Advanced Databases/Databases Technologies

2022/2023

Group 2 - Alexandre Sobreira (59451) André Dias (59452) Miguel Catarro (52740) Tiago Rodrigues (49593)

1 - Introduction

The primary purpose of this project was to compare relational databases and NoSQL databases regarding data modelling, querying, and optimizations. This project was performed in python using different libraries for each type of database: for relational databases (SQLite), the *sqlite3* library was used; and for the non-relational databases (MongoDB), the *pymongo* library was used. For this project, a dataset from the website Kaggle was chosen.

The "World Happiness Report" dataset was chosen, with happiness scores from 2015 to 2019 for several countries according to their economic production, social support, etc. Each Comma-Separated-Value (CSV) file contains the happiness scores for the countries in a specific year (hence a total of 5 CSV files exist), their respective ranking, the countries, and all the variables that were used to calculate the happiness score.

During the analysis of the CSV files, it was noted that some variables had different names in different files. The approach to resolving this issue was standardizing all the columns' names so that a given variable had the same name across all files. Moreover, some CSV's had variables that others did not. In the 2016 CSV, there were columns with information regarding "Lower Confidence Interval" and "Upper Confidence Interval" and in the 2017 CSV, there were columns with "Whiskers High" and "Whiskers Low". Since these columns were not seen in the other files, these were discarded, except for the column "Region".

After the mentioned alterations, the CSV's had the following columns: Country; Region (only on the 2015 and 2016 CSV files); Happiness Rank; Happiness Score; Economy; Social Support; Life Expectancy; Freedom; Absence of Corruption; Generosity.

Having standardized all the CSV files, the data was read in python using pandas, and data frames were created for each file. Every dataframe was examined for null and duplicated values. Before moving forward, the database schema/data model had to be considered to create new data frames with the information required for each table/collection.

2 - Schema/Data model

For SQLite, the schema presented in Figure 1 was chosen. A table named "Country" was created with all the countries ("Country" – Primary-key) and their regions ("Region") present in the data. The table "Happiness_Rank" had the happiness ranks of each country in each year ("Happiness_Rank_2015" to 2019) and two more columns. The first had all the countries ("Country" – Foreign-key), and the second ("Global_Rank" – Primary-key) was created by calculating the averages of the "Happiness_Score" of all the years and attributing a rank to each country based on this mean. The table "Happiness_Mean_Stats" has the average of all the variables used to calculate the happiness score and the column ("Global_Rank" – Foreign-key).

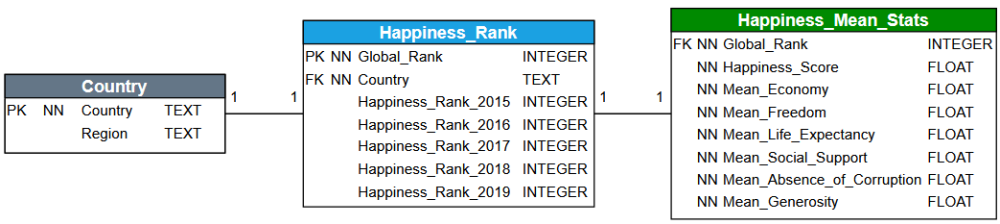

Figure 1 *– SQLite database schema. A total of three tables were created ("Country"; "Happiness_Rank"; "Happiness Mean Stats"). PK – Primary key, FK – Foreign key, NN – Not null.*

The tables "Country" and "Happiness_Rank" were connected through the "Country" column, and the tables "Happiness_Rank" and "Happiness_Mean_Stats" through the "Global_Rank" column. These relations enable the use of joins in queries. Since not all countries had "Region" nor "Happiness_Rank" at given years, null values were allowed in these columns.

For MongoDB, the data model seen in Figure 2 was created. This data model is very similar to the SQLite schema. The only difference is that the database automatically gives the "_id" and states no relations between collections.

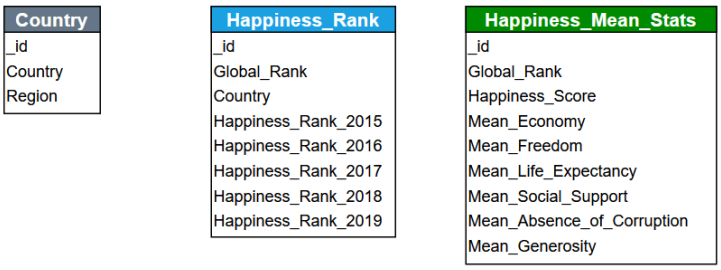

Figure 2 *– MongoDB database data model. A total of three collections were created ("Country"; "Happiness_Rank"; "Happiness_Mean_Stats").*

Since the data only had the variables for each year, the values of the "Happiness_Mean_Stats" table needed to be pre-processed. The pre-processing method consisted in creating a data frame for each column of the "Happiness_Mean_Stats" table and calculating the average of all the years for each country. This average would then be used to represent the values of each variable in the "Happiness_Mean_Stats" table. The "Global_Rank" column was calculated by ordering the averages of the happiness score and ranking the countries according to it. Figure 3 shows part of the data frame created to calculate the "Happiness_Score" and "Global_Rank":

	Country	Happiness Score 2015	Happiness Score 2016	Happiness Score 2017	Happiness Score 2018	Happiness Score 2019	Mean	Global Rank
0	Denmark	7.527	7.526	7.522	7.555	7.600	7.54600	1
1	Norway	7.522	7.498	7.537	7.594	7.554	7.54100	2
2	Finland	7.406	7.413	7.469	7.632	7.769	7.53780	3
3	Switzerland	7.587	7.509	7.494	7.487	7.480	7.51140	4
4	Iceland	7.561	7.501	7.504	7.495	7.494	7.51100	5
...	...	...	...	...	...	...	...	...
165	Rwanda	3.465	3.515	3.471	3.408	3.334	3.43860	166
166	South Sudan	NaN	3.832	3.591	3.254	2.853	3.38250	167
167	Syria	3.006	3.069	3.462	3.462	3.462	3.29220	168
168	Central African Republic	3.678	NaN	2.693	3.083	3.083	3.13425	169
169	Burundi	2.905	2.905	2.905	2.905	3.775	3.07900	170

Figure 3 *– Dataframe with the happiness score values of each country for each year, the average of the scores and creation of the "Global Rank" column based on the order of the averages.*

3 - Queries

To test the performance of the databases, several queries with different levels of complexity were performed: two simple; two complex; one update; and one insert query.

The first simple query was designed to respond to the question, "which are the five happiest countries for the five years?" while the second focused on answering which are the five least happy countries. In SQLite, the queries were performed by selecting the column "Country" from the table "Happiness_Rank," where the column "Global_Rank" was smaller than 5 and superior to 166 for the first and second queries, respectively. In MongoDB, a find was performed in the "Happiness_Rank"

collection regarding the "Global_Rank" key, with a comparison operator "lower than" 6 and "greater than" 166 for the first and the second queries correspondingly.

The first complex query was meant to answer the following question: "which countries have a happiness score above 7.5, and what are their respective regions?". In SQLite, the "Country" and "Region" columns were selected from the table "Country" followed by two "LEFT JOIN" operations. For the first, the "Country" table was joined with the "Happiness_Rank" table by the "Country" column, and for the second, the "Happiness_Rank" table was joined with "Happiness_Mean_Stats" table by the "Global_Rank" column. A "WHERE" clause was used to filter the results with "Happiness_Score" above 7.5. In MongoDB, instead of "LEFT JOIN" operations, the aggregation operation "$lookup" was used twice, with a "$unwind" in between to deconstruct the array field formed from the "$lookup". In the end, a "$match" was used to filter the data according to the "Happiness_Score", and a "$project" was applied to specify which variables appeared in the output.

The second complex query was meant to answer, "how many countries does each region have, and what are the average happiness ranks and absence of corruption score?". In SQLite, the "Region" column, its respective frequency ("COUNT"), and "AVG" of both "Global_Rank" and "Mean_Absence_of_Corruption" were firstly selected. Two "LEFT JOIN" operations followed this, the first joining the "Country" table with the "Happiness_Rank" table by the "Country" column and the second joining the "Happiness_Rank" table with the "Happiness_Mean_Stats" table by the "Global_Rank" column. In the end, a "GROUP BY" statement and "ORDER BY" command were used to obtain the results grouped by region and ordered by the mean of absence of corruption in descending order. In MongoDB, the "LEFT JOIN" operations were replaced with "$lookup", "GROUP BY" with "$group" and "ORDER BY" with "$sort". It was necessary to use "$replaceRoot" in between lookups to merge the resulting array from the "$lookup" with the root of the query result.

A new country and respective values were inserted in each table for the insert query. In SQLite, three "INSERT" queries were used, while for MongoDB, three "Insert_one" queries were used (one for each table/collection).

For the update query, the countries with a missing region were updated by giving them regions according to their location. For SQLite and MongoDB, a for loop was created to "UPDATE"/"update_one" each country with the corresponding region.

4. Indexing and Optimization

4.1 - Query Optimization

One of the ways to achieve better query performance in the databases is to optimize the queries. Through the analysis of all the queries, only the first complex query was deemed to be capable of optimization. As such, it was optimized for both databases. The optimization went as follows: instead of firstly querying the table/collection "Country", the "Happiness_Score" column in the table/collection "Happiness_Mean_Stats" was selected first. This allowed processing the information with the condition ("Happiness_Score"> 7.5) in the beginning, minimizing the amount of data to be aggregated in the subsequent "LEFT JOIN" / "$lookup" since it will just join the data that was compliant with this condition.

4.2 - Indexes

Indexing often allows databases to speed up queries by giving an index to rows of specific columns through which a query search is done immediately, avoiding having to search row by row for the results. As such, indexes should be carefully considered according to the database schema/data model and the queries to be performed.

In the beforementioned queries, only the two complex queries require information from more than one table/collection. As such, indexes were created for relevant columns of each table/collection required for these queries. For SQLite, two unique indexes were created: one for the "Country"

column on the table "Happiness_Rank", and another for the column "Global_Rank" on the table "Happiness_Mean_Stats". For MongoDB, two hashed indexes were created: one for the "Country" key in the collection "Happiness_Rank", and another for the "Global_Rank" key in the collection "Happiness_Mean_Stats". The utilization of the indexes by the queries was confirmed using DB Browser (“EXPLAIN QUERY PLAN”) or Mongoshell (“.explain(executionStats)”). This information is present as images in the provided script of the project.

4.3 – Changes to the relational schema of the SQLite database

Another possibility for making queries faster comes from changing the database schema. By analysing the most common and relevant queries, it is possible to make changes to the schema of the database to allow the grouping of the most pertinent information in a single table to make its querying faster.

Considering the dataset used, the most relevant information was "Country", "Region", "Global_Rank" and "Happiness_Score". As such, the "Country" table was modified to contain the four mentioned columns (Figure 4). The primary key "Global_Rank" of the "Happiness_Rank" table was deleted. The foreign key of the "Happiness_Mean_Stats" table was changed from "Global_Rank" to "Country". With these changes in the relational database schema, most queries also had to be modified to provide the information initially desired. The queries were altered as follows:

The only difference for both simple queries was that they were applied to the "Country" table instead of the "Happiness_Rank" table.

For the complex queries, both had to be adapted. The first complex query initially required two "LEFT JOIN" operations. However, after changing the database schema, this query could be performed using only a simple select in the "Country" table since the happiness score was already incorporated in the table. The second complex query also required two "LEFT JOIN" operations initially. However, after schema changes, only a "LEFT JOIN" operation was needed from the "Country" table to the "Happiness_Mean_Stats" table to obtain the average of the absence of corruption.

For the insert/update queries, no meaningful changes were made. However, the insert query required slight modifications to ensure that all the information needed by the new tables was given.

The columns each index had to be applied to also changed. For the modified schema, the indexes were applied to the "Country" column from the "Happiness_Rank" and "Happiness_Mean_Stats" tables.

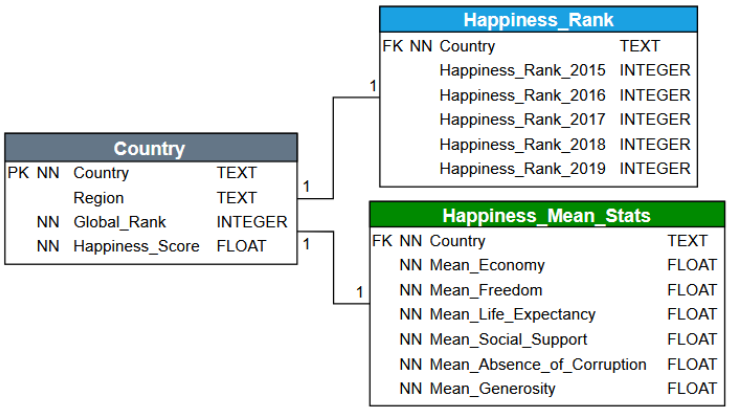

Figure 4 *– Modified SQLite database schema. A total of three tables were created (“Country”; “Happiness_Rank”; “Happiness_Mean_Stats”). PK – Primary key, FK – Foreign key, NN – Not null.*

4.4 – Changes to the data model of the MongoDB database

Similarly to the relational database schema, analogous changes were made to the MongoDB database data model to make queries faster.

Considering the same relevant information stated before, the "Country" collection was modified to contain the four mentioned keys ("Country"; "Region"; "Global_Rank"; "Happiness_Score"). In the "Happiness_Rank" collection, the "Global_Rank" key was deleted. In the "Happiness_Mean_Stats" collection, the "Global_Rank" key was changed to a "Country" key, as exemplified in Figure 5.

With these changes in the data model, most queries also had to be modified to give the information initially desired. Therefore, the queries were adapted as follows:

For the two simple queries and the insert/update, none or minimal changes were made, requiring only the adaptation of the several keys for the insert operation and changing the collection whose simple queries were applied to the "Country" collection.

For the complex queries, both required adaptations after the data model changes. Similarly to the SQL schema change, the first complex query no longer needed two "$lookup" operations. Instead, the information could be acquired from a simple find query applied to the "Country" collection. On the other hand, the second complex query required a single "$lookup" operation instead of two since only information from the "Country," and "Happiness_Mean_Stats" collections were needed.

The columns each hashed index had to be applied also changed. For the modified schema, the hashed indexes were applied to the "Country" key from the "Happiness_Rank" and "Happiness_Mean_Stats" collections.

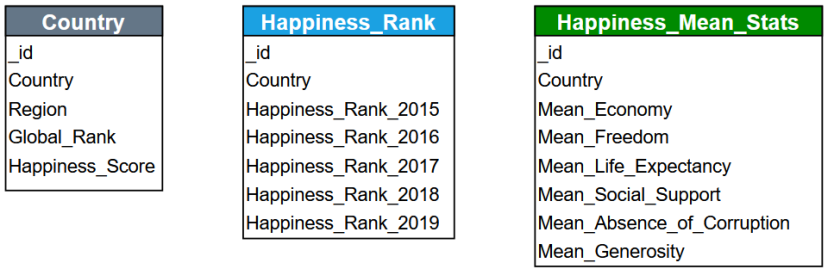

Figure 5 *– Modified MongoDB database data model. A total of three collections were created ("Country"; "Happiness Rank"; "Happiness Mean Stats").*

6 - Performance during each step

To better understand the performance of the databases for each query, the time it took to perform it before and after each optimization step was recorded. However, since the chosen dataset only included 170 rows, any query that is applied to the databases would be extremely fast given its small size, making the differences before and after optimization more challenging to visualize. Therefore, to better understand if optimization led to better performance, 499830 new rows were inserted into the dataset, which should make the query process slow enough to visualize differences in performance.

Since the query performance depends on the computer on which these queries are carried on, it is important to note that all the results presented below were obtained on a computer with the following characteristics: Windows 11 Home 64-bit, processor AMD Ryzen 5 5600G Six-Core Processor 4.0 GHz, 16 GB RAM.

Table 1 presents the time it took to complete each query for both databases with the original data (170 rows) and 500000 rows.

From Table 1, it is possible to see that the complex queries took a substantial amount of time when compared with the simple queries on both databases. Optimizations for one of the complex queries were performed, which led to considerable time improvements. Considering 500000 rows,

for SQLite, there was a tenfold reduction of the time needed to complete the query, while for MongoDB, the optimization allowed a fivefold time reduction. When using indexes in both databases, a drastic reduction in the amount of time was noticed for one of the complex queries. In contrast, there were no significant changes in the performance of the other complex query. With the optimized schema, another drastic reduction in time was noticed for both the complex queries in both databases.

Table 1 *– Time to perform each query in both databases. For SQLite, times were obtained using DB Browser. For MongoDB, times were obtained using the time function in python. The incremental betterment of the query times is displayed using a colour gradient (red for the worst times before optimization and green after all optimizations). A comparison between databases is observed on the "Queries" column, whose colour indicates which database had the best performance for a given type of query (Green – Better database; Red – Worse database).*

DB	Queries		1. Original Queries		2. Optimized Queries		3. Indexes applyed		4. Optimized schema	
			nº rows		nº rows		nº rows		nº rows	
			170	500000	170	500000	170	500000	170	500000
SQLite	Simple	1	2 ms	3 ms						
		2	2 ms	3 ms						
	Complex	1	5 ms	3237 ms	4 ms	208 ms	3 ms	188 ms	3 ms	63 ms
		2	6 ms	5455 ms			3 ms	2455 ms	3 ms	1905 ms
	Insert		1 ms	9292 ms						
	Update		1 ms	225 ms						
MongoDB	Simple	1	1ms	1 ms						
		2	1ms	1 ms						
	Complex	1	34 ms	188188 ms	10 ms	34676 ms	3 ms	34522 ms	1 ms	1 ms
		2	21 ms	> 5 min			25 ms	44172 ms	9 ms	6828 ms
	Insert		2 ms	17310 ms						
	Update		4 ms	3869 ms						

Comparing the query times of MongoDB and SQLite, MongoDB simple queries were faster. For SQLite, the complex, insert and update queries were faster than MongoDB. Since complex queries related information between tables/collections it makes sense that these are faster in SQLite, which uses primary and foreign keys to explicit relations. This is corroborated by the query time of the first complex query using MongoDB after data model changes, which transformed the query into a simple find and became must faster than SQLite. The insert/update queries were also faster in SQLite, which could be due to using distinct ways to see query performance or due to the small size of the data (500000 rows is not a lot in Big Data).

7 - Trade-offs between each design choice for each query

The optimization of the complex queries did not have any trade-offs, as the performance was gained simply by using a cleaver approach to the query.

Regarding the use of indexes, a possible trade-off may occur if the database has a lot of rows. In this case, implementing indexes may take a lot of time, which is undesirable.

Lastly, a clear trade-off exists for the database schema/data model changes as the gained performance from these changes was accompanied by database denormalization.

8 - Conclusions

This project allowed a comparison between SQL and NoSQL databases using SQLite and MongoDB. The same schema/data model was designed for both databases to allow a fair comparison. Different queries with distinct levels of complexity were developed, through which it was concluded that SQL queries are generally easier to make and more understandable compared to NoSQL queries. Furthermore, several database optimizations were performed, from query optimization to indexing and schema/data model alteration. These optimizations allowed an incremental betterment of the query performance. Lastly, query performance between databases was also investigated, where it was concluded that MongoDB was generally faster, except for complex queries, which only got faster after optimization.