Ciências ULisboa Faculdade de Ciências da Universidade de Lisboa

Advanced Databases/Databases Technologies 2022/2023

Project Report

Inês Fragoso 54454, Robert Morosan 54456, Miłosz Włoch 59437, Martim Silva 51304

Description of how to replicate the project

Both of the infrastructures involved in this project, SQLite and MongoDB, have 2 separate files, one for creating the respective database and one for the queries and indexes. The beginning of the filenames identifies which database will be used in it. For each of the four files, all cells from top to bottom should be executed by clicking on “Cell -> Run All”, and the order to run the files is the following:

1. *sqlite_create_db.ipynb*, *mongodb_create_db.ipynb* -> Used for database creation and data cleaning/insertion into each database;
2. *sqlite_queries.ipynb*, *mongodb_queries.ipynb* -> Used for query creation and also optimization coupled with the implementation of adequate indexes for each database.

Note:

The original csv files should be kept in a folder titled “archive” in the root directory where all the *Jupyter Notebook* files are located and without alterations to their titles and contents.

Source:

https://www.kaggle.com/datasets/kwullum/fatal-police-shootings-in-the-us?select=PoliceKillingsUS.csv

Description of the dataset

This dataset is a compilation done by the American newspaper "The Washington Post" since the start of 2015 up until around 2017 of reported US police shootings that resulted in casualties. The information gathered includes the race, age and gender of the person who was fatally shot, if the person was carrying a weapon, when/where this shooting took place, whether the victim showed signs of mental illness, if the police officer had a body camera, among other relevant information about each shooting.

This dataset also contains information regarding the median household income, the percentage of people over the age of 25 that have completed high school, the poverty rate and the shares of people of a given race (white, black, hispanic, asian, native american or other) per each city. All of this information was originally extracted from the United States Census.

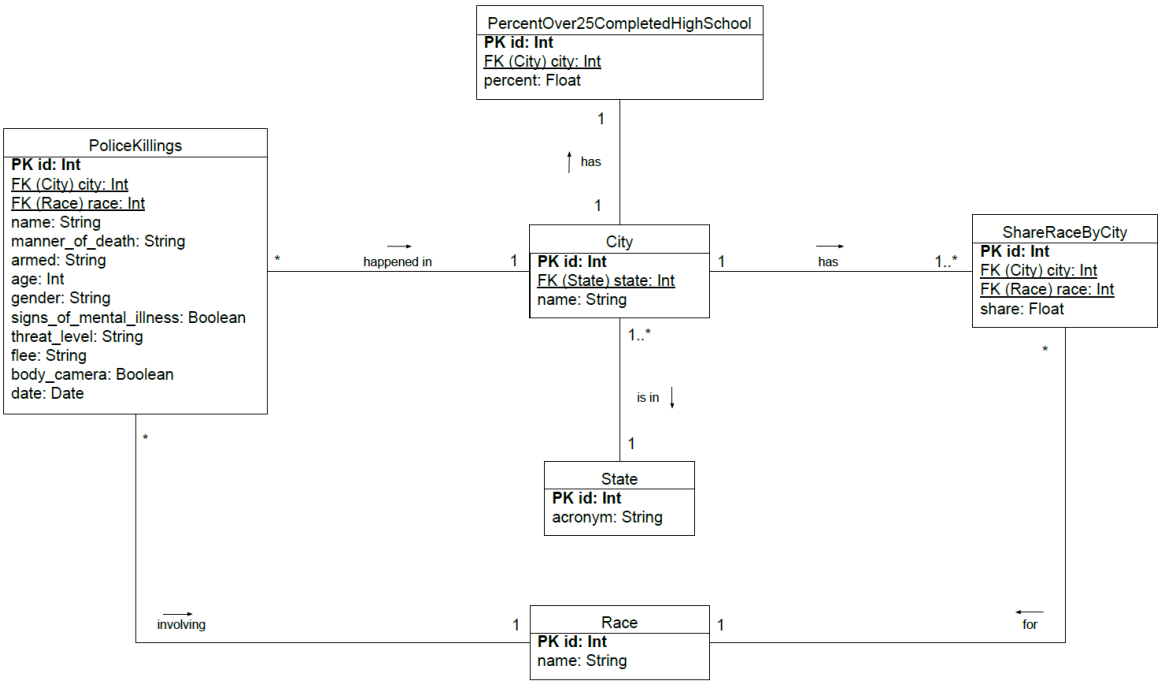
Note:

For our project, we decided to only use 3 of the CSVs (PoliceKillingsUS.csv, ShareRaceByCity.csv, PercentOver25CompletedHighschool.csv).

Goal 1

We decided to select this dataset over the others as it provides a big quantity of *relatively* clean data and a small amount of missing/mismatched data. This topic also provides interesting information upon further exploratory analysis. It is worth mentioning that every CSV file has at least one column that is somehow linked to another file, easily allowing setting foreign keys when converting these files to collections/tables.

We elaborated the following diagram to represent our schema, which is based on the Unified Modelling Language standard and is the same for both of the databases.



Note:

For the SQLite implementation, all id columns are auto incremental.

These several tables/collections were created in order to remove repetition of data and to group it logically:

- **State Table / Collection**

  Each city is located inside a state. One state can have multiple cities. So it would make sense to create the table/collection state, which has a static amount of rows (50 states). Now, every city can reference its state by its id, removing repetition of state acronyms.

  CSVs used: PercentOver25CompletedHighschool.csv

- **City Table / Collection**

Each police killing happens inside a city and a state. State is also provided since there can be two cities with the same name in different states. Given this, we know that the pair (city, state_id) must be unique. By creating a table/collection city, which has an id, the name of the city and the id of the state it's in, a police killing report can reference the city by using the city_id.

CSVs used: PercentOver25CompletedHighschool.csv

- **Race Table / Collection**

  ShareRaceByCity references races, and police killing does as well. Instead of having the race saved as a simple piece of text, creating a table/collection race allows us to connect these two tables/collections, which is helpful for queries, and remove repetition. This table will have a static amount of rows (5 races + "Other").

  CSVs used: ShareRaceByCity.csv

- **ShareRaceByCity Table / Collection**

  Instead of having a row which provides us with the share of all the races in a given city, we decided to create several rows which give us the share of **one** race per city, making use of the Race table. This table has a city_id, which connects it to its city, race_id which makes the connection to race, and the float value corresponding to the share. By choosing this way of structuring the data, we can benefit from knowing what is the share of a specific race, using the race table.

  CSVs used: ShareRaceByCity.csv

- **PercentOver25CompletedHighschool Table / Collection**

  For each city (represented as city_id), we have a value representing the percentage of people over 25 that completed highschool.

  CSVs used: PercentOver25CompletedHighschool.csv

- **PoliceKillings Table / Collection**

  For each police killing report, we store the race of the deceased (represented by race_id), the city in the state where the killing happened (represented by city_id) and all of the other relevant information.

  CSVs used: PoliceKillingsUS.csv

All the tables above have the column id (or _id in MongoDB) as a primary key.

In **SQLite**, this column stores integers and is auto incremented. Note that, even if the PoliceKillingsUS.csv provides us with an id column, we decided to simply generate our own to maintain a sequential id order after cleaning the data.

In **MongoDB**, this column stores ObjectIds which are autogenerated by Mongo.

Separating the data in the dataset by creating these tables/collections also helps us have better data maintenance and less redundancy.

Goal 2

For the second goal of the project, we were expected to create the database for the relational infrastructure in SQLite and the NoSQL infrastructure in MongoDB using Python and the libraries studied in the practical classes, all while maintaining the structure of the data defined in the previously built UML diagram.

In order to import the CSV files into the databases, the data had to undergo prior processing using **Pandas**:

- Replacing recurring, but not universal, ambiguous suffixes of city names (namely "town", "city", "CDP") for the data obtained from PercentOver25CompletedHighschool.csv and ShareRaceByCity.csv. This was done so that there could be a connection between the city name from PoliceKilling and the city in the City Table, given that the cities names in PercentOver25CompletedHighschool.csv and ShareRaceByCity.csv were mismatched;

- Date formatting (12/02/2015 -> 12-02-15) so that SQLITE recognized the date as a DATE type;

- In the City Table, converting column state to state_id. This was done with the use of a dictionary, since, after inserting the unique states in the db, a dictionary was created where we mapped the acronym of the state to its corresponding id in the db;

- In the PoliceKillings & ShareRaceByCity Table, converting the race to race_id. This was done with the use of a dictionary, since, after inserting the unique races in the db, a dictionary was created where we mapped the race name to its corresponding id in the db;

- In PoliceKillings, ShareRaceByCity & PercentOver25CompletedHighschool Table, converting the unique pair of columns (city, state) to city_id. This was done with the use of a dictionary, since, after inserting the unique pairs of (city, state_id) in the db, a dictionary was created where we mapped the pair (city, state) to its corresponding id in the db (also using the dictionary to convert state to state_id);

- In the PercentOver25CompletedHighschool Table/Collection, removing rows containing shares of “-” value;

- In the ShareRaceByCity Table/Collection, removing rows containing shares of “(X)” value;

- In the ShareRaceByCity & PercentOver25CompletedHighschool Table/Collection, translate the share and percentage (stored as string) to float;

- Renaming column names;

- In some cases, the pair (city, state) in the police report does not exist in our city table. In those cases, the row is not included.

In **SQLite**, the columns converted to columns_id were considered foreign keys.

In **MongoDB**, the same applies, even if there isn't any constraint of foreign keys since the db is non relational.

Goal 3

For the third goal of the project, we were supposed to create at least six different queries for both databases, each one needing to correspond to the ones performed in the other database. We created in total 6 queries, 2 simple queries which selected data from one or two columns in the same table/collection, 2 complex queries that used joins/lookups and aggregates, one update and one insert query.

Below is a table with the output obtained from the simple and complex queries implemented in both databases.

	Simple Query 1	**Simple Query 2**	**Complex Query 1**	**Complex Query 2**
Query Details	Unarmed Males under 18 killed by Police	Women killed by Police while not fleeing and not armed	People above the age of 20 armed with a knife who were shot and tasered by Police in the state of Texas	State with the most shootings against males under the age of 18
Result Table for the query in both DBs	[('Jeremy Mardis', 6), ('Jordan Edwards', 15), ('Jose Raul Cruz', 16), ('Deven Guilford', 17), ('Armando Garcia-Muro', 17), ('David Joseph', 17)]	[('Ciara Meyer', 12), ('Alteria Woods', 21), ('India Kager', 28), ('Autumn Steele', 34), ('Justine Damond', 40)]	('Randall Lance Hughes', 48, 'TX', 'knife') ('Henry Reyna', 49, 'TX', 'knife') ('Gregory Mathis', 36, 'TX', 'knife') ('Ray Valdez', 55, 'TX', 'knife') ('Rodney Henderson', 48, 'TX', 'knife')	{'_id': 'CA', 'shootings': 11}

In the insert query, we created a new police killing report filled with the necessary information.

In the update query, we updated the previously inserted police killing report (by using its id) and simply changed the date of the occurrence.

Goal 4

For the fourth and final goal of the project, we tried to make our simple and complex queries faster by applying indexes for specific attributes of the databases, and optimising said queries.

For **SQLite** and **MongoDB**, we created three same indexes.

- **Unique index:**
  acronym (**state** table)

This index is unique since values cannot be repeated. It is useful for complex_query_1 since it helps to find the state with the acronym “TX”.

- **Compound index:**
  armed, gender, age, flee, manner_of_death, name (**police_killings** table)

  Useful for the simple queries and complex_query_1, since the WHERE conditions involve some of these columns. Also, regarding the simple queries, since they SELECT name, the usage of this index will be **covering** (all necessary information is in the index).

- **Compound index:**
  gender, age, city_id (**police_killings** table)

  Useful for complex_query_2, since, besides the WHERE conditions, the GROUP will use the city_id section of the index for better performance.

After indexes were created, we tried to optimise our complex query statements. In the case of the first complex query we changed our LEFT JOIN clauses into INNER JOIN, since an INNER JOIN would not have to check for null values like LEFT JOIN thus making the operation faster. In the case of the second query we replaced the subquery by JOIN instructions and made it less composite. JOIN clauses tend to execute faster as they use optimised query plans.

Performance for SQLite and MongoDB

To evaluate the performances obtained after the last goal, we made a query performance comparison between using and not using indexes. This comparison was obtained by calculating the mean of the query time, with and without indexes, in 20 iterations, ensuring a more reliable value.

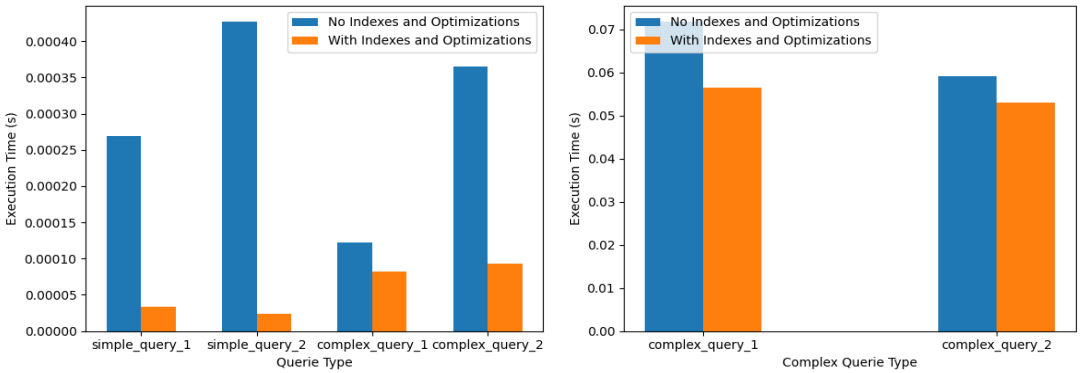
Given the results in the table below, it is possible to obtain the following results:

- In SQLite, using indexes leads to a faster query execution times for all queries;
- In MongoDB, using indexes doesn’t significantly affect execution time for simple queries;
- In MongoDB, using indexes leads to a slightly faster query execution time for complex queries;
- MongoDB is significantly faster for queries that regard only one table/collection. However, the times for complex queries using SQLite are better than the ones using MongoDB;

Comments and explanations for the obtained results:

Regarding simple queries in MongoDB, the execution time does not vary much since the execution time is already incredibly small. On the other hand, we verified that the number of examined documents when using the indexes is significantly smaller compared to no indexes (5 vs 2540 and 6 vs 2540).

Regarding MongoDB performance with complex queries, it was expected that MongoDB, being a non-relational database, would theoretically have faster query time. However, we verified that for complex queries in our database, SQLite was faster. This is because MongoDB is better suited for Big Data databases, in contrast to the one used in our project. In addition, SQLite relational databases handle complex queries and relations between tables comparatively well, since NoSQL databases weren't created to be able to perform great in these specific areas.



Note:

Results for Complex Query 1 and Complex Query 2 in SQLite, with indexes, were obtained using optimised query statements.