Bases de Données Avancées, Rapport Phase 2 : Migration MongoDB

Guilhem ALBALADEJO
Info 4A
Polytech Marseille

TABLE DES MATIÈRES

I Introduction 1

II Migration des données 1

III Traduction des requêtes 1
III-A Indexes . . . 1
III-B Pipelines des requêtes . . . 1
III-B1 Filmographie d'un acteur . . . 1
III-B2 Top N films . . . 2
III-B3 Acteurs multi-rôles . . . 3
III-B4 Collaborations . . . 3
III-B5 Genres populaires . . . 4
III-B6 Évolution de carrière . . . 4
III-B7 Classement par genre . . . 5
III-B8 Carrière propulsée . . . 5
III-B9 Enfants stars . . . 6
III-C Comparaison avec SQLite . . . 6

IV Documents structurées 7

V Conclusion 7

Bases de Données Avancées, Rapport Phase 2 : Migration MongoDB

I. Introduction

Ce rapport correspond au compte-rendu de la deuxième phase du cours de bases de données avancées. Nous verrons au sein de celui-ci comment créer une base de données mongoDB, gérer la migration de nos données depuis SQLite vers des collections plates dans une premier temps, puis vers des documents structurés. Nous verrons comment reconstruire nos requêtes sous mongoDB à l'aide de pipelines d'agrégation. Enfin nous verrons les avantages et inconvénients des bases de données noSQL comparées aux bases SQL.

II. Migration des données

On cherche dans cette partie à migrer les données contenues dans notre base de données `imdb.db` créée dans la phase 1, vers une base MongoDB. On commence par installer la dernière version de MongoDB Community Edition depuis le site officiel, ainsi que la bibliothèque pymongo pour pouvoir interagir avec notre base de données via des scripts python. On lance notre base de données à l'aide de la commande :

```
mongod --dbpath data/mongo/standalone
```

La migration s'effectue grâce à un script `migrate_flat.py`, qui convertit nos tables telles quelles en collections plates. Comme on effectue aucune transformation sur nos tables, ce script est très court et indépendant de notre schéma. La seule exception concerne la conversion des booléens, qui sont stockés comme nombres entiers dans SQLite et doivent donc être convertis pendant la migration. Heureusement, les colonnes booléennes ont été créées avec le type "BOOLEAN" dans le schéma de notre base SQL, on peut donc vérifier le type des colonnes pour correctement convertir les champs booléens dans notre script d'import.

III. Traduction des requêtes

Dans MongoDB, les requêtes s'effectuent à l'aide de pipelines d'aggrégation. Nous ne pourrons donc pas réutiliser nos requêtes SQL écrites précédemment. Il faudra toutes les adapter pour pouvoir les lancer sur nos collections.

A. Indexes

Pour cela, on crée un script queries_mongo.py qui contiendra nos requêtes sous la forme de fonctions python. Un autre point à considérer avant de commencer à écrire nos requêtes concerne les indexes, en effet MongoDB étant un système de gestion de base de donnée non relationnelle, l'exécution de requêtes relationnelles risque d'être très longue, de plus, MongoDB ne crée pas d'indexes automatiques comme SQLite sur nos clés primaires, mais sur les clés "_id", que nous n'utiliserons pas dans nos requêtes.

On crée donc les indexes suivants :

- Sur la collection persons : person_id.
- Sur la collection titles : title_id.
- Sur la collection movies : movie_id.
- Sur la collection ratings : movie_id.
- Sur la collection principals : movie_id, (person_id, profession_id, movie_id).
- Sur la collection movie_titles : movie_id, title_id.
- Sur la collection person_profession : person_id, profession_id.
- Sur la collection cast : movie_id, person_id.
- Sur la collection movie_genres : movie_id, genre_id.
- Sur la collection title_ordering : movie_id, title_id

B. Pipelines des requêtes

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

1) Filmographie d'un acteur: On lance le pipeline suivant sur la collection "persons" :

```
[
    {"$match": {"name": {
        "$regex": actor_name,
        "$options": "i"
    }}},
    {"$lookup": {
        "from": "principals",
        "localField": "person_id",
        "foreignField": "person_id",
        "as": "pr"
    }},
    {"$unwind": "$pr"},
    {"$lookup": {
        "from": "movie_titles",
        "localField": "pr.movie_id",
        "foreignField": "movie_id",
        "as": "mt"
    }},
    {"$unwind": "$mt"},
    {"$match": {
        "mt.is_primary": True
    }},
    {"$lookup": {
        "from": "movies",
```

```
        "localField": "mt.movie_id",
        "foreignField": "movie_id",
        "as": "m"
    }},
    {"$unwind": "$m"},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$lookup": {
        "from": "cast",
        "let": {"person_id":
        ↪  "$person_id",
        ↪  "movie_id":"$mt.movie_id"},
        "pipeline": [
            {"$match": {
                "$expr": {
                    "$and": [
                        {"$eq":
                        ↪  ["$person_id",
                        ↪  "$$person_id"]},
                        {"$eq": ["$movie_id",
                        ↪  "$$movie_id"]}
                    ]
                }
            }}
        ],
        "as": "ca"
    }},
    {"$lookup": {
        "from": "characters",
        "localField": "ca.character_id",
        "foreignField": "character_id",
        "as": "c"
    }},
    {"$lookup": {
        "from": "ratings",
        "localField": "mt.movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$sort": {
        "m.year": -1
    }},
    {"$project": {
        "_id": 0,
        "title": "$t.title_name",
        "year": "$m.year",
        "character_name": "$c.name",
        "average_rating": "$r.average_rating"
    }}
]
```

On commence par filtrer la collection persons avec le nom de l'acteur passé dans la variable `actor_name`, puis on joint les collections contenant les informations à récupérer. On pense bien à filtrer pour garder uniquement les titres principaux et à joindre cast à la fois sur person_id et movie_id. Finalement on trie par année décroissante et on projette uniquement les champs qui nous intéressent.

2) Top N films: On lance le pipeline suivant sur la collection "movies" :

```
[
    {"$match": {
        "year" : {"$gte": start_year, "$lte":
        ↪  end_year}
    }},
    {"$lookup": {
        "from": "movie_genres",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "mg"
    }},
    {"$unwind": "$mg"},
    {"$lookup": {
        "from": "genres",
        "let": {"genre_id": "$mg.genre_id"},
        "pipeline": [
            {"$match": {
                "$expr": {
                    "$eq" : ["$genre_id",
                    ↪  "$$genre_id"]
                }
            }},
            {"$match": {
                "genre_name":{
                    "$regex": genre,
                    "$options": "i"
                }
            }}
        ],
        "as": "g"
    }},
    {"$unwind": "$g"},
    {"$lookup": {
        "from": "ratings",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$sort": {"r.average_rating": -1}},
    {"$limit": N},
    {"$lookup": {
        "from": "movie_titles",
        "let": {"movie_id": "$movie_id"},
        "pipeline": [
            {"$match": {"$expr": {"$eq":
            ↪  ["$movie_id",
            ↪  "$movie_id"]}}},
            {"$match": {"is_primary": True}}
        ],
        "as": "mt"
    }},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$project": {
```

```
        "_id": 0,
        "movie_id": 1,
        "title": "$t.title_name",
        "year": 1,
        "average_rating":
        ↪  "$r.average_rating",
        "num_votes": "$r.num_votes"
    }}
]
```

On commence par filtrer movies pour garder uniquement les films sortis dans l'intervalle [start_year, end_year]. On joint "movie_genres" et "genres" pour garder uniquement les films du genre qui nous intéresse à l'aide d'une pipeline imbriquée dans le lookup sur "genres" pour ne joindre que sur les films du bon genre. Finalement on joint les collections contenant les informations restantes, on ne garde que les titres principaux, on trie par note moyenne décroissante et on projette les champs voulus.

3) Acteurs multi-rôles: On lance le pipeline suivant sur la collection "cast" :

```
[
    {"$group": {
        "_id": {
            "person_id": "$person_id",
            "movie_id": "$movie_id"
        },
        "charactersPlayed": {"$addToSet":
        ↪  "$character_id"}
    }},
    {"$addFields": {
        "charactersCnt": {"$size":
        ↪  "$charactersPlayed"}
    }},
    {"$match": {
        "charactersCnt": {"$gt": 1}
    }},
    {"$sort": {"charactersCnt": -1}},
    {"$lookup": {
        "from": "persons",
        "localField": "_id.person_id",
        "foreignField": "person_id",
        "as": "pe"
    }},
    {"$unwind": "$pe"},
    {"$lookup": {
        "from": "movie_titles",
        "localField": "_id.movie_id",
        "foreignField": "movie_id",
        "as": "mt"
    }},
    {"$unwind": "$mt"},
    {"$match": {"mt.is_primary": True}},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$project": {
        "_id": 0,
        "person_id": "$_id.person_id",
        "actor_name": "$pe.name",
        "movie_id": 1,
        "title": "$t.title_name",
        "charactersCnt": 1,
    }}
]
```

On regroupe la collection par films et acteurs, pour garder uniquement les films dans lequel un même acteur est associé à plusieurs personnages, on compte le nombre de personnages pour chaque groupe en les stockant dans un tableau puis en lisant la taille de se tableau. On filtre pour ne garder que les acteurs multi-rôles. Finalement on récupère le reste des informations à renvoyer.

4) Collaborations: On commence par récupérer tous les films joués par l'acteur dans un tableau avec :

```
pmPipeline = [
    {"$match": {
        "name": {
            "$regex": actor_name,
            "$options": "i"
        }
    }},
    {"$lookup": {
        "from": "principals",
        "localField": "person_id",
        "foreignField": "person_id",
        "as": "pr"
    }},
    {"$lookup": {
        "from": "professions",
        "localField": "pr.profession_id",
        "foreignField": "profession_id",
        "as": "p"
    }},
    {"$unwind": "$p"},
    {"$match": {
        "p.job_name": {
            "$regex": "actor",
            "$options": "i"
        }
    }},
    {"$project": {
        "_id": 0,
        "movie_id": "$pr.movie_id"
    }}
]
# array contenant les films joués par
↪  l'acteur
playedMovies =
↪  next(db["persons"].aggregate(pmPipeline))
["movie_id"]
```

Puis on utilise ce tableau pour filtrer toutes les personnes ayant travaillé dans ces films, avant de filtrer pour ne garder que les directeurs, en lançant la pipeline suivante sur la collection "persons" :

```
[
    {"$match": {
        "movie_id": {"$in": playedMovies}
    }},
```

```
    {"$lookup": {
        "from": "professions",
        "localField": "profession_id",
        "foreignField": "profession_id",
        "as": "p"
    }},
    {"$unwind": "$p"},
    {"$match": {
        "p.job_name": {
            "$regex": "director",
            "$options": "i"
        }
    }},
    {"$lookup": {
        "from": "persons",
        "localField": "person_id",
        "foreignField": "person_id",
        "as": "pe"
    }},
    {"$unwind": "$pe"},
    {"$group": {
        "_id": {"person_id": "$person_id"},
        "name": {"$first": "$pe.name"},
        "movies": {"$addToSet": "$movie_id"}
    }},
    {"$addFields": {"movieCnt": {"$size":
    ↪  "$movies"}}},
    {"$sort": {"movieCnt": -1}},
    {"$project": {
        "_id": 0,
        "name": 1,
        "movieCnt": 1
    }}
]
```

5) Genres populaires: On lance le pipeline suivant sur la collection "movie_genres" :

```
[
    {"$lookup": {
        "from": "genres",
        "localField": "genre_id",
        "foreignField": "genre_id",
        "as": "g"
    }},
    {"$unwind": "$g"},
    {"$lookup": {
        "from": "movies",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "m"
    }},
    {"$unwind": "$m"},
    {"$lookup": {
        "from": "ratings",
        "localField": "m.movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$group": {
        "_id": {"genre_id": "$genre_id"},
        "genre_name": {"$first":
        ↪  "$g.genre_name"},
        "avg_rating": {"$avg":
        ↪  "$r.average_rating"},
        "movies": {"$addToSet": "$movie_id"}
    }},
    {"$addFields": {
        "movieCnt": {"$size": "$movies"}
    }},
    {"$match": {
        "avg_rating": {"$gt": 7.0},
        "movieCnt": {"$gt": 50}
    }},
    {"$project": {
        "_id": 0,
        "genre_name": 1,
        "avg_rating": 1,
        "movieCnt": 1
    }}
]
```

On joint les collections concernées, on groupe par genre et on filtre par genre ayant une note moyenne > 7.0 et un nombre de films > 50. Finalement on projette les champs qui nous intéressent.

6) Évolution de carrière: On lance le pipeline suivant sur la collection "movies" :

```
[
    {"$lookup": {
        "from": "cast",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "c"
    }},
    {"$unwind": "$c"},
    {"$lookup": {
        "from": "persons",
        "localField": "c.person_id",
        "foreignField": "person_id",
        "as": "pe"
    }},
    {"$unwind": "$pe"},
    {"$match": {
        "pe.name": {
            "$regex": actor_name,
            "$options": "i"
        }
    }},
    {"$lookup": {
        "from": "ratings",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$addFields": {
        "decade": {
            "$subtract": [
                "$year",
                {"$mod": [
                    "$year",
                    10
                ]}
            ]
        }
```

```
    }},
    {"$group": {
        "_id": "$decade",
        "movie_list": {"$addToSet":
        ↪  "$movie_id"},
        "avg_rating": {"$avg":
        ↪  "$r.average_rating"}
    }},
    {"$addFields": {
        "movieCnt": {"$size": "$movie_list"}
    }},
    {"$project": {
        "_id": 0,
        "decade": "$_id",
        "movieCnt": 1,
        "avg_rating": 1
    }},
    {"$sort": {
        "decade": -1
    }}
]
```

On commence par filtrer par le nom de l'acteur après deux lookups, puis on calcule la décade de chaque film selon decade = year - (year % 10). Finalement on regroupe par décade, sur lesquelles on calcule la note moyenne de tous les films et le nombre de films.

7) Classement par genre: On lance le pipeline suivant sur la collection "movie_genres" :

```
[
    {"$lookup": {
        "from": "ratings",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$setWindowFields": {
        "partitionBy": "$genre_id",
        "sortBy": {"r.average_rating": -1},
        "output": {
            "rank": {"$rank": {}}
        }
    }},
    {"$match": {
        "rank": {"$lte": 3}
    }},
    {"$lookup": {
        "from": "genres",
        "localField": "genre_id",
        "foreignField": "genre_id",
        "as": "g"
    }},
    {"$unwind": "$g"},
    {"$lookup": {
        "from": "movie_titles",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "mt"
    }},
    {"$unwind": "$mt"},
    {"$match": {"mt.is_primary": True}},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$sort": {"g.genre_name": 1}},
    {"$project": {
        "_id": 0,
        "genre": "$g.genre_name",
        "title": "$t.title_name",
        "rank": 1
    }}
]
```

On utilise `$setWindowFields` pour partitionner par genre et trier par note moyenne décroissante, on garde les 3 notes les plus haute à l'aide d'un champ "rank" que l'on ajoute lors de la partition. Finalement on récupère et projette les champs que l'on veut renvoyer.

8) Carrière propulsée: On commence par créer deux pipelines distinctes "fh_pipeline" et "lnh_pipeline" qui récupèrent respectivement les premier "hit" et dernier "non hit" de chaque personne, puis on lance la pipeline suivante sur la collection "persons" :

```
[
    {"$lookup": {
        "from": "principals",
        "let": {"person_id": "$person_id"},
        "pipeline": fh_pipeline,
        "as": "fh"
    }},
    {"$unwind": "$fh"},
    {"$lookup": {
        "from": "principals",
        "let": {"person_id": "$person_id"},
        "pipeline": lnh_pipeline,
        "as": "lnh"
    }},
    {"$unwind": {
        "path": "$lnh",
        "preserveNullAndEmptyArrays": True
    }},
    {"$match": {
        "$or": [
            {"lnh": None},
            {"$expr": {"$gt": ["$fh.year",
            ↪  "$lnh.year"]}}
        ]
    }},
    {"$lookup": {
        "from": "movie_titles",
        "localField": "fh.movie_id",
        "foreignField": "movie_id",
        "as": "mt"
    }},
    {"$unwind": "$mt"},
    {"$match": {"mt.is_primary": True}},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
```

```
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$project": {
        "_id": 0,
        "name": 1,
        "first_hit_title": "$t.title_name",
        "first_hit_year": "$fh.year"
    }}
]
```

On utilise les deux pipelines définies précédemment sur principals lors de l'agrégation pour connaître l'année du first et last non hit pour chaque personne. On filtre ensuite pour garder les personnes n'ayant eu aucun non hit après leur premier hit. Finalement on récupère les champs que l'on veut renvoyer.

9) Enfants stars: On lance le pipeline suivant sur la collection "cast" :

```
[
    {"$lookup": {
        "from": "ratings",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "r"
    }},
    {"$unwind": "$r"},
    {"$match": {
        "r.num_votes": {"$gt": 200000}
    }},
    {"$lookup": {
        "from": "movies",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "m"
    }},
    {"$unwind": "$m"},
    {"$lookup": {
        "from": "persons",
        "localField": "person_id",
        "foreignField": "person_id",
        "as": "pe"
    }},
    {"$unwind": "$pe"},
    {"$addFields": {
        "age": {
            "$subtract": ["$m.year",
            ↪  "$pe.birth_year"]
        }
    }},
    {"$match": {
        "age": {"$lt": 18}
    }},
    {"$lookup": {
        "from": "movie_titles",
        "localField": "movie_id",
        "foreignField": "movie_id",
        "as": "mt"
    }},
    {"$unwind": "$mt"},
    {"$match": {"mt.is_primary": True}},
    {"$lookup": {
        "from": "titles",
        "localField": "mt.title_id",
        "foreignField": "title_id",
        "as": "t"
    }},
    {"$unwind": "$t"},
    {"$sort": {
        "pe.name": 1,
        "age": 1
    }},
    {"$project": {
        "_id": 0,
        "name": "$pe.name",
        "movie_title": "$t.title_name",
        "votes": "$r.num_votes",
        "age": 1
    }}
]
```

On commence par garder uniquement les films ayant plus de 200000 votes, puis on ajoute un champ contenant l'âge des acteurs au moment où ils ont joués dans un film donné, finalement on filtre pour ne garder que ceux ayant moins de 18 ans et on projette les champs que l'on veut retourner.

C. Comparaison avec SQLite

On mesure à présent le temps d'exécution de chaque requête dans un fichier test_queries.py pour comparer leurs performances avec celles que l'ont avait définies sur SQLite (on prends ici les temps que l'on avait mesuré après la création d'indexes, pour refléter la performance réelle de SQLite) :

Requête	SQLite (ms)	mongoDB (ms)	Augmentation (%)
Q1 - Filmographie	1 850	2 027	9
Q2 - Top N films	72	2 492	3 361
Q3 - Multi-rôles	931	2 012	116
Q4 - Collaborations	204	196	-4
Q5 - Genres populaires	111	16 275	14 562
Q6 - Évolution de carrière	108	17 006	15 646
Q7 - Classement par genre	260	4 686	1 702
Q8 - Carrières propulsées	1765	78 427	4 343
Q9 - Enfants star	78	9 312	11 838

TABLE I. TABLEAU DE COMPARAISON DES TEMPS D'EXÉCUTION

On voit que si l'augmentation du temps d'exécution est non négligeable pour les requêtes les plus simples, il devient extrêmement important pour les requêtes plus complexes. Certaines d'entre elles atteignent des temps d'exécutions beaucoup trop élevés pour pouvoir être utilisé dans le contexte d'une application réactive sans impacter l'expérience utilisateur.

En comparant l'écriture de nos requêtes à celle que l'on avait écrit dans la phase 1 du projet, on remarque également que nos pipelines d'agrégation sont beaucoup plus long et complexes que nos requêtes SQL. Cela s'explique en partie par la forme que prennent ces pipelines : des documents JSON, ou dicts python, mais également par l'écriture de celles-ci, très longues et complexes.

Cela s'explique par la forme que prends notre base de données : elle suit encore le schéma relationnel que l'on avait défini pour notre base de données SQLite, avec des collections très normalisées et beaucoup de tables d'associations. Si MongoDB permet de traiter ce genre de base de données,

on remarque vite que cela impacte fortement la complexité des requêtes : la majeure partie de nos pipelines est occupée par des lookups et des unwinds, qui en plus d'augmenter la complexité et réduire la lisibilité de nos requêtes, impacte fortement la performance de celles-ci, en effet MongoDB n'est pas optimisé pour cela.

IV. Documents structurées

Afin de tirer pleinement partie des capacités noSQL de mongoDB, on va créer une collection de documents structurés movies_complete contenant toutes les informations associées à un film pour pouvoir les récupérer à partir d'un seul document sans lookup.

Les documents de notre collection suivront le modèle :

```
{
    "_id": "tt0111161",
    "title": "The Shawshank Redemption",
    "year": 1994,
    "runtime": 142,
    "genres": ["Drama"],
    "rating": {
        "average": 9.3,
        "votes": 2500000
    },
    "directors": [
        {"person_id": "nm0001104", "name": "Frank Darabont"}
    ],
    "cast": [
        {
            "person_id": "nm0000209",
            "name": "Tim Robbins",
            "characters": ["Andy Dufresne"],
            "ordering": 1
        },
        {
            "person_id": "nm0000151",
            "name": "Morgan Freeman",
            "characters": ["Red"],
            "ordering": 2
        }
    ],
    "writers": [
        {"person_id": "nm0175840", "name": "Stephen King", "category": "novel"},
        {"person_id": "nm0001104", "name": "Frank Darabont", "category": "screenplay"}
    ],
    "titles": [
        {"region": "US", "title": "The Shawshank Redemption"},
        {"region": "FR", "title": "Les Évadés"}
    ]
}
```

Pour construire ces documents, on se base sur les données déjà présentes dans notre base de données, pour construire les documents structurés à l'aide d'une pipeline d'agrégation dans le fichier migrate_structured.py.

Cette pipeline, servant à récupérer un film complet est une pipeline très complexe est très longue, récupérant des informations dans de nombreuses collections, en effet si l'on voulait se passer de la collection movies_complete, et reconstruire un film à l'aide de cette pipeline, cela prends : 146 ms, alors que simplement aller chercher le même document dans movies complete prends : 0,1 ms.

De plus, en utilisant la méthode storageSize() sur les collections dans mongosh, on remarque que la taille totale de nos collections plates avec index est de : 82,69 Mo, alors que celle de notre collection structurée est de : 36,32 soit plus de deux fois moins grande.

Créer une collection structurée représente donc une manière de stocker l'information sous une forme plus condensée et de la récupérer de manière plus performante, on voit ici clairement l'avantage de construire notre base de données en suivant le modèle de document dénormalisé classique de MongoDB plutôt que d'utiliser un schéma relationnel.

V. Conclusion

Si les bases de données noSQL comme mongoDB peuvent être manipulées comme des bases relationnelles, elles pour cet usage là bien moins performantes que des bases de données relationnelles classiques. Leur avantage se trouve dans leurs capacité à gérer des données structurées autrement que sous forme de tableau rigide, et ainsi gagner en flexibilité, en espace de stockage, et en performance, à condition de stocker sous forme pré-agrégée les résultat de nos requêtes.