Term Paper Modern Database Systems

submitted by: Domenic Gonzalez (11140329)
Maxi Tix (11139102)

submitted to: Prof. Dr. Johann Schaible

Gummersbach, 28.06.2023

Table of contents

List of tables

List of figures

.

1 Introduction

The choice between a relational database and a non-relational database can be a crucial factor in the development of software applications, especially when managing data with complex relationships. In this work, we will focus on managing user feedback for films.

The user feedback data is represented by a five-star rating and is associated with each film. Different genres of films will be considered to allow for a comprehensive evaluation. The aim of this work is to develop a comprehensive application example for both database types and then compare the quality of the databases.

Through the utilization of our use case, we aim to analyze user feedback and provide film recommendations from various perspectives.

The results of the comparison can help simplify the decision-making process when choosing between relational and non-relational databases and thus improve the effectiveness and efficiency of software applications.

The data and code used to implement the use case are available in a GitHub repository via the following link: https://github.com/DomGonzalez/MDS_MovieLensData_UseCase

2 Data

The dataset pertains to the ratings and free-text tagging activities from MovieLens, which is a movie recommendation service. It comprises 20000263 ratings and 465564 tag applications across 27278 movies. These data were collected from 138493 users who used the service between January 09, 1995, and March 31, 2015. The dataset was created on October 17, 2016.

The users were selected randomly for inclusion and all selected users had rated a minimum of 20 movies. No demographic information is provided, and each user is represented by an id only.

The data are distributed across six files, namely tag.csv (contains tags applied to movies by users), rating.csv (contains movie ratings provided by users), movie.csv (contains movie information), link.csv (contains identifiers that can be linked to other sources), genome_scores.csv (contains movie-tag relevance data), and genome_tags.csv (contains tag descriptions). (cf. Kaggle Dataset: MovieLensData)

In our specific use case, we rely on the utilization of three CSV files: Movie, Rating, and Tag. These files serve as essential data sources for our analysis and operations. However, due to certain limitations during the data import process, we were compelled to work with a reduced overall size of 4530 kilobytes for these files. The Rating table is the largest, with a size of 2032 kilobytes. The Movie table contains data in the order of 1261 kilobytes, while the Tag table has a size of 1237 kilobytes.

3 Database

For the selected data set and use case we decided to use MongoDB. MongoDB is a document-based NoSQL database system that was developed and designed to be fast, scalable, and flexible. To achieve these properties, MongoDB as a NoSQL database system does not follow the ACID principle, but the BASE principle. A crucial difference between these principles can be seen in the importance of consistency. A database system based on the ACID principle must have consistency as a property. In contrast, database systems based on the BASE principle are not obliged to be consistent but can soften this consistency. MongoDB also sacrifices consistency for greater scalability, speed, and flexibility. It does not use relations like relational database systems to store data but uses documents for this purpose. More precisely, they are JSON documents (JavaScript Object Notation). These offer more flexibility for the design of the database system because they do not have a schema. Each JSON document can be different in that it has different fields or contains different types of data in one field. The JSON documents can be stored in collections. Collections are seen in MongoDB as a collection of JSON documents in which the data is stored as key-value pairs. Key-value means that a field in a document always has a description as key, e.g. "city" and one or more values, e.g. "Cologne". (cf. Sabharwal and Edward, 2015, p. 26f.)

To store the JSON documents, MongoDB uses the BSON (Binary-JSON) standard format, which binary encodes the JSON documents and allows to nest arrays and objects within other arrays and index them. (cf. Sabharwal and Edward, 2015, p. 31f.)

Another important feature in MongoDB is the indexing of documents. The indexing is intended to make queries in MongoDB faster. In MongoDB, all documents automatically have a primary index on the identifier of the document. This id and therefore the index cannot be deleted. It is also possible to create your own secondary indexes. You can create unique indexes like the primary index or non-unique indexes. However, unique indexes can also be set only on fields that have unique values. Beside the mentioned indexes there is also the possibility to create composite indexes. These indexes can be set on the combination of several fields. For example, on "first name and last name" if you want to quickly search for the full name. (cf. Hows et al., 2015, p. 11f.)

MongoDB has several areas of application. Two of these areas are Analytics and Internet of Things. In Analytics, MongoDB's ability to store large amounts of structured and unstructured data enables comprehensive and versatile data analysis. MongoDB's flexible data storage allows users to store and analyse a wide variety of data, which is especially useful when data is heterogeneous and not strictly structured. In addition, MongoDB supports complex data processing and analysis tasks through features such as aggregation pipelines. However, it is important to note that MongoDB is not optimal for all types of analytics requirements. For applications that require complex transactions to be executed atomically, other database systems may be better suited. (cf. MongoDB use case: Analytics)

On the Internet of Things application area, MongoDB's ability to store and process large amounts of data makes it an appropriate choice for IoT applications. The data generated by IoT devices is often heterogeneous and unstructured, and this is where MongoDB's flexible data structure offers a significant advantage. MongoDB's horizontal scalability also enables efficient processing of large volumes of data. As with analytics, however, MongoDB is not optimal for all IoT requirements. For example, it may be less suitable for applications that require extremely low latency. (cf. MongoDB use case: Internet of Things)

4 Data Modelling

4.1 Relational Database

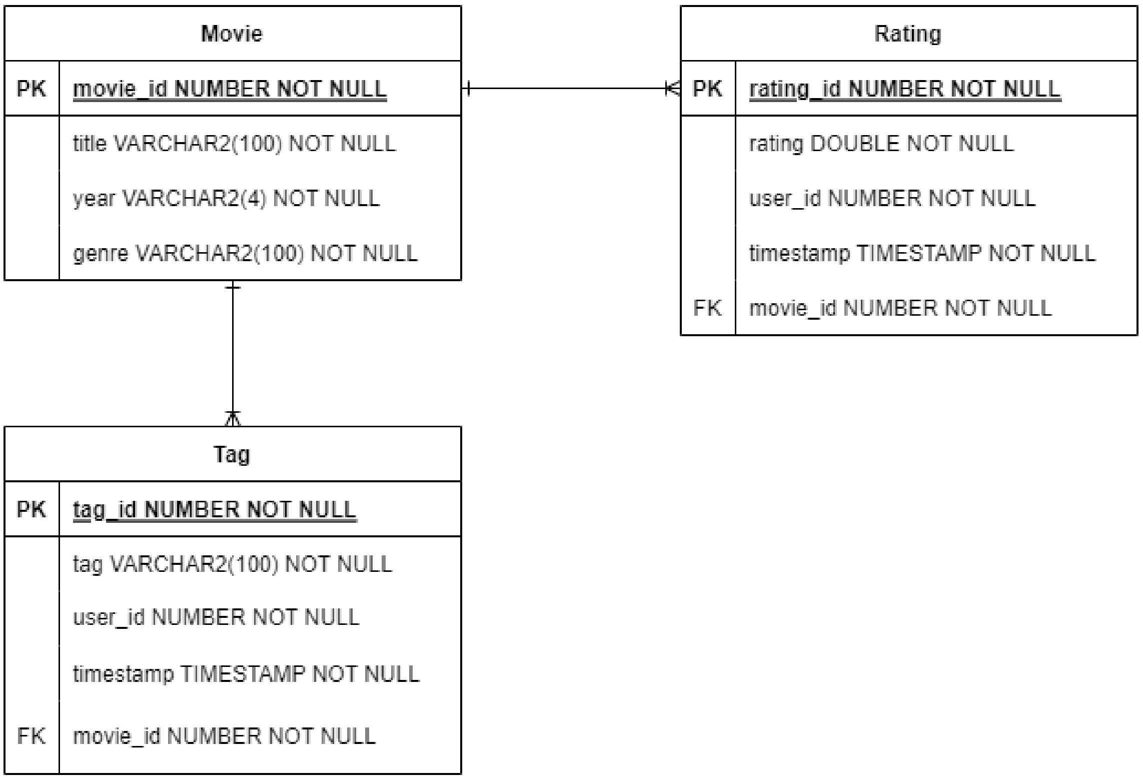

Movie
PK
movie_id NUMBER NOT NULL
title VARCHAR2(100) NOT NULL
year VARCHAR2(4) NOT NULL
genre VARCHAR2(100) NOT NULL
Rating
PK
rating_id NUMBER NOT NULL
rating DOUBLE NOT NULL
user_id NUMBER NOT NULL
timestamp TIMESTAMP NOT NULL
FK
movie_id NUMBER NOT NULL
Tag
PK
tag_id NUMBER NOT NULL
tag VARCHAR2(100) NOT NULL
user_id NUMBER NOT NULL
timestamp TIMESTAMP NOT NULL
FK
movie_id NUMBER NOT NULL

Figure 1: Entity Relationship Model

The given ER model consists of three tables: Movie, Rating, and Tag. The Movie table represents movies and contains the primary key movie_id, as well as the attributes Title, Year, and Genre. Each movie is identified by a unique movie_id, while Title stores the movie's name, Year denotes the release year, and Genre specifies the movie's genre.

The Rating table is used to store movie ratings. It includes the primary key rating_id, along with attributes such as rating, user_id, timestamp, and the movie_id as a foreign key. Each rating is identified by a unique rating_id and includes the actual rating value denoted by the rating attribute. The user_id indicates the ID of the user who submitted the rating, and timestamp records the time of the rating. The foreign key movie_id establishes a relationship to a specific movie, associating the rating with that particular movie.

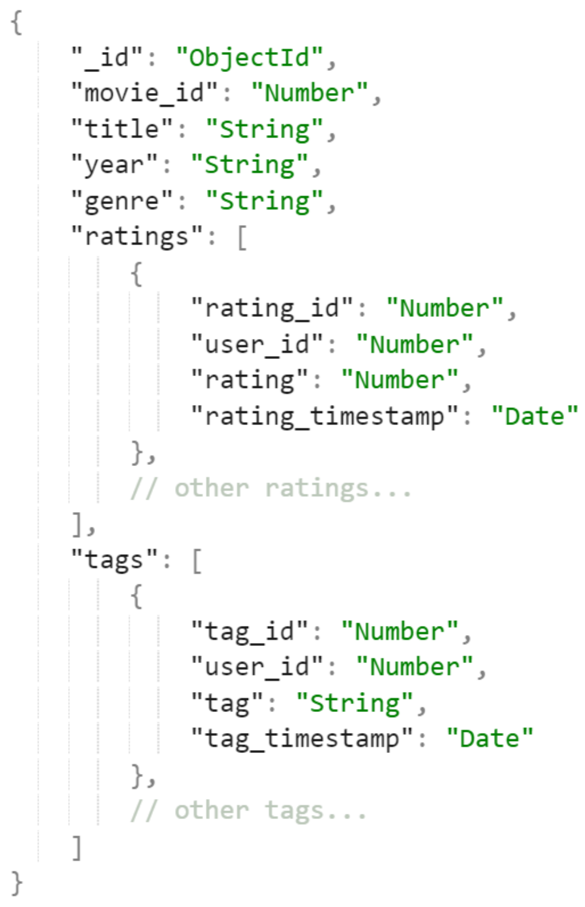
The Tag table contains keywords or tags that can be associated with movies. It comprises the primary key tag_id and attributes such as tag, user_id, and timestamp. Each tag is assigned a unique tag_id, while the tag attribute stores the actual tag text. The user_id identifies the user who added the tag, and timestamp records the time of addition. The Tag table also includes a foreign key relationship to a specific movie through the movie_id, allowing the tag to be associated with a particular film.

The ER model demonstrates a one-to-many (1:n) relationship between the Movie table and the Rating and Tag tables. This indicates that a movie can have multiple ratings and tags, while each rating and tag belongs to only one movie.

By utilizing primary keys (movie_id, rating_id, tag_id), the model ensures the uniqueness of records within each table. The foreign key relationships (movie_id in the Rating and Tag tables) facilitate the association of ratings and tags with their respective movies.

Overall, this ER model enables efficient management of movies, their ratings, and associated tags, as well as the establishment of relationships between them.

4.2 NoSQL MongoDB



```
{
    "_id": "ObjectId",
    "movie_id": "Number",
    "title": "String",
    "year": "String",
    "genre": "String",
    "ratings": [
        {
            "rating_id": "Number",
            "user_id": "Number",
            "rating": "Number",
            "rating_timestamp": "Date"
        },
        // other ratings...
    ],
    "tags": [
        {
            "tag_id": "Number",
            "user_id": "Number",
            "tag": "String",
            "tag_timestamp": "Date"
        },
        // other tags...
    ]
}
```

Figure 2: JSON Document Structure

The data modelling in MongoDB was done considering the requirements of the application and the type of queries that will be executed. In this use case, the MOVIE, RATING, and TAG tables from the relational database were consolidated into a single document in MongoDB. This was done to take advantage of the embedded documents in MongoDB, which provide an efficient and natural way to group related data together. The ratings and tags fields are arrays of embedded documents, each representing a rating or tag. This allows multiple ratings and tags to be linked directly to a movie without the need for separate tables or collections. This is especially useful for the queries that require information about a movie and its ratings or tags at the same time, as they can be executed in a single step without requiring JOINs or similar operations. In the use case under consideration, we have several such queries where we need both the data about a movie, its ratings, and its tags.

The selection of indexes in MongoDB also has been carefully tuned to the specific queries that will be executed in the application. Indexes allow MongoDB to directly access the relevant documents instead of having to search each document in the collection.

An index has been created for the movie_id because this key is used as a search or filter criterion in many queries. For example, in the movie recommendation query and in the query to select movies of a particular genre, movie_id is used to identify the relevant movies. An index on movie_id can significantly improve the performance of these queries by allowing MongoDB to directly access the documents with the corresponding movie_id.

Similarly, the indexes on ratings.rating_id and tags.tag_id were chosen because these keys are used in the update and select queries. For example, in the update query, rating_id is used to identify the specific rating to be updated. An index on rating_id can improve the performance of this query by allowing MongoDB to directly access the document with the corresponding rating_id.

Another index was created for 'year' because this field is used in the aggregation query that selects the best movie for each year. In this query, 'year' is used to group the movies and calculate the average rating for each year. An index on 'year' can improve the performance of this query by allowing MongoDB to efficiently sort and group the documents by ‘year’.

Finally, an index on 'genre' was created because this field is used in the queries for movie recommendation and for selecting movies of a certain genre. In these queries, 'genre' is used to filter the movies that are recommended to users or that are selected. An index on 'genre' can improve the performance of these queries by allowing MongoDB to directly access the documents with the corresponding genre. However, it is important to note that there are drawbacks to creating indexes in general. They can consume additional storage space and affect the performance of write operations. Therefore, we have selected only a limited number of indexes that make sense for the use case and the associated queries.

5 Queries

Table 1 shows selected queries of this work. The present use case mainly requires read operations, as retrieving information is an essential component in creating rankings. In contrast, write and delete operations are rarely required, as adding a new movie or rating occurs less frequently than creating a rating.

Query	CREATE	READ	UPDATE	DELETE
The most popular films based on the rating		X		
Most popular films of a genre		X		
Best rated films of a user		X		
Add a rating	X			
Add a film	X			
Delete a rating				X
Update a rating			X	
Delete a film				X
Film recommen-dation based on genre		X		
All films rated bet-ter than 4		X		
All films rated bet-ter than 4 for a genre		X		
Most awarded tags		X		
Film with the most tags		X		
Film with the most tags in the different years		X		

Table 1: CRUD Matrix

6 Performance and interpretation

Query	Oracle (in ms)	Oracle with In-memory (in ms)	MongoDB (in ms)
The most popular movies based on the rating	1315	1186	137
Most popular movies of a genre	96	11	2
Best rated movies of a user	139	9	91
Add a rating	1	1	11
Add a movie	1	1	5
Delete a rating	1	4	20
Update a rating	2	4	45
Delete a movie	1	3	12
Movie recommendations based on genre	142	9	40
All movies rated better than 4	604	11	127
All movies rated better than 4 for a genre	146	11	10
Most awarded tags	74	7	82
Movie with the most tags	722	596	73
Movie with the most tags in the different years	142	9	59

Table 2: Performance of the different database types

The results demonstrate that read operations in MongoDB are significantly faster compared to the relational database. This can be attributed to the utilization of indexes and the data organization structure of nested tables. On the other hand, write, update, and delete operations in the non-relational database are more than 20 times slower. However, considering the specific use case, the focus of performance lies in reading the data, as reflected in the ratio of read operations compared to write, update, and delete operations.

Operations in Oracle In-Memory are significantly faster than those in MongoDB. However, the use of In-Memory databases for large data sets is not advisable due to limited direct access to memory, making these operations costly for businesses.

In particular, the query "Most popular movies based on rating" clearly demonstrates that the operation in the non-relational database is much faster. This is because the implementation in the relational database involves two aggregations, two sorts, and a join command. In contrast, the non-relational database does not require a join since the corresponding ratings for each movie are already included in each MongoDB document. Due to the large amount of data, the join operation in Oracle, including Oracle with In-Memory implementation, takes considerably more time. This indicates that join operations are inefficient and costly, highlighting an advantage of non-document-based non-relational databases.

The query "Most popular movies of a genre" serves as a vivid example of the efficient functioning of indexes in MongoDB. In this case, the index defined on the "Genre" field is utilized, accelerating the search query by 145 ms. Without the index, the performance duration is 147 ms, slower than queries in relational databases. This is because the data volume involved in the join operation in the relational database is reduced through filtering based on a genre, leading to more efficient processing. However, it should be noted that as the data volume increases, the operation in MongoDB remains significantly more efficient and cost-effective than an Oracle implementation, based on the specific use case.

Additionally, the query "Movie with the most tags" stands out for its performance. Once again, this can be attributed to the structure of nested tables. In a relational database, the tags need to be filtered based on the movie ID in the tag table, and the number of entries needs to be counted. In MongoDB, however, only a search based on the movie ID in the table is required. The count of tags can be directly obtained using the length of the array that represents the tags. This illustrates that the structure of nested tables offers better performance compared to filtered queries in Oracle.

7 Discussion

Although the majority of queries are faster in in-memory databases, the overall performance of the document-oriented implementation is faster. Often, the in-memory implementation is only a few milliseconds faster, especially for read operations, while implementations in MongoDB can save thousands of milliseconds, especially for aggregations with join commands. In addition, in our use case we mainly focus on the read operations and here the time difference is often only a few milliseconds.

In our use case, both the implementations in MongoDB and the implementation with Oracle In-Memory are characterized by their particularly good performance. Although the performance of some queries in MongoDB can be somewhat worse, the use of a document-oriented, non-relational database is recommended. In-memory databases are characterized by direct access to local memory, which makes them very fast. However, this is only useful for smaller amounts of data, as local memory is limited and very expensive. In our use case, where the number of movies, ratings and tags is continuously increasing, the use of in-memory databases becomes very expensive. Based on these factors mentioned above, we recommend using MongoDB as it is more cost effective for enterprises and offers good performance even with growing data volumes.

8 Implementation of NoSQL Database concepts

Over the long term there may be a continuous increase of movies, ratings, and tags in the database. Therefore, a strategy for horizontal scaling should be developed. MongoDB offers a range of options for this, including consistency models, sharding, replication, and various types of secondary indexes for query processing.

In terms of consistency, which refers to how up-to-date and synchronized a data view is across all nodes in distributed systems, a form of eventual consistency could be chosen. This model allows for temporary inconsistencies between nodes, which eventually resolve without any further updates. This is particularly suitable for the use case, as it is not of the highest priority for users to always see the latest ratings and tags. A possible choice could be ‘Read-your-writes’ consistency, as this ensures that users receive immediate feedback about their own ratings and tags, while performance is improved by avoiding the need to wait for the synchronization of all nodes.

Sharding is a method of dividing and storing data sets across multiple servers. These servers are also called shards. One way to apply sharding in MongoDB is via range-based sharding. In range-based sharding, data is distributed among shards based on range values of the sharding key. Each shard contains data for a specific range of key values. (cf. Sabharwal and Edward, 2015, p. 124ff.)

In the use case, sharding based on "movie_id" can distribute data evenly, since each movie has its own ratings and tags. In addition, there are queries that require movies in a specific range that rarely changes, such as "Most popular movies based on the rating". Therefore, range-based sharding is probably the best type of sharding for this use case.

In addition to sharding, replication could be implemented using replica sets, which are a group of MongoDB servers that hold the same data. A replica set consists of a primary node that processes all write operations, and several secondary nodes that replicate data from the primary node. The replica set is a newer type of traditional master-slave replication. A main feature of the replica sets is that the master node is not fixed and there is an automatic failover system so that if the master node fails, a slave node becomes the master node. So, this replication strategy provides both data availability and durability. (cf. Sabharwal and Edward, 2015, p. 96f.)

Finally, both local and global secondary indexes could be useful for the query processing process. Local secondary indexes could be more efficient for queries that relate to a specific movie, as they are limited to data on a single shard. On the other hand, global secondary indexes, which span all shards and are not limited to data on a single shard, could provide better performance for queries that relate to multiple movies. This is because a query that uses a global secondary index can efficiently search data across multiple shards. The queries in this use case are mostly related to multiple movies, e.g. "Most popular movies based on the rating" or "Most popular movies of a genre", so using global secondary indexes would be better for query processing in the long run.

9 Conclusion

In this work, we were able to gain comprehensive insights into the different concepts of relational and NoSQL databases. The divergences in the practical implementation of a specific use case in both database systems have enabled a deeper understanding of these concepts.

One major difference is in the data modeling for the use case. While Oracle, as a relational database, stores the data in separate tables and links them via foreign key relationships, a document-oriented database like MongoDB offers different approaches. These include referencing related documents in different collections or embedding related objects in a document.

In the context of the use case, numerous CRUD operations in both database systems were compared. However, the most relevant operations for the use case are read operations. Therefore, an attempt was made to model the data in both databases in such a way that the performance of the read operations is optimized. In MongoDB, this led us to choose to embed the tag and rating objects in their respective movie document. In the Oracle relational database, there is not such flexible data modeling, but the performance of the queries could be improved by using Oracle's built-in in-memory functionality by loading the tables into memory.

Comparison of query performance in both databases showed that, as expected, MongoDB has better performance, especially for read operations. In Oracle, on the other hand, store, adjust and delete operations are faster than in MongoDB. However, by storing the data in memory, the performance of read operations in Oracle could be significantly improved.

We consider MongoDB more suitable for our use case, especially in a long-term view where the amount of data is steadily increasing. MongoDB is the more cost-efficient alternative, especially compared to in-memory in Oracle, and offers the possibility of horizontal scaling. Various concepts such as sharding and replication can be used to ensure efficient performance even with large data volumes.

In conclusion, the choice of database strongly depends on the specific use case and the associated requirements. In our case, MongoDB won us over due to its flexibility in data modeling, its ability to scale horizontally, and its superior performance in read operations. These findings underscore the importance of careful and thoughtful database system selection to maximize application efficiency and performance.

Going further, it might also be interesting for to compare other NoSQL database systems that are also suitable for this use case with the MongoDB implementation in the aspects of data modeling, query performance and concepts for horizontal scaling.

References

Hows, D., Membrey, P., Plugge, E. and Hawkins, T. (2015) *The definite way to MongoDB: A complete guide to dealing with Big Data using MongoDB*, New York, NY, Apress Springer Science+Business Media.

Kaggle Dataset: MovieLensData [Online] (Accessed 27 June 2023).

MongoDB use case: Analytics [Online]. Available at https://www.mongodb.com/use-cases/analytics (Accessed 20 June 2023).

MongoDB use case: Internet of Things [Online]. Available at https://www.mongodb.com /use-cases/internet-of-things (Accessed 20 June 2023).

Sabharwal, N. and Edward, S. G. (2015) *Practical MongoDB: Architecting, developing, and administering MongoDB*, New York, NY, Apress Springer.