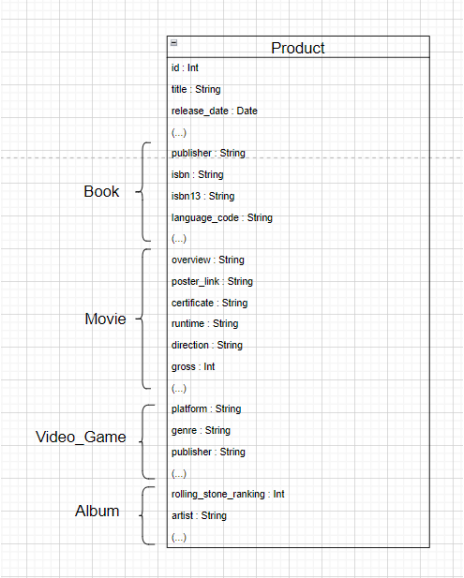
Dans ce modèle d'architecture, il sera nécessaire de créer une table pour chaque classe concrète de l'application. Ainsi chaque type de produit possèdera sa table, en l'occurrence, une table 'movie', une table 'book', une table 'cd' et une table 'video game'.



À chaque fois que nous souhaiterons incorporer une nouvelle catégorie de produit à notre catalogue, il sera donc nécessaire de créer une nouvelle table pour la catégorie de produit concernée. Les champs communs à chaque produit (le prix, les dimensions...) sont répétés dans chaque table. Toutes les requêtes spécifiques au Produit devront être réécrites pour chacune des catégories. Le modèle est peu évolutif et très rigide. Il ne répond pas à nos 2 premières contraintes.

4.1.2 SINGLE TABLE INHERITANCE

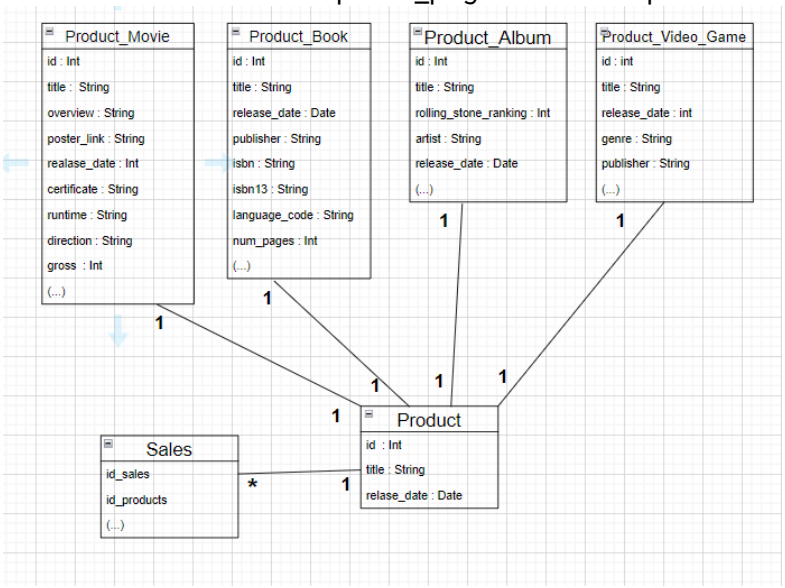
Cette approche consiste à créer une seule et même table qui regroupe toutes les catégories de produits, avec le besoin d'ajouter de nouvelles colonnes (un nouveau set d'attributs) à chaque fois que nous souhaiterons insérer un nouveau type de produit dans notre catalogue.



L'avantage par rapport à la solution précédente (Concrete Table Inheritance) est relatif au fait qu'une même requête pourra s'appliquer aux différents types de produits. En effet, ce modèle d'architecture limite les JOIN dans les requêtes et homogénéise la structure de ces dernières pour tous les types de produits. L'ajout d'un nouveau type de produit est possible mais uniquement via la modification du schéma et l'ajout de nouvelle colonne.
De plus, nous assistons ici à un gaspillage d'espace mémoire avec un nombre important de champ qui resteront vide pour produit/ligne. Encore une fois le schéma rigide empêche le respect de plusieurs de nos contraintes.

4.1.3 MULTIPLE TABLE INHERITANCE

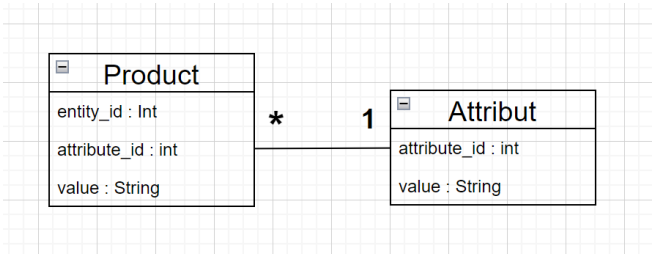
Dans cette solution, nous créons d'abord une table pour chaque classe concrète mais également abstraite. Ainsi une table générique 'produit', qui regroupe les attributs communs à chaque type de produit sera créé pour centraliser les données communes de tous les produits. Ensuite, nous créons des tables spécifiques pour chaque type de produits qui contiendront uniquement leurs champs spécifiques / propres / individuels. Ainsi le champs « runtime » sera bel et bien uniquement dans la table Movie et le champ « nb_pages » sera uniquement dans la table Livre



Cette solution aura l'avantage d'être moins gourmande en espace (mémoire) par rapport à la solution 'Single Table Inheritance' et elle offrira également plus de flexibilité que la solution 'Concrete Table Inheritance', Cependant, l'ajout de nouveau type de produit verra forcément une modification du schéma avec la création de table, et beaucoup de JOIN devront être utilisé lors des requêtes de calculs et d'affichage se qui péjorera les performances.

4.1.4 ENTITIES ATTRIBUTES VALUES

Cette ultime solution est certainement la plus flexible des implémentations relationnelles vu jusqu'à présent. Ici, nous ne créerons qu'une table Produit avec trois colonne : Id du produit/entité, id de l'attribut représenté ainsi qu'une colonne représentant la valeur de ce dernier.



Entity	Attribute	Value
1	title	"The Godfather"
1	certificate	"A"
1	runtime	175
1	direction	"Francis Ford Coppola"
(...)	(...)	(...)

L'avantage que possède cette architecture par rapport à toutes les autres démontrés jusqu'à présent c'est la flexibilité de son schéma. En effet, l'ajout d'attribut ou de nouveau type de produit ne pose plus aucun problème et ne provoque aucune modification de schéma. Ainsi ce modèle répond à deux de nos contraintes majeures en termes d'évolution dans le temps de nos types et attributs de produits. Cependant, il est fort probable que la majorité des requête complexe lié au calcul d'indicateurs et autres agrégations auront recours à l'utilisation massive de JOIN et SUBQUERY. La performance risque donc d'être fortement impactées et ne répondra pas au contrainte de disponibilité.

4.2 IMPLÉMENTATION EN NOSQL

4.2.1 IMPLÉMENTATION EN BASE DE DONNÉE GRAPH

Dans le contexte d'une simulation d'un catalogue de produits comme le nôtre, notre objectif ici est le stockage de données en tenant compte de leur hétérogénéité, de manière souple et non structurée, et non pas les interactions entre les entités que nous stockons.

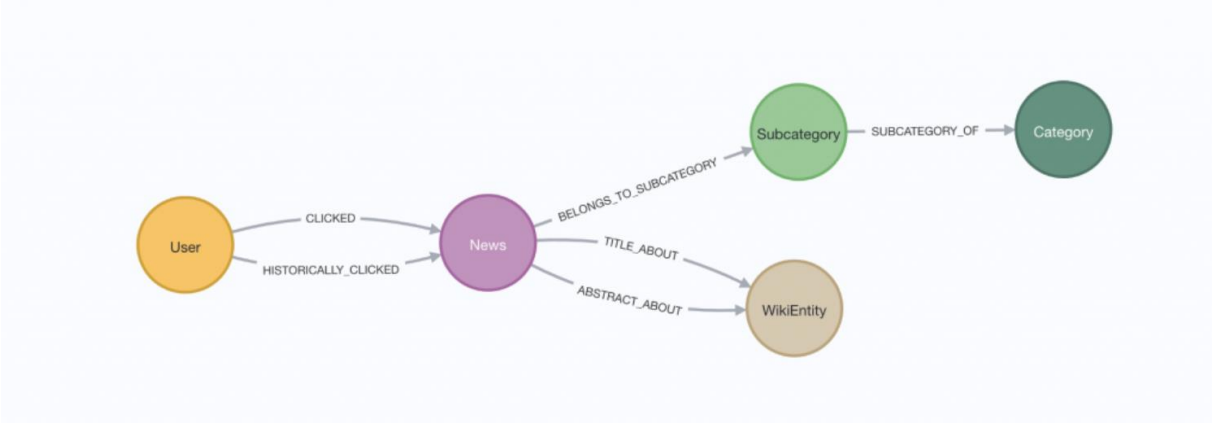
Si nous nous étions tournés vers Neo4J, il aurait fallu dans un premier temps déterminer les relations entre nos différents objets, ce qui n'est pas l'objectif de notre cas d'application en l'espèce. Cibler (Mettre en évidence) les relations entre les produits ne nous intéressent pas dans la gestion de notre catalogue. Nous ne souhaitons ni savoir quel acteur a joué dans tel film, ni quel auteur a écrit quel livre et enregistré quel album. En effet par nature, un modèle de catalogue ne s'intéresse pas aux relations mais à son contenu hétéroclite.

En effet, dans le cas d'une implémentation avec Neo4j après la définition des différents types de nodes avec leurs attributs représentant les différents produits, il nous incomberait de lier les produits entre eux d'une manière ou d'une autre. L'ajout de nodes de genres, d'auteur/acteur/artiste ou encore de tags pourraient nous permettre de déployer et d'utiliser le moteur graphes à bon escient mais ces relations ne sont pas évidentes au premier abord pour notre use case.

Les avantages offerts par une base de données orientée graphe (adaptation aux objets complexes organisés en réseaux avec beaucoup de liens, grande rapidité pour manipuler des données fortement connectées...etc) ne sont pas pertinents pour la gestion de notre catalogue de produits dans la mesure où ce qui nous intéressé c'est le contenu et pas les relations entre les objets. On compte parmi les exemples d'utilisation les plus communs pour les bases de données orientées graphe :

- Les réseaux sociaux
- Les moteurs de recommandations
- Les données géospatiales, les calculs d'itinéraires

Un moteur de recommandation serait le use case parfait pour étendre les capacité de notre projet avec l'ajout des nodes d'utilisateurs et les relations avec les produits à l'instar de ce modèle de recommandations de news.



Pour résumer, Il devient clair qu'une base orientée graph n'est pas adapté à notre situation même si ces dernières répondent à l'ensemble de nos contraintes (schéma évolutif, partition tolerant, disponibilité...). Les raisons de cette non pertinence sont les suivantes :

- Nos données sont déconnectées et les relations n'importe pas

- Notre partie analytique ce concentre sur le contenu et non sur les relations

5 Implémentation en Base de données Document

Dans ce contexte, nous nous sommes naturellement tournés vers les bases données orientées document, notamment MongoDB. Mais quelles sont les avantages des bases de données documents dans le cas d'un catalogue ?

5.1 Les avantages

5.1.1 SCAABILITÉ

MongoDB est conçu pour être horizontalement évolutif, ce qui signifie que vous pouvez facilement ajouter des serveurs pour gérer une croissance de votre catalogue de produits. (A REFORUMLER)

5.1.2 FLEXIBILITÉ

Grâce à son modèle de données orienté document, il est possible de procéder au stockage de nos données de manière flexible et non structurée. Ainsi, nous sommes en mesure d'ajouter de nouvelles catégories de produits quand cela est nécessaire, sans la contrainte de devoir restructurer notre schéma d'origine.

5.1.3 PERFORMANCE

Grâce à son haut niveau de performance, MongoDB nous permet de traiter une grande quantité de données, et son utilisation s'applique donc parfaitement à la situation d'espèce, à savoir la gestion d'un catalogue de produits (plus de 17'000 au total).

5.1.4 INDEXATION

Grâce à la création d'index proposée par MongoDB, nous sommes en mesure d'optimiser les différentes requêtes de recherche des différents produits contenus dans notre catalogue

Nous avons fait le choix de nous tourner vers MongoDB, une base de données orientée "documents" en raison de l'hétérogénéité des données que nous serons amenés à traiter.

La problématique réside ici dans la capacité de stocker plusieurs types d'objets avec différents sets d'attributs. C'est dans ce genre de collection de données que le modèle MongoDB s'avère particulièrement utile.

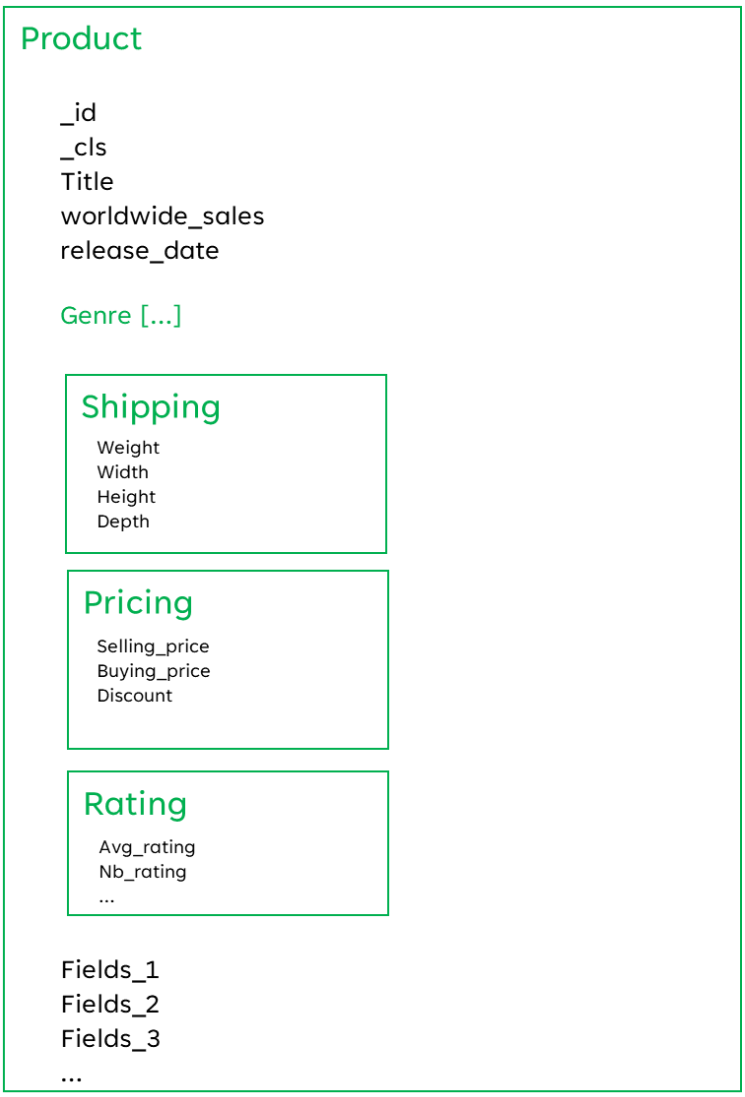
5.2 MODÉLISATION DES DOCUMENTS

5.2.1 COLLECTION

Notre base de données ne se constituera d'une seule et même collection que nous appellerons Products. Une seule collection suffit à gérer l'ensemble de nos produits en un seule et même endroit. Aucune jointure et ni lookup n'est nécessaire pour réaliser les quelques objectifs que nous étions fixé.

5.2.2 DOCUMENTS

Une fois la collection définie, nous devions nous attardez sur la modélisation des documents. Vous trouvez ci-dessous le modèle conceptuel de notre document Produit arrêté pour notre solution :



Les documents de chaque type de produit se compose de deux parties distinctes la partie communes à tous les produits et la partie « custom » pour chaque type de produit.

Voici une liste exhaustive des champs de produits communs à tous les types de produits que vous pourrez retrouvez dans tous les documents.

Champs Communs	Type	Description
_id	ObjectId	primary Key
_cls	String	Référence du POJO à utiliser pendant le désérialization. Indique également le type de produit
Shipping	Object	Données dimensionnelle
Pricing		Object Données Financière
Rating	Object	Données de Popularité
Worldwide_sales	Long	Statistique du CA Mondiale de chaque produit
delease_date	Int/Date	Date de la sortie du produit avec la particularité d'être soit une date soit une année. Le type diffère entre les produits
Genres	Array	Liste des genres associées à une oeuvre

Voici la liste non exhaustive des champs de produits spécifiques à certain type de produits

Champs Spécifique	Type	Description
runtime	int	La dureé d'un film
Num pages	int	Le nombre de page d'un livre
publisher	String	Éditeur d'un livre ou d'un jeu vidéo
overview	String	Synopsis du Film
cast	Array	Casting des acteurs présent dans le film
authors	Array	Listes des auteurs d'un livre
...		

Dessous vous trouverez les snapshots des documents chargés dans la collection en fonction de leurs types.

Movie


```
  _id: ObjectId('63a56c0b2511cd0ede35cd81')
  _cls: "domaine.Movie"
  title: "The Shawshank Redemption"
> shipping: Object
v pricing: Object
    _cls: "Pricing"
    selling_price: 14
    buying_price: 1
    discount: 0.8553351024045016
> rating: Object
  worldwide_sales: 28341469
  release_date: 1994
  overview: "Two imprisoned men bond over a number of years, finding solace and eve…"
  poster_link: "https://m.media-amazon.com/images/M/MV5BMDFkYTc0MGEtZmNhMC00ZDIzLWFmNT…"
  certificate: "A"
  runtime: 142
v genres: Array
    0: "Drama"
  direction: "Frank Darabont"
v cast: Array
    0: "Tim Robbins"
    1: "Morgan Freeman"
    2: "Bob Gunton"
    3: "William Sadler"
```

Book


```
_id: ObjectId('63a56c112511cd0ede35e4f1')
_cls: "domaine.Book"
title: "Harry Potter and the Half-Blood Prince (Harry Potter  #6)"
shipping: Object
  weight: 1.098810095715601
  width: 191
  heigth: 136
  depth: 15
pricing: Object
rating: Object
worldwide_sales: 42609823
release_date: 2006-09-15T22:00:00.000+00:00
authors: Array
  0: "J.K. Rowling"
  1: "Mary GrandPré"
publisher: "Scholastic Inc."
isbn: "0439785960"
isbn13: "9780439785969"
language_code: "eng"
num_pages: 652
```


Album


```
_id: ObjectId('63a56c0d2511cd0ede35d169')
_cls: "domaine.Album"
title: "OK Computer"
shipping: Object
pricing: Object
rating: Object
  avg_rating: 4.23
  nb_rating: 70382
  nb_review: 1531
  rolling_stone_ranking: 1
worldwide_sales: 5790155
release_date: 1997-06-15T22:00:00.000+00:00
artist: "Radiohead"
genres: Array
  0: "Alternative Rock"
  1: "Art Rock"
descriptors: Array
  0: "melancholic"
  1: " anxious"
  2: " futuristic"
  3: " alienation"
  4: " existential"
  5: " male vocals"
  6: " atmospheric"
  7: " lonely"
  8: " cold"
  9: " introspective"
```

Video Game


```
  _id: ObjectId('63a56c1a2511cd0ede361068')
  _cls: "domaine.VideoGame"
  title: "Wii Sports"
> shipping: Object
v pricing: Object
    _cls: "Pricing"
    selling_price: 52
    buying_price: 18
    discount: 0.3715271202613878
  platform: "Wii"
  release_date: 2006
v genres: Array
    0: "Sports"
  publisher: "Nintendo"
  worldwide_sales: 82740000
```


5.3 ARCHITECTURE DE L'APPLCATION

5.3.1 INTERACTION AVEC LA BASE

Classe concernée s:

- dao.Bdd

Le client de base de données et la récupération de la collection Product s'effectue via cette Classe qui utilisent le design pattern du Singleton pour distribuer son utilisation à travers l'application sans dupliquer ces instances inutilement.

5.3.2 CHARGEMENT DES DONNÉES

Classes concernées :

- Dao.DataLoader
- Dao.FileToStr

Pour charger les différents csv dans la collection, la classe Data Loader est utilisé. Cette dernière possède une méthode publique par type de produits :

- loadMovie
- loadAlbums
- loadVideoGames
- loadBook

Chacune de ces méthodes exploites les méthodes de la class Dao.FileToStr pour lire les csv et les désérialiser dans des Arrays.

Puis le choix a été fait pour chaque ligne des csv de générer des Documents Bson manuellement avant de les insérer dans la collection. Lors de cette étape, il les différentes méthodes cast les différents attributs et génères également des données random afin de créer des données

cohérentes. Par exemple les chiffres d'affaires mondiaux ont été généré aléatoirement pour les albums et les livres.

Le choix d'insérer directement des Bson Documents et non des POJOS sérialisé est délibéré car le use case du chargement en masse ne nous semblait pas pertinent pour la sérialisation des POJO.

5.3.3 EXÉCUTION DES QUERY

Classes concernées :

- Toutes les classes du packages Domaine
- Toutes les classes du packages Agrégations

La stratégie choisie ici pour exécuter les requêtes et manipulé les résultats est l'utilisation des Codecs Automatic présent dans le Driver Java.

En effet toutes les classes de ces domaines ont été annotés avec les descripteurs dédié à l'encodage/décodage des POJO to Bson et inversement. De plus le client mongodb est initialisé avec les registres de codecs automatiques qui eux sont en charge de scannée les classes de l'application et ainsi géré les règles de sérializazion et désérialization des Bson Document au POJO.

```java
@BsonDiscriminator(value="domaine.VideoGame", key="_cls")
public class VideoGame extends Product {
    @BsonProperty("platform")
    private String platform;
    @BsonProperty("release_date")
    private int releaseDate;
    @BsonProperty("genres")
    private List<String> genres;
    @BsonProperty("publisher")
    private String publisher;
    @BsonProperty("sale")
    private Sale sale;

    @BsonCreator
    public VideoGame(@BsonProperty("_id") ObjectId id,
                     @BsonProperty("title") String title,
                     @BsonProperty("shipping") Shipping shipping,
                     @BsonProperty("pricing") Pricing pricing,
                     @BsonProperty("platform") String platform,
                     @BsonProperty("release_date") int releaseDate,
```

```java
PojoCodecProvider defaultPojoCodecProvider = PojoCodecProvider.builder()
                                                              .automatic(true)
                                                              .conventions(Arrays.asList(Conventions.ANNO

CodecRegistry fromProvider = CodecRegistries.fromProviders(defaultPojoCodecProvider);
CodecRegistry defaultCodecRegistry = MongoClientSettings.getDefaultCodecRegistry();

CodecRegistry pojoCodecRegistry = CodecRegistries.fromRegistries(defaultCodecRegistry, fromProvider);
```

Le résultat de cette stratégie est que l'application est désormais capable de reconnaitre le type de document reçu en résultat d'une requête et d'instancier le bon objet pour permettre une manipulation adaptée lors de sa vie dans l'application. Le véritable défi a été de configurer cela pour l'ensemble des cas de figure complexe :

- L'héritage
- Les classes parents/enfants

L'ensemble de ces cas ont été pris en compte et à la manière d'un ORM en relationnel la sérialization/désérialization des objets et documents est désormais facilité.

6 QUERIES

Vous trouverez ci-joint le descriptif et justificatifs des 3 query que nous avons sélectionné pour notre cas d'usage.

6.1 USE CASE 1

Trouver des produits avec une simple chaine de caractère

Je suis un client et je souhaiterais rechercher tous les articles concernant un certain sujet. En tant qu'utilisateur, je souhaite pouvoir rechercher des produits avec une barre de recherche afin de facilité mon interaction avec le grand catalogue et éviter de me perdre.

Les champs sur lesquels cette requêtes s'exécutent sont les suivants :

```
if(searchType == SearchType.Title) {
    idx = Indexes.compoundIndex(
            Indexes.text( fieldName: "title"),
            Indexes.text( fieldName: "overview")
    );
}else if(searchType == SearchType.Persons) {
    idx = Indexes.compoundIndex(
            Indexes.text( fieldName: "authors"),
            Indexes.text( fieldName: "cast"),
            Indexes.text( fieldName: "direction"),
            Indexes.text( fieldName: "artist"),
            Indexes.text( fieldName: "publisher")
    );
}else{
    idx = Indexes.compoundIndex(
            Indexes.text( fieldName: "title"),
            Indexes.text( fieldName: "overview"),
            Indexes.text( fieldName: "authors"),
            Indexes.text( fieldName: "cast"),
            Indexes.text( fieldName: "direction"),
            Indexes.text( fieldName: "artist"),
            Indexes.text( fieldName: "publisher")
    );
```

En fonction du type de recherce souhaités

- par Titre/description

- par Persons
- dans tous les champs

L'application supprime et régénère l'indexation Text de la collection afin de permettre à la requête de s'exécuté de manière optimale.

6.1.1 FONCTIONNEMENT

Comment fonctionne cette requête ?

- Gestion de l'indexations texte dans la collection
- Combinaison des opérateurs $text / $search
- Tri des résultats en fonction de la pertinence des résultats (Utilisation de l'opérateur $meta)

Une recherche de texte dans tous ces différents champs en relationnel reviendrait à chainer les LIKE opérateur dans chacun des champs et une requête par type de recherche devra être écrite et exécuté. Ici la simple application du filtre texte avec la combinaison des deux opérateurs ci-dessous permet une recherche de texte complète sans modification de query.

```
{ $text: { $search: "Lord of" } }
```

La deuxième particularité de cette requête réside dans l'utilistion des meta données généré de la requête.

```
Filter    { $text: { $search: "Lord of" } }
Project   {score: { $meta: "textScore" }}
Sort      {score: 1}
```

En effet l'opérateur $meta permet de récupérer le textScore qui *« Returns the score associated with the corresponding $text query for each matching document. The text score signifies how well the document matched the search term or terms »*

L'utilisation de ces méta données rend cette recherche texte unique à mongodb et rend la recherche dans notre catalogue aisée et pratique en proposant le meilleur résultat en premier

Le résultat de cette requête sera une collection de produit hétéroclite avec tous leurs champs.

6.2 USE CASE 2

Quelle est le genre le plus populaire ?

Je suis le magasin et je souhaiterais savoir pour un certain type de produit quel genre est le plus populaire

6.2.1 FONCTIONNEMENT

Comment fonctionne cette requête ?

- Calcul de moyenne et compte
- Aggrégation en group
- Opération sur des arrays

Les deux prochaines requêtes profitent des Pipelines d'agrégations de Document disponible nativement dans MongoDB. Ces pipelines permettent la restructuration des documents en ensemble cohérent en effectuant des calculs. Leur utilisation lors de calcul d'indicateurs à travers une collection distribué est plus que pertinent pour leurs résiliations, leur vitesse et leur mise à profit des différents opérateurs mongodb.
Cette requête à la différence de la suivante à été uniquement conçu avec les Builders disponible dans le Java Driver. Ces derniers permettent de préparer la requête manière sécurisé et lisible. La requête suivante est une génération automatique du client MongoDb en Bson Document (voir les deux méthodes :

- prepareAggregationPipelineForMostPopularGenre
- prepareAggregationPipelineForMarginByReleaseDate

6.2.2 ÉTAPES

Les étapes de ces requêtes sont les suivantes :

- **Projection :** Sélection des champs utilse
- **Unwind :** Déconstruit un champ de tableau à partir des documents d'entrée pour générer un document pour chaque élément. (ici le champ Genre)
- **Group :** Regroupe les documents crée après le unwind par genre, fais des calculs de moyenne sur le champ de rating et crée un nouveau tableau récoltant les titres de tous les produits concernés
- **Sort :** Tri descendant par la moyenne des ratings

6.2.3 RÉSULTAT

Le résultat de cette query est stockées dans un classe Agrégation spécialement crée pour ce résultat : MostPopularGenre. Le codec désérialise donc sans problème ce résultat.

Cette requête profite au maximum des opérateurs sur les attributs array $unwind / $push pour recréer une nouvelle collection d'Object pertinente pour nos analyses.

6.2.4 JUSTIFICATION

La pertinence de cette requête se justifie par :

- Les attributs de type array n'existant pas en relationnel
- L'exploitation des pipelines d'agrégation

- La désérialisation facilité dans un pojo customisé pour l'application

6.3 USE CASE 3

Quelle décennie possède les produits les plus rentables ?

Je suis le magasin et je souhaiterais savoir quels sont les produits avec la meilleure marge en fonctions leurs date de sortie.

6.3.1 FONCTIONNEMENT

Comment fonctionne cette requête ?

- Calcul de moyenne et compte
- Gestion conditionnel basé sur le type de champs
- Agrégation en bucket
- Opération sur des arrays

6.3.2 ÉTAPES

Les étapes de ces requêtes sont les suivantes :

- **Projection :** Sélection des champs utile. C'est à cette étape du pipeline que le type du champ release date est testé. En effet, l'objectif est de groupé les produits par décennies de sortie. Or certain produit possède une date complète et nous devons pré-traité cette dernière pour s'assurer de n'avoir dans ce champ que des années de sortie (int)
- **Bucket :** Opérateur permettant de définir les bornes des différents bucket où les documents vont être groupés. Dans le paramètre output de cet opérateur, un membre array est crée est créé avec l'opérateur push afin de sauvegarder dans chaque document par décade la liste des produits concernées

6.3.3 RÉSUTLAT

Le résultat de cette query est stockées dans un classe Agrégation spécialement créé pour ce résultat : MarginPerPeriod. Le codec déserialize donc sans problème ce résultat.

6.3.4 JUSTIFICATION

La pertinence de cette requête se justifie par :

- La gestion conditionnelle du traitement de champs en fonction de leurs types
- Les attributs de type array n'existant pas en relationnel
- L'exploitation des pipelines d'agrégation et d'opérateur de groupement comme Bucket
- La désérialisation facilité dans un pojo customisé pour l'application

7 CONCLUSION ET RETOUR D'EXPÉRIENCE

En conclusion de ce travail, nous pouvons affirmer que nous avons tenter d'exploiter les capacités d'une base de données MongoDB au maximum de leurs capacités dans notre environnement de tests.

Les défis ont été nombreux. Entre la sérialization/désérialization des objets et la recherches des opérateurs pertinent pour nos requêtes et la découverte du langages de requêtes, nous avons mis du temps à nous approprié l'ensemble des outils mais la documentation complète et la communauté très active nous ont facilité le travail.

Nous aurions pu aller plus loin en ajoutant une collection d'utilisateurs et en embarquant dans les produits un array d'object Commande (quantité, Users...) ou un array d'object Commentaire (titre, commentaires, tags) afin de créer de nouvelle requête encore plus « nested ». Il nous semblait que les requêtes étaient cependant adaptées à notre cas d'usage et que leur implémentation justifiait notre choix de la base de données Mongodb.

Il également évident que dans un cas réelle la collection rassemblerait bien plus de 17000 références et exploiterais également le système de sharding (distritbution) de MongoDB. De plus pour le prototype de recherche de texte que nous avons mis en place serait renforcer avec l'utilisation de véritable moteur de recherche.

8 BIBLIOGRAPHIE

https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows

https://www.kaggle.com/datasets/jealousleopard/goodreadsbooks

https://www.kaggle.com/datasets/atharvaingle/video-games-dataset

https://www.kaggle.com/datasets/notgibs/500-greatest-albums-of-all-time-rolling-stone

https://mongodb.github.io/mongo-java-driver/4.0/driver/getting-started/quick-start-pojo/

https://www.youtube.com/watch?v=7y2qWbBkuz0

https://www.mongodb.com/industries/retail

https://www.mongodb.com/features/database-sharding-explained

https://openclassrooms.com/fr/courses/4462426-maitrisez-les-bases-de-donnees-nosql/4462471-maitrisez-le-theoreme-de-cap

https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/crud/read-operations/sort/#std-label-sorts-crud-text-search

https://neo4j.com/developer-blog/exploring-practical-recommendation-systems-in-neo4j/

https://martinfowler.com/eaaCatalog/concreteTableInheritance.html#:~:text=Represents%20an%20inheritance%20hierarchy%20of,concrete%20class%20in%20the%20hierarchy.&text=As%20any%20object%20purist%20will,that%20complicates%20object%2Drelational%20mapping.

https://www.martinfowler.com/eaaCatalog/singleTableInheritance.html

https://www.martinfowler.com/eaaCatalog/classTableInheritance.html

https://www.martinfowler.com/eaaCatalog/classTableInheritance.html