Database Systems

Project 2

Student Name: Chandrika Mukherjee
Student ID: 32808289

Part 1: Understanding the data

1-1. No, not each column value is present in each data entry.
The columns with missing entries for each file are mentioned below.

Data File	Columns
airport	city, island, airport_province, elevation
city	population, elevation, *country_capital*
country-other -localname	othername
country	capital, province
countrypopulations	capital
economy	agriculture, service, inflation, industry, gdp, unemployment
ethnicgroup	country_capital, ethnic_group_percentage
language	country_capital, percentage
organization	country, city, established, province
politics	independence, dependent, wasdependent, government
population	infantmortality, populationgrowth, country_province
province	capprov, capital, population

1-2. Yes, there are redundant columns in some data file.

- **airport:** country_name is redundant as using country_code, we can infer the name of the country from country file.
- **city:** country_name and country_capital are redundant as we can infer these information using associated country_code from country file.
- **cityothername** : country_area, country_capital are redundant as we can infer these information using associated country from country file.
- **countrypopulations:** name and capital are redundant information as all those are already stored in country file.

- **ethnicgroup:** country and country_capital are redundant information as all those are stored in country file.
- **language:** country_area, country_capital are redundanat information as these information can be obtained from country file.
- **located-on:** province_area is redundant as this information is stored in province table.
- **population:** country_name is redundant as this information can be obtained from country file.
- **religion:** country_name, country_population are redundant information which can be obtained from country file.

1-3. Logical Constraints as per below -

- **airport:** latitude range [-90,90], longitude range [-180,180], gmtoffset range[-12, 14]
- **city:** latitude [-90,90], longitude[-180,180]
- **country:** encompass_percentage [0-100]
- **ethnicgroup:** ethnic_group_percentage [0-100]
- **language:** percentage [0-100]
- **religion:** percentage [0-100]

Part 2: Relational database schema

2-1. SQL schema design: Included the file named - **schema_oracle_sql.sql** in the zip file.

2-2. entity-relationship diagram : Included the file named - **Relational_Schema.pdf** in the zip file

---

Part 3: Inserting the data to Oracle DBMS

3-1. Software Parsing Tool:
I have used Python3 to create the parser for Oracle SQL.

Included the file **parse_oracle.py** in **parser_oracle_sql** folder within zip file.

Example-1: python3 parse_oracle.py borders.json borders country2 country1 length

Example-2: python3 parse_oracle.py country.json country name code province capital population area

Example-3: python3 parse_oracle.py country.json continent code encompasses_continent encompass_percentage

Input: program_name json_file_name table_name column_names
(The columns can be in any order.)

Create a script folder where the dataset are present.

Output: new file **table_name_script.sql** will be saved in script folder.
I have included the oracle_script folder within zip folder which contains the output of different execution.

Part 4: Using MongoDB

4-1. I have used the same schema concept according to Part 2 (schema of oracle sql)for Data modelling. Here, I used **Document Embedding** Concept for faster processing of queries. For example, in the country_continent collection, I added the associated country and continent in nested format. This reduces the burden of data lookup among several collections.

The above picture shows one document inside country_continent collection. code contains the country object and encomapasses_continent contains the associated continent object.

4-2. MongoDB Document Generator:

I have used Python3 to write the MongoDB Document Generator.
Required Extra Libraries: pymongo, json, bson
Created a Database named **cs541** for this project in MongoDB.
Included the MongoDB Document Generator file **parse_mongo.py** in **parser_mongo** folder within zip file.
Create a folder - **mongo_script** in the same directory with all dataset.
Output - **table_name_script.json** file will be stored in mongo_script folder.
I have included the mongo_script folder within zip folder which contains the output of different execution.

Example-1: python3 parse_mongo.py borders.json borders 3 2 country1 country2 length country1 country2
This suggests that number of columns to be selected - 3, and out of them 2 will be nested component.

So, next 3 will be names of columns [country1, country2, length]
after that, last 2 columns will be nested component [country1, country2]

Now, for each of the nested columns,

- user gives input: (how many constraints are required to pick the particular object from its collection)
  For this example, for country1, user gives 1 [As, we can pick the country object by directly querying country collection. But if it was an instance of city, then we had to input 3 as we need city, province and country to uniquely identify a city]

- For each constraint,
  - then user gives three space separated command- country code country1
    first is collection name (country)
    second is name of the attribute in that collection (code)
    third is the same column name in this current json file (country1)
    Then we form a query similar to

    db.country.find("code":val[country1])
    and for complicated queries, it will be similar to below -

    db.province.find(
    $and: [

“country.code”: “IRQ” ,
“name”: “Basrah”
])
These nested objects are stored against their column name in a dictionary. Finally the dictionary is written as a json file that can be directly uploaded in mongo compass.

Part 5: Queries
5-1:

I have included Oracle Sql query file (**sql_queries.sql**) and MongoDB query file (**mongo_queries.txt**) in the zip file. All 10 queries for 2 different Database systems are stored in two separate files.

5-2:

	Execution Time(s)	
Query Number	Oracle	MongoDB
5-1-a	0.135	0.001
5-1-b	0.126	0.968
5-1-c	0.129	0.002
5-1-d	0.131	0.022
5-1-e	0.157	0.497
5-1-f	0.15	0.073
5-1-g	0.21	0.015
5-1-h	0.142	1.119
5-1-i	0.139	0.016
5-1-j	0.109	0.099

5-3: With 3NF normalization in oracle sql, the data is segregated into small tables with primary and foreign key associations. Also we create the Oracle SQL schema according to data inter-relations, not according to the queries we will execute. Therefore, Use of several joins during querying the tables enhances the overall execution time in oracle sql.

On the other hand, MongoDB data model is shaped based on the queries which we will be executing most likely so that the execution time is less on queries. We can embed other documents with unique identifier to another document - which reduces the requirement of join (lookup) operations within different collections. Apart from this, there are several other reasons about fast data processing by MongoDB like MongoDB is not ACID: Availability is given preference over consistency.

Here, we saw that for 7 out of 10 queries MongoDB outperformed SQL by drastic margin. But for 3 queries where it depends more on lookup than embedding faces performance issue.

5-4 One Performance Improvement Technique which is applicable to both of the Databases is : **Use Projections (Selections) to Return Only Necessary Data.** This can be used in the intermediate query processing before finally obtaining the result.

	Execution Time(s)	
Query Number	Oracle	MongoDB
5-1-a	0.135	0.001
5-1-b	**0.103**	**0.786**
5-1-c	**0.113**	0.002
5-1-d	0.129	0.022
5-1-e	**0.149**	0.214
5-1-f	0.15	0.073
5-1-g	**0.14**	0.015
5-1-h	0.142	**1.115**
5-1-i	0.139	0.016
5-1-j	0.109	0.099

Highlighted numbers are improvement over previous query performance.

5-5 After Implementing the projection changes in some queries which didn't perform well last time in both the databases, performed a little better. For example, execution time of 5-1-b improved from 0.968 s to 0.786 s (MongoDB). Similarly, execution time of 5-1-g improved from 0.21s to 0.14s in Oracle Sql. But None of the improvement helped one database system to outperform the other database system.