Faculdade de Ciências da Universidade de Lisboa

Master's Degree in Data Science

&

Master's Degree in Informatics

Advanced Databases (2022/2023)

Project Report

"Comparison between relational and noSQL databases

for data modelling, querying and optimization"

Group 5

Cláudia Afonso 36273, Matilde Carvalho 54838, Nihan, Rita Rodrigues 54859

4th of December 2022

I. Introduction

The goal of this project is to compare a relational and a non-relational database with respect to data modelling, querying and optimization. As an example of a relational database, SQLite through the sqlite3 module from the Python standard library was used. On the other hand, as an example of a non-relational database, the document-oriented database program MongoDB was employed. To replicate the creation of our databases, the files should be run in the following order: (1) “SQLite.py”, (2) “csv_to_json.py”, (3) “mongo_write.py”, (4) “queries.py”, and (5) “indexing.py”.

II. Database selection

To tackle this project, an appropriate dataset was first selected from Kaggle. As per the project description, the dataset needed to have at least three CSV files with at least one column in common. After an extensive search, the dataset related to flight delays and cancellations in the United States of America during the year 2015 was chosen. This dataset can be found in the following link: https://www.kaggle.com/datasets/usdot/flight-delays. This dataset is composed of 3 CSV files named “airlines”, “airports” and “flights”. As the name implies, the “airlines” and “airports” CSV files contain information related to the airline companies and airports, while the “flights” CSV file contains information pertaining to the flights.

III. Database schema design

Before creating the databases in SQLite and MongoDB, their respective schemas were first designed. The first step in this process consisted in evaluating how the CSV files were related to each other by inspecting which columns contained the same information in the three files. Here, it was possible to conclude that the “IATA_CODE” column in the “airlines” CSV file is related to the “AIRLINE” column in the “flights” CSV file, while the “IATA_CODE” column in the “airports” CSV file is related to the “ORIGIN_AIRPORT” and “DESTINATION_AIRPORT” columns in the “flights” CSV file. Therefore, we decided to create three tables for the relational database and three collections for the non-relational database, where each table or collection corresponds to each CSV file.

To create a primary key for the “flights” table in the relational database, it was decided to create a new column called “ID” with distinctive values for each record. Furthermore, to facilitate the comprehension of our databases, the “IATA_CODE” attribute (SQLite) or field (MongoDB) in the “airlines” table (SQLite) or collection (MongoDB) was renamed to “AIRLINE_CODE” during the creation of the databases, so as to not confuse it with the attribute/field of the same name in the “airports” table/collection.

Concerning the design of the relational database named “group5_sql_no_index”, the table “airlines” has the attribute “AIRLINE_CODE” as a primary key, the table “flights” has the “ID” attribute as a primary key and the table “airports” has the attribute “IATA_CODE” as a primary key. In the “flights” table, three foreign keys were defined, called “ORIGIN_AIRPORT”, “DESTINATION_AIRPORT” and “AIRLINE_CODE”. The first two reference the primary key “IATA_CODE” in the “airports” table, while the last one references the “AIRLINE_CODE” in the “airlines” table.

To design the schema of the non-relational database “group5_mongo_no_index”, three JSON files representing the collections named “airlines”, “flights” and “airports” were created. Each JSON file is a list of dictionaries, where each dictionary corresponds to a record called document, which is a data structure composed of field and value pairs.

In the non-relational database “group5_mongo_no_index” created using MongoDB, it is not necessary to define a primary key for each collection. This is because the “_id” field, which is automatically generated during the creation of any document, corresponds to a unique identifier for each document and is therefore treated as the primary key within a MongoDB collection.

The next step in the database schema design consisted in deciding which columns in the CSV files we wanted to maintain as attributes or fields, since some of these were dispensable for our project. Finally, the data types associated with each attribute or field in the relational and non-relational databases were defined.

The resulting schemas of the relational and non-relational databases named, respectively, “group5_sql_no_index” and “group5_mongo_no_index”, are shown in Figure 1 and Figure 2.

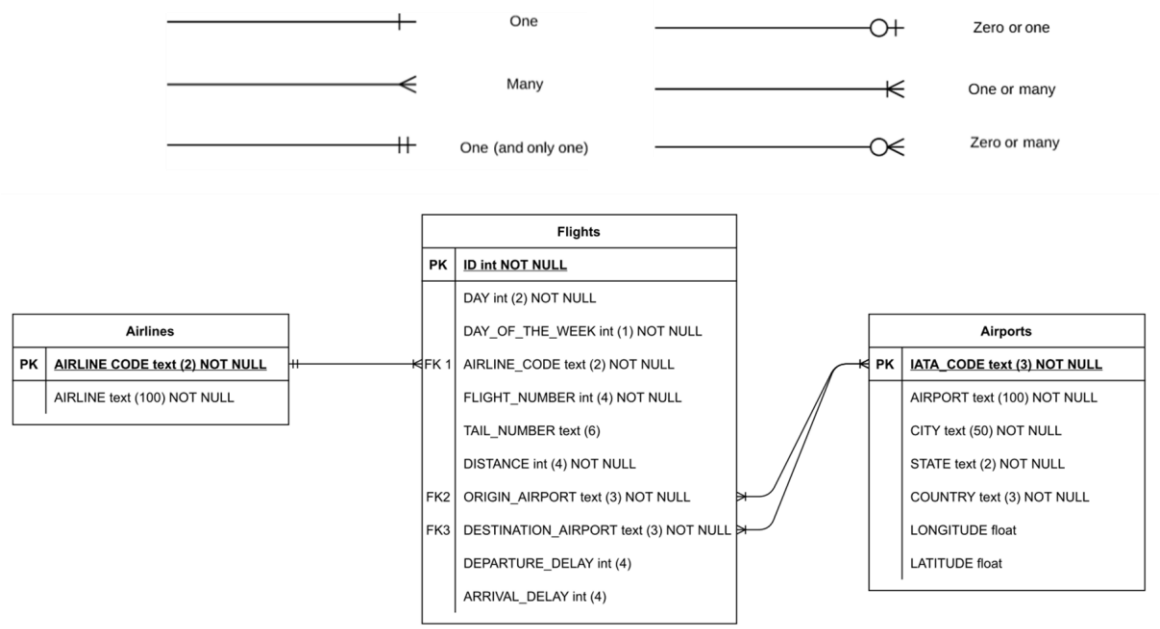


Figure 1 – Schema of the relational database named “group5_sql_no_index”.

Collection “airlines”

```
[
    {
      "AIRLINE_CODE ": TEXT,
      "AIRLINE": TEXT
    }
]
```

Collection “flights”

```
[
    {
      "DAY": INTEGER,
      "DAY_OF_WEEK": INTEGER,
      "AIRLINE_CODE": TEXT,
      "FLIGHT_NUMBER": INTEGER,
      "TAIL_NUMBER": TEXT,
      "DISTANCE" : INTEGER,
      "IATA_CODE":{
          "ORIGIN_AIRPORT": TEXT,
          "DESTINATION_AIRPORT": TEXT
      },
      "DELAYS":{
          "DEPARTURE_DELAY": FLOAT,
          "ARRIVAL_DELAY": FLOAT,
      }
    }
]
```

Collection “airports”

```
[
    {
      "IATA_CODE": TEXT,
      "AIRPORT": TEXT,
      "CITY": TEXT,
      "STATE": TEXT,
      "COUNTRY": TEXT,
      "COORDINATES":{
          "LONGITUDE": FLOAT,
          "LATITUDE": FLOAT
      }
    }
]
```

Figure 2 – Schema of the non-relational database named “group5_mongo_no_index”. The three JSON files created in the first step of the project correspond to the three collections named “airlines”, “flights” and “airports” of the database “group5_mongo_no_index”.

IV. Creation of the databases

After designing their schema, the second step of the project consisted in creating the relational and non-relational databases using SQLite and MongoDB, respectively. To accomplish this, three files with a “.py” extension were created at this stage.

Relational Database

The first file, appropriately named “SQLite.py”, contained the code necessary to create the relational database according to the schema presented in Figure 1. In this file, each CSV was read into a pandas dataframe. In the “airlines” dataframe, the column named “IATA_CODE” was renamed to “AIRLINE_CODE”. In the “flights” dataframe, the column named “AIRLINE” was renamed to “AIRLINE_CODE”. In addition, in this dataframe only the records corresponding to the January month were selected, several unnecessary columns were dropped and the column named “DISTANCE” was placed between the columns “TAIL_NUMBER” and “ORIGIN_AIRPORT”. In the “airports” dataframe, the column named “LONGITUDE” was placed before the column named “LATITUDE”. For each dataframe, a list of tuples (with each tuple corresponding to a record of the dataframe) was created, in order to subsequently introduce the data in the tables of the relational database.

The three tables, named “airlines”, “airports” and “flights” of the relational database were then created according to the schema presented in Figure 1 and the corresponding data was introduced.

Non-relational Database

To create the non-relational database using MongoDB, each CSV file was first read into a pandas dataframe, the dataframes modified according to the schema shown in Figure 2, and finally the dataframes were written to JSON files. The code corresponding to this task is contained within the “csv_to_json.py” file.

To maintain the relational and non-relational databases consistent with each other, in the “csv_to_json.py” the column named “IATA_CODE” in the “airlines” dataframe was renamed to “AIRLINE_CODE”. Furthermore, in the “flights” dataframe the column named “AIRLINE” was renamed to “AIRLINE_CODE”, only the records corresponding to the January month were selected and the same unnecessary columns that had been dropped in the “SQLite.py” file were also dropped here. Additionally, for the “IATA_CODE” field, a dictionary was created for its value, where the keys are “ORIGIN_AIRPORT” and “DESTINATION_AIRPORT”. For the “DELAYS” field a dictionary was also created, where the keys are “DEPARTURE_DELAY” and “ARRIVAL_DELAY”. In the “airports” dataframe, according to the schema presented in Figure 2, for the “COORDINATES” field a dictionary was initially created for its value, where the keys are “LONGITUDE” and “LATITUDE”. However, to allow the creation of the geospatial index later in the project, this had to be changed to a list, where the first element corresponds to “LONGITUDE” and the second element to “LATITUDE”.

After creating the JSON files from their respective CSVs with the “csv_to_json.py” file, the non-relational database was finally created in MongoDB. The code corresponding to this task is contained within the “mongo_write.py” file. It was possible to visualize the created database using MongoDB Compass, with representative images of the 3 collections (“airlines”, “flights” and “airports”) that compose our database shown in Figure 3.

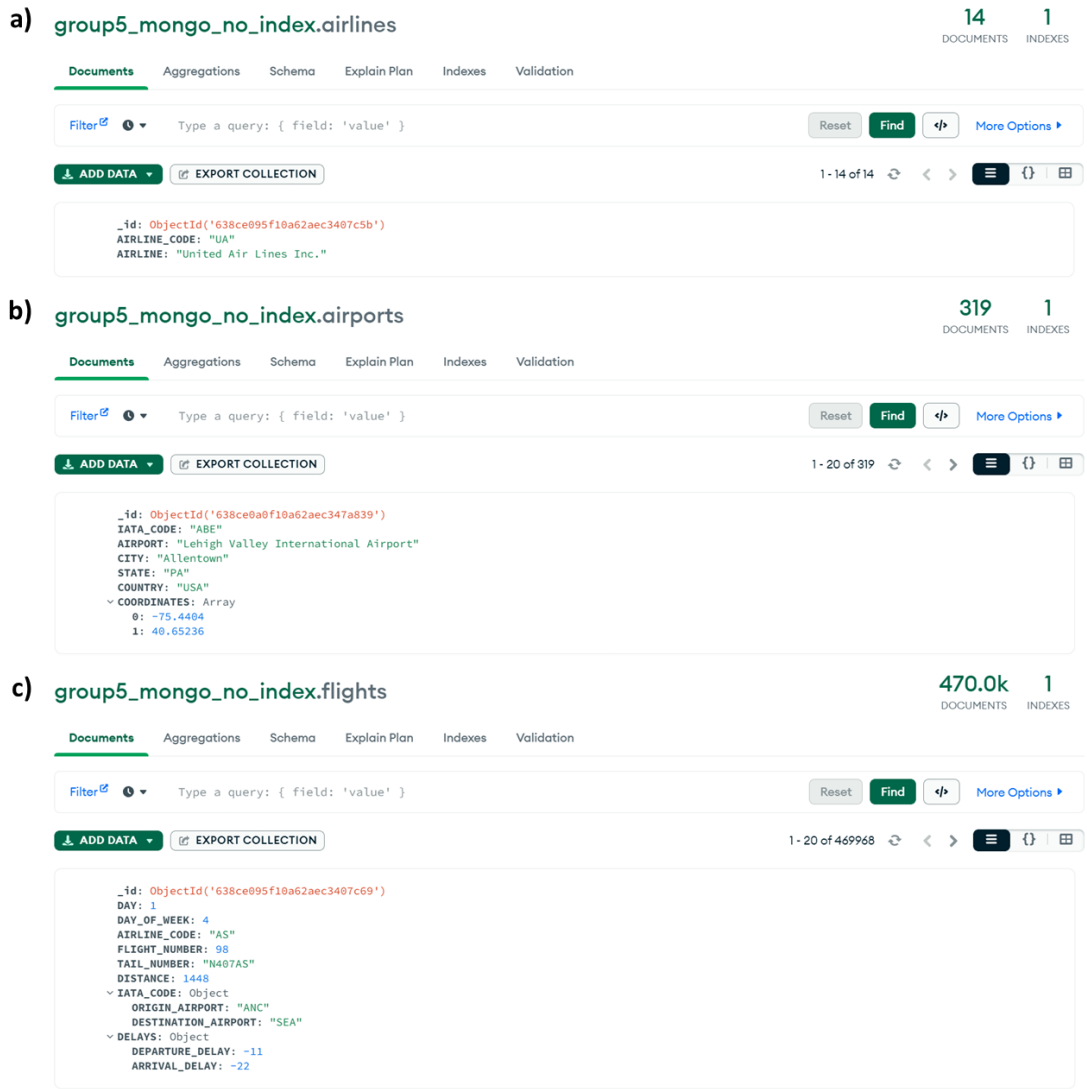


Figure 3 – Visualization of the non-relational database named “group5_mongo_no_index” in MongoDB Compass. In **a)** 1 document from the “airlines” collection is shown, while in **b)** 1 document from the “airports” collection is shown, and in **c)** 1 document from the “flights” collection is shown.

V. Creation of the queries

The subsequent step in the project consisted in creating queries for each relational and non-relational database. The created queries are within the file called “queries.py”.

The first three were simple queries and consisted only in the selection of data from one or two columns (in SQLite) or fields (in MongoDB). Therefore, the following three queries were created:

- Query 1: Select the “FLIGHT_NUMBER” from “flights” table or collection where “FLIGHT_NUMBER” is greater than 7000. The first five results for this query in SQLite and MongoDB are:

```
The first five results for the 1st query in SQLite are:
(7404,)
(7419,)
(7370,)
(7423,)
(7392,)
```

```
The first five results for the 1st query in MongoDB are:
{'_id': ObjectId('638ce095f10a62aec3407cab'), 'FLIGHT_NUMBER': 7404}
{'_id': ObjectId('638ce095f10a62aec3407cac'), 'FLIGHT_NUMBER': 7419}
{'_id': ObjectId('638ce095f10a62aec3407cb8'), 'FLIGHT_NUMBER': 7370}
{'_id': ObjectId('638ce095f10a62aec3407dbc'), 'FLIGHT_NUMBER': 7423}
{'_id': ObjectId('638ce095f10a62aec3407dd1'), 'FLIGHT_NUMBER': 7392}
```

- Query 2: Select the (“AIRPORT”,”CITY”) pairs from the “airports” table or collection. The first five results for this query in SQLite and MongoDB are:

```
The first five results for the 2nd query in SQLite are:
('Lehigh Valley International Airport', 'Allentown')
('Abilene Regional Airport', 'Abilene')
('Albuquerque International Sunport', 'Albuquerque')
('Aberdeen Regional Airport', 'Aberdeen')
('Southwest Georgia Regional Airport', 'Albany')
```

```
The first five results for the 2nd query in MongoDB are:
{'_id': ObjectId('638ce0a0f10a62aec347a839'), 'AIRPORT': 'Lehigh Valley International Airport', 'CITY': 'Allentown'}
{'_id': ObjectId('638ce0a0f10a62aec347a83a'), 'AIRPORT': 'Abilene Regional Airport', 'CITY': 'Abilene'}
{'_id': ObjectId('638ce0a0f10a62aec347a83b'), 'AIRPORT': 'Albuquerque International Sunport', 'CITY': 'Albuquerque'}
{'_id': ObjectId('638ce0a0f10a62aec347a83c'), 'AIRPORT': 'Aberdeen Regional Airport', 'CITY': 'Aberdeen'}
{'_id': ObjectId('638ce0a0f10a62aec347a83d'), 'AIRPORT': 'Southwest Georgia Regional Airport', 'CITY': 'Albany'}
```

- Query 3: Select the “AIRPORT”, “LATITUDE” and “LONGITUDE” from the “airports” table or collection where “LATITUDE” is greater than 30 and lower than 50. The first five results for this query in SQLite and MongoDB are:

```
The first five results for the 3rd query in SQLite are:
('Lehigh Valley International Airport', -75.4404, 40.65236)
('Abilene Regional Airport', -99.6819, 32.41132)
('Albuquerque International Sunport', -106.60919, 35.04022)
('Aberdeen Regional Airport', -98.42183, 45.44906)
('Southwest Georgia Regional Airport', -84.19447, 31.53552)
```

```
The first five results for the 3rd query in MongoDB are:
{'_id': ObjectId('638ce0a0f10a62aec347a839'), 'AIRPORT': 'Lehigh Valley International Airport', 'COORDINATES': [-75.4404, 40.65236]}
{'_id': ObjectId('638ce0a0f10a62aec347a83a'), 'AIRPORT': 'Abilene Regional Airport', 'COORDINATES': [-99.6819, 32.41132]}
{'_id': ObjectId('638ce0a0f10a62aec347a83b'), 'AIRPORT': 'Albuquerque International Sunport', 'COORDINATES': [-106.60919, 35.04022]}
{'_id': ObjectId('638ce0a0f10a62aec347a83c'), 'AIRPORT': 'Aberdeen Regional Airport', 'COORDINATES': [-98.42183, 45.44906]}
{'_id': ObjectId('638ce0a0f10a62aec347a83d'), 'AIRPORT': 'Southwest Georgia Regional Airport', 'COORDINATES': [-84.19447, 31.53552]}
```

The next two were more complex queries using joins and aggregates involving at least 3 tables (in SQLite) or collections (in mongoDB) of our databases. Therefore, the following two queries were created:

- Query 4: Select the “FLIGHT_ID”, “ORIGIN_AIRPORT” (from the “flights” table or collection), “AIRPORT” (from the “airports” table or collection and which corresponds to the name of the origin airport of the flight), “AIRLINE_CODE” (from the “flights” table or collection) and “AIRLINE” (from the “airlines” table or collection). The first five results for this query in SQLite and MongoDB are:

```
The first five results for the 4th query in SQLite are:
(1, 'ANC', 'Ted Stevens Anchorage International Airport', 'AS', 'Alaska Airlines Inc.')
(2, 'LAX', 'Los Angeles International Airport', 'AA', 'American Airlines Inc.')
(3, 'SFO', 'San Francisco International Airport', 'US', 'US Airways Inc.')
(4, 'LAX', 'Los Angeles International Airport', 'AA', 'American Airlines Inc.')
(5, 'SEA', 'Seattle-Tacoma International Airport', 'AS', 'Alaska Airlines Inc.')
```

```
The first five results for the 4th query in MongoDB are:
{'_id': ObjectId('638ce095f10a62aec3407c69'), 'AIRLINE_CODE': 'AS', 'IATA_CODE': {'ORIGIN_AIRPORT': 'ANC'}, 'airport':
 [{'AIRPORT': 'Ted Stevens Anchorage International Airport'}], 'airline': [{'AIRLINE': 'Alaska Airlines Inc.'}]}
{'_id': ObjectId('638ce095f10a62aec3407c6a'), 'AIRLINE_CODE': 'AA', 'IATA_CODE': {'ORIGIN_AIRPORT': 'LAX'}, 'airport':
 [{'AIRPORT': 'Los Angeles International Airport'}], 'airline': [{'AIRLINE': 'American Airlines Inc.'}]}
{'_id': ObjectId('638ce095f10a62aec3407c6b'), 'AIRLINE_CODE': 'US', 'IATA_CODE': {'ORIGIN_AIRPORT': 'SFO'}, 'airport':
 [{'AIRPORT': 'San Francisco International Airport'}], 'airline': [{'AIRLINE': 'US Airways Inc.'}]}
{'_id': ObjectId('638ce095f10a62aec3407c6c'), 'AIRLINE_CODE': 'AA', 'IATA_CODE': {'ORIGIN_AIRPORT': 'LAX'}, 'airport':
 [{'AIRPORT': 'Los Angeles International Airport'}], 'airline': [{'AIRLINE': 'American Airlines Inc.'}]}
{'_id': ObjectId('638ce095f10a62aec3407c6d'), 'AIRLINE_CODE': 'AS', 'IATA_CODE': {'ORIGIN_AIRPORT': 'SEA'}, 'airport':
 [{'AIRPORT': 'Seattle-Tacoma International Airport'}], 'airline': [{'AIRLINE': 'Alaska Airlines Inc.'}]}
```

- Query 5: Select the “FLIGHT_ID”, “AIRPORT”, “AIRLINE” with average delay in ascending order. The first five results for this query in SQLite and MongoDB are:

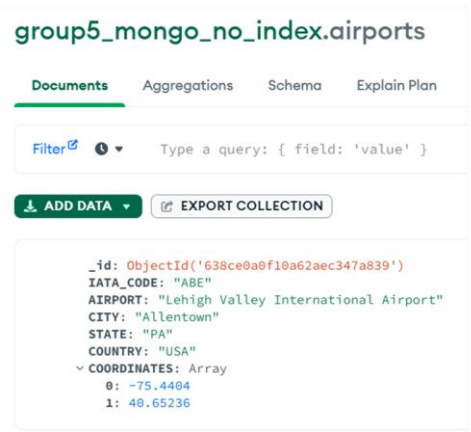
```
The first five results for the 5th query in SQLite are:
(38302, 'Billings Logan International Airport', 'Atlantic Southeast Airlines', None)
(265005, 'Evansville Regional Airport', 'Skywest Airlines Inc.', None)
(26048, 'Central Wisconsin Airport', 'Skywest Airlines Inc.', -28.0)
(4406, 'Canyonlands Field', 'Skywest Airlines Inc.', -26.09259259259259)
(2492, 'Springfield-Branson National Airport', 'Delta Air Lines Inc.', -20.70967741935484)
```

```
The first five results for the 5th query in MongoDB are:
{'_id': {'origin airport': ['Billings Logan International Airport'], 'airline': ['Atlantic Southeast Airlines']}, 'average delays': None}
{'_id': {'origin airport': ['Evansville Regional Airport'], 'airline': ['Skywest Airlines Inc.']}, 'average delays': None}
{'_id': {'origin airport': ['Central Wisconsin Airport'], 'airline': ['Skywest Airlines Inc.']}, 'average delays': -28.0}
{'_id': {'origin airport': ['Canyonlands Field'], 'airline': ['Skywest Airlines Inc.']}, 'average delays': -26.09259259259259}
{'_id': {'origin airport': ['Springfield-Branson National Airport'], 'airline': ['Delta Air Lines Inc.']}, 'average delays': -20.70967741935484}
```

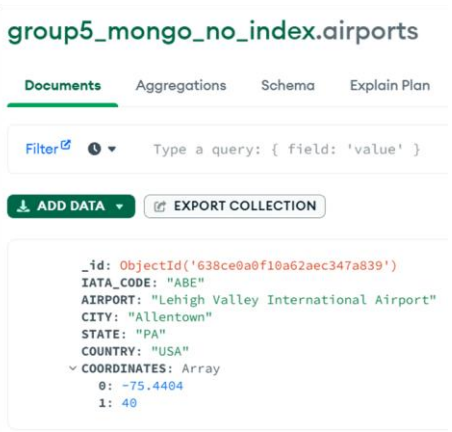
The final two consisted in queries to update existing records or insert new ones:

- Query 6: Update the “airports” table or collection, setting the “LATITUDE” equal to 40 where “IATA_CODE” is ‘ABE’. The document before and after this update query was visualized in MongoDB Compass, as shown below:

Before Update Query

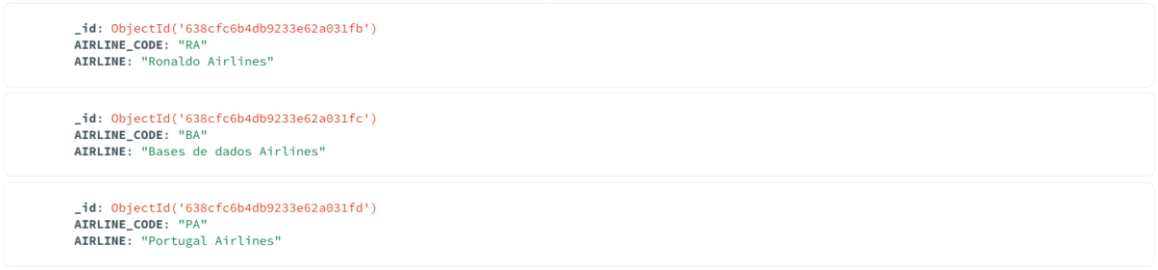


After Update Query

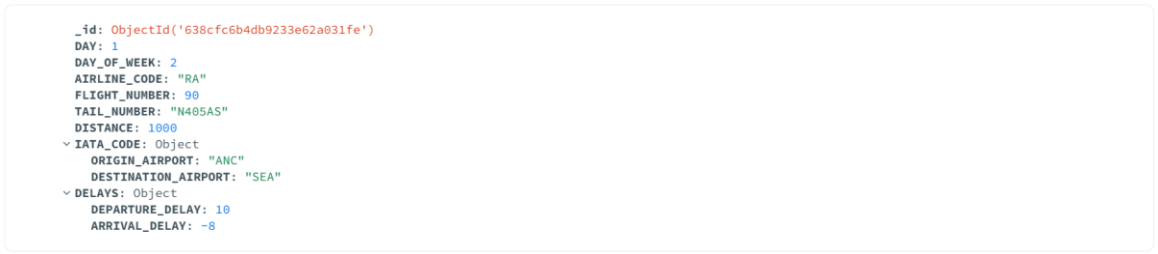


- Query 7: Insert into the “airlines” table or collection three new airlines (‘RA’, ‘Ronaldo Airlines’), (‘BA’, ‘Bases de dados Airlines’), (‘PA’, ‘Portugal Airlines’)

To perform query 7, a new JSON file called “airlinesinsert.json” containing the data to be introduced in the “airlines” collection was created. The inserted documents in the “airlines” collection were visualized in MongoDB Compass, as shown below:



- Query 8: insert a new record in the “flights” table or collection. The inserted document in the “flights” collection was visualized in MongoDB Compass, as shown below:



VI. Indexing and optimization

The final step in the project consisted in creating the indexes to optimize some of the queries that were previously developed. The code for the creation of the indexes is within the file called “indexing.py”. To compare the performance of the queries with and without prior indexing of our databases, two new databases with the created indexes were thus generated, one in SQLite and another in MongoDB called “group5_sql_index” and “group5_mongo_index”, respectively.

Indexes were created to optimize the first, second and third queries described previously. For the first query, a hash index for the “FLIGHT_NUMBER” field was created in MongoDB. Since SQLite does not support hash indexing, a simple index was created for the “FLIGHT_NUMBER” attribute. For the second query, a compound text index on the “AIRPORT” and “CITY” fields or attributes was created. For the third query, a geospatial index for the “LONGITUDE” and “LATITUDE” fields was created in MongoDB. Since SQLite does not support this type of index, a compound index using the “LONGITUDE” and “LATITUDE” attributes was created.

To create the latter geospatial index, the original schema of the “airlines” collection of the non-relational database had to be changed to support this type of indexing. Thus, in the “COORDINATES” field, instead of having for its value a dictionary with “LATITUDE” and “LONGITUDE” as keys, a list was introduced, where the first element corresponds to “LONGITUDE” and the second element to “LATITUDE”.

VII. Performance test of the queries after optimization with database indexing

To compare the performance of the queries 1 to 3 before and after the creation of the indexes, the execution times of the non-optimized and optimized queries in SQLite and MongoDB were determined. These results, which were graphically displayed using the matplotlib library in Python, are shown in Figure 4 and Figure 5, as well as Table 1.

The execution time for query 1 in MongoDB as well as SQLite was significantly lower when using an index created on the search key “FLIGHT_NUMBER”. Furthermore, the execution time for query 1 with and without prior optimization through indexing was considerably lower in MongoDB (without index = $5{,}51 \times 10^{-5}$ s; with index = $3{,}60 \times 10^{-5}$ s) than in SQLite (without index = 5.23×10^{-2} s; with index = 1.96×10^{-3} s). Thus, it is possible to conclude that the choice of an index on the “FLIGHT_NUMBER” search key had a considerable positive impact on the performance of query 1 in SQLite. Furthermore, the choice of a hash index on this same search key in MongoDB also had a positive effect on the performance of query 1.

In MongoDB, the execution time for query 2 was twice as fast when using a compound text index on the “AIRPORT” and “CITY” fields relative to the same query without prior optimization (without index = $2{,}31 \times 10^{-5}$ s; with index = $1{,}22 \times 10^{-5}$ s). Meanwhile, in SQLite the execution times between non-optimized and optimized query 2 were relatively

comparable to each other (without index = 3.40 x 10^{-4} s; with index = 3.52 x 10^{-4}s). For this query, the execution times were approximately 10 times lower in MongoDB than in SQLite. Thus, it is possible to conclude that the choice of a compound index had a positive impact on the performance of query 2 in MongoDB, but not in SQLite.

Finally, the execution time for query 3 was relatively similar before and after optimization in both MongoDB and SQLite. Particularly, in MongoDB the execution time following geospatial indexing for the "LONGITUDE" and "LATITUDE" fields was only slightly lower than without indexing (without index = 1,07 x 10^{-5} s; with index = 9,78 x 10^{-6} s). In SQLite, the execution time for this query was actually twice as high following the creation of a compound index on the "LONGITUDE" and "LATITUDE" attributes than without indexing, although both times were still relatively low (without index = 4,89 x 10^{-4} s; with index = 9,83 x 10^{-4} s). Thus, we can conclude the choice of a index on both coordinates to improve query performance was not adequate. Instead of creating an index for both coordinates, it would have been better to create an index only for the "LATITUDE" or for "AIRPORT" and "LATITUDE" pairs.

Overall, we can conclude that the execution times for the three queries tested in this project were lower in MongoDB than in SQLite and that indexing when properly applied can increase query performance.

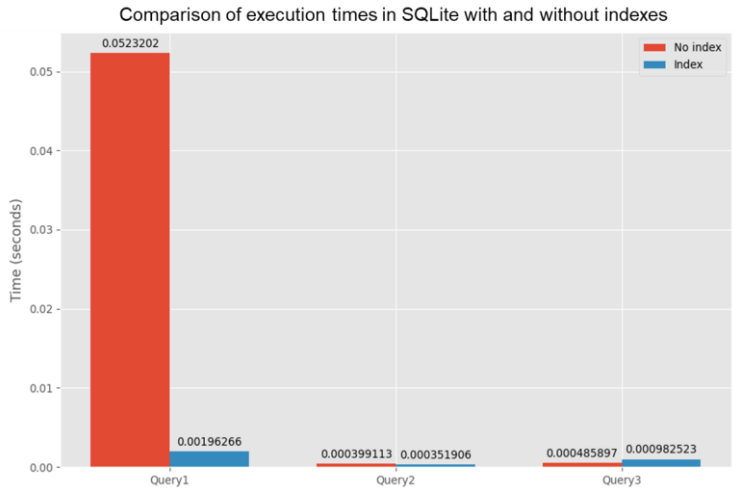


Figure 4 – Graphical representation of execution times in SQLite for queries 1 to 3 with and without indexes.

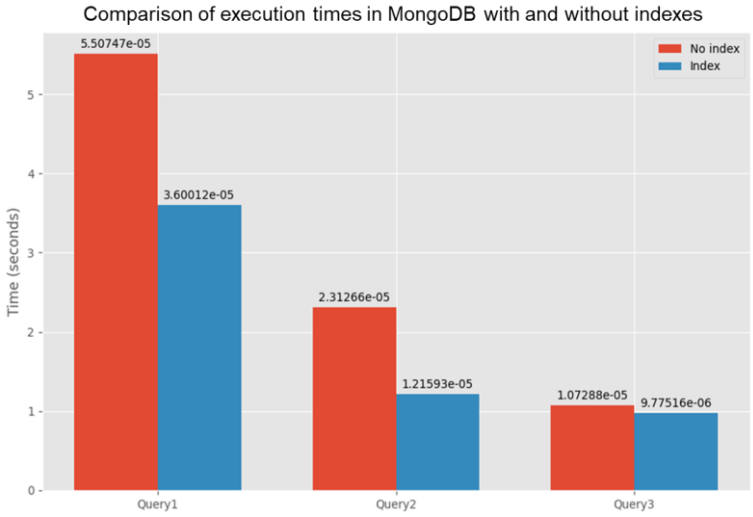


Figure 5 – Graphical representation of execution times in Mongo DB for queries 1 to 3 with and without indexes.

Table 1 – Tabular representation of execution times in MongoDB and SQLite for queries 1 to 3 before and after database indexing

	No index			Index		
	Query1	Query2	Query3	Query1	Query2	Query3
SQLite	0,0523202	0,000399113	0,000485897	0,00196266	0,000351906	0,000982523
MongoDB	5,50747e-05	2,31266e-05	1,07288e-05	3,60012e-05	1,21593e-05	9,77516e-06