Practical Assignment:

Coursework

Designing and Implementing a Database Structure

[Mohammad Faraj P418072]

Database Design and Implementation

16/4/2024

I confirm that this assignment is my own work. Where I have referred to academic sources, I have provided in-text citations and included the sources in the final reference list.

Introduction and Context

The "Internet Movies Database" website, IMDb, was scraped to gather a list of movies that was then compiled into a CSV file named "movies.csv". The objective was to standardize the data in the file and add it to a database using Python code and SQL statements. This method ensures accurate access to the information by utilizing SQL statements. The report will encompass the normalization process, entity-relationship diagram, code implemented to generate tables and input values, SQL statements utilized to extract information and the corresponding outputs, and a reflective segment that will evaluate the design's strengths, weaknesses, and potential areas of improvement.

Normalization Process

To begin the process, it is important to normalize the CSV file housing an extensive 15-column movie database. These columns comprise of crucial details such as the movie's name, rating, genre, year, country-specific release information, score, votes, director, writer, star, shooting location, budget, gross income, production company, and runtime. Given the diverse nature of this information, it is essential to ensure proper normalization to avoid any issues such as update, deletion, or insertion anomalies.

Normalizing to First Normal Form (1NF):

Ensuring that each row is unique, all values are atomic, and there are no repeating groups in the table is crucial for bringing it to the first normal form. Although there are no repeating groups in the table, the "released" column contains two types of data, date and country, which can still be considered atomic, since it does not affect any other data in the table. However, there are non-unique rows in the table, such as two movies with the same name "The Captain," and there may be one production company that produced multiple movies. As a result, unique IDs must be created, and the table must be split into two.

The table will be split into two distinct tables, the first table, "movie_production", will consist of crucial information such as the movie's name, director, stars, writer, and production company. This data can have an impact on other records, as a single writer may create scripts for multiple films, resulting in a one-to-many relationship with the movie's name. However, the name can be used to identify the production company and other pertinent information only if it is unique. Thus, to maintain uniqueness, unique IDs must be created for these fields, alongside a primary key named "production ID".

The second table however, “movie_general”, will include all secondary information that does not depend on or rely on any other information, like the rating, genre, year, released, votes, score, country, budget, gross, and runtime. With a unique ID named “general ID” to serve as a primary key. It will be linked to the first table using the “production ID” however, it shall be a foreign key in the “movie_general” table referencing the “movie_production” table.

Normalizing to Second Normal Form (2NF):

To convert tables to the second normal form (2NF), all non-key attributes in the tables must rely on the primary key after applying normalization to the first normal form (1NF). The "movie_general" table already meets this requirement, since all attributes depend on the primary key, "general ID". However, in the "movie_production" table, each non-key attribute is dependent on its ID, not the primary key "production ID". As a result, the table should be split into six new tables, each with only one attribute and its corresponding ID, with the ID serving as the primary key. There will also be one "movie_production" table that has the "production ID" as its primary key and includes all the other primary keys from the other five tables, to serve as foreign keys to link all tables to it.

Now, the tables are not just in 2NF, but they are also in 3NF since the 3NF points that there must not be any transitive dependencies in the tables, and there are not.

Entity-relationship Diagram

Here is an entity-relationship diagram, Figure 1, that presents and clarifies the relation between the tables mentioned before.

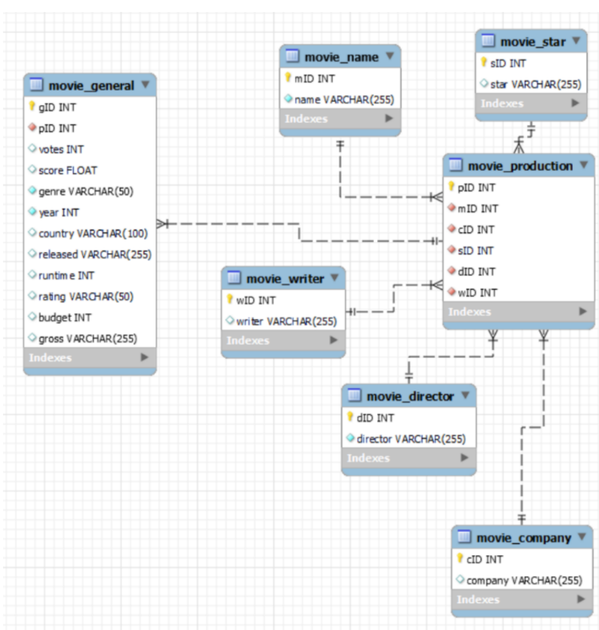

Figure 1: An Entity-relationship Diagram

As how is clear in Figure 1, the relation between the five tables containing the non-key attributes and their IDs, and the “movie_production” tables is a non-identifying one-to-many (1:n) relationship. While the relation between the “movie_production” table and the “movie_general” is also a non-identifying one-to-many relationship.

SQL Database Implementation using Python

In this section, the Python code, along with the SQL statements, used to create the database “coursework1” and the tables, will be presented in Figures 2 and 3, as in below.

```python
import mysql.connector
from mysql.connector import Error

# Establish database connection
connection = mysql.connector.connect(
        host="localhost",
        user="root",
        password="2542",
        database='coursework1'
    )

cursor = connection.cursor()

# cursor.execute("CREATE DATABASE IF NOT EXISTS coursework1")
# cursor.execute("SHOW DATABASES")
# for x in cursor:
#     print(x)

# cursor.execute("CREATE TABLE movie_name (\
# mID int NOT NULL,\
# name varchar(255) NOT NULL,\
# primary key (mID)\
# );")

# cursor.execute("create table movie_company (\
# cID int NOT NULL,\
# company varchar(255),\
# primary key (cID)\
# );")

# cursor.execute("create table movie_star (\
# sID int NOT NULL,\
# star varchar(255),\
# primary key (sID)\
# );")

# cursor.execute("create table movie_director (\
# dID int NOT NULL,\
# director varchar(255) NOT NULL,\
# primary key (dID)\
# );")

```

Figure 2: Code for creating the database and tables, displayed in Visual Studio

```
# cursor.execute("create table movie_writer (\
# wID int NOT NULL,\
# writer varchar(255),\
# primary key (wID)\
# );")

# cursor.execute("create table movie_production (\
# pID int NOT NULL,\
# mID int NOT NULL,\
# cID int NOT NULL,\
# sID int NOT NULL,\
# dID int NOT NULL,\
# wID int NOT NULL,\
# primary key (pID),\
# foreign key (mID) references movie_name (mID),\
# foreign key (cID) references movie_company (cID),\
# foreign key (sID) references movie_star (sID),\
# foreign key (dID) references movie_director (dID),\
# foreign key (wID) references movie_writer (wID)\
# );")

# cursor.execute("create table movie_general (\
# gID int NOT NULL,\
# pID int NOT NULL,\
# votes int,\
# score float,\
# genre varchar(50) NOT NULL,\
# year int NOT NULL,\
# country varchar(100),\
# released varchar(255),\
# runtime int,\
# rating varchar(50),\
# budget int,\
# gross varchar(255),\
# primary key (gID),\
# foreign key (pID) references movie_production (pID)\
# );")

connection.commit()

cursor.close()
connection.close()
```

Figure 3: Completion of the Python code

Data Importing using Python

In Figures 4 through 8, there will be the code for importing information from the CSV file “movies” and inserting them into the tables created before.

database.py > ...

```python
# Importing the random module to generate random numbers
import random
# Importing the pandas module and aliasing it as pd for ease of use
import pandas as pd
# Importing the mysql.connector module to connect to the MySQL database
import mysql.connector
# Importing the Error class from mysql.connector module to handle errors
from mysql.connector import Error

# Function to generate a unique mID
def generate_mID(cursor):
    # Infinite loop to generate a unique mID
    while True:
        # Generate a random mID between 10000 and 99999
        mID = random.randint(10000, 99999)
        # Check if the generated mID already exists in the movie_name table
        cursor.execute("SELECT mID FROM movie_name WHERE mID = %s", (mID,))
        # If mID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return mID

# Function to generate a unique cID
def generate_cID(cursor):
    # Infinite loop to generate a unique cID
    while True:
        # Generate a random cID between 10000 and 99999
        cID = random.randint(10000, 99999)
        # Check if the generated cID already exists in the movie_company table
        cursor.execute("SELECT cID FROM movie_company WHERE cID = %s", (cID,))
        # If cID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return cID

# Function to generate a unique sID
def generate_sID(cursor):
    # Infinite loop to generate a unique sID
    while True:
        # Generate a random sID between 10000 and 99999
        sID = random.randint(10000, 99999)
        # Check if the generated sID already exists in the movie_star table
        cursor.execute("SELECT sID FROM movie_star WHERE sID = %s", (sID,))
        # If sID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return sID

```

Figure 4: Python code for importing information displayed in Visual Studio

```python
# Function to generate a unique dID
def generate_dID(cursor):
    # Infinite loop to generate a unique dID
    while True:
        # Generate a random dID between 10000 and 99999
        dID = random.randint(10000, 99999)
        # Check if the generated dID already exists in the movie_director table
        cursor.execute("SELECT dID FROM movie_director WHERE dID = %s", (dID,))
        # If dID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return dID

# Function to generate a unique wID
def generate_wID(cursor):
    # Infinite loop to generate a unique wID
    while True:
        # Generate a random wID between 10000 and 99999
        wID = random.randint(10000, 99999)
        # Check if the generated wID already exists in the movie_writer table
        cursor.execute("SELECT wID FROM movie_writer WHERE wID = %s", (wID,))
        # If wID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return wID

# Function to generate a unique pID
def generate_pID(cursor):
    # Infinite loop to generate a unique pID
    while True:
        # Generate a random pID between 10000 and 99999
        pID = random.randint(10000, 99999)
        # Check if the generated pID already exists in the movie_production table
        cursor.execute("SELECT pID FROM movie_production WHERE pID = %s", (pID,))
        # If pID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return pID

```

Figure 5: The rest of the code

```python
# Function to generate a unique gID
def generate_gID(cursor):
    # Infinite loop to generate a unique gID
    while True:
        # Generate a random gID between 10000 and 99999
        gID = random.randint(10000, 99999)
        # Check if the generated gID already exists in the movie_general table
        cursor.execute("SELECT gID FROM movie_general WHERE gID = %s", (gID,))
        # If gID does not exist, it's unique, so return it
        if not cursor.fetchone():
            return gID

# Try block to handle potential errors
try:
    # Establish database connection
    connection = mysql.connector.connect(
        host="localhost",
        user="root",
        password="2542",
        database="coursework1"
    )

    # Creating a cursor object to interact with the database
    cursor = connection.cursor()

    # Read movie data from CSV file using pandas
    movieData = pd.read_csv("movies.csv")

    # Iterate over each row in the CSV
    for index, row in movieData.iterrows():
        # Generate unique IDs for each table
        mID = generate_mID(cursor)
        cID = generate_cID(cursor)
        sID = generate_sID(cursor)
        dID = generate_dID(cursor)
        wID = generate_wID(cursor)
        pID = generate_pID(cursor)
        gID = generate_gID(cursor)

        # Replace 'nan' values with None
        row = row.where(pd.notnull(row), None)

```

Figure 6: The rest of the code

As in Figures 4, 5, and 6, there are some functions to generate the unique IDs needed for the tables, where the “random” library was used to generate the random number for each ID, and it only generates a unique ID for each row, by checking if it is present already in the table or not.

Also to deal with ‘nan’ values, this line of code: “row = row.where(pd.notnull(row), None)”, replaces 'nan' values in a Pandas DataFrame row with 'None', using the “where()” function from the Pandas library. (Geeks for Geeks, 2023)

```python
        # Insert values into movie_name table
        cursor.execute("INSERT INTO movie_name (mID, name) VALUES (%s, %s)", (mID, row['name']))

        # Insert values into movie_company table
        cursor.execute("INSERT INTO movie_company (cID, company) VALUES (%s, %s)", (cID, row['company']))

        # Insert values into movie_star table
        cursor.execute("INSERT INTO movie_star (sID, star) VALUES (%s, %s)", (sID, row['star']))

        # Insert values into movie_director table
        cursor.execute("INSERT INTO movie_director (dID, director) VALUES (%s, %s)", (dID, row['director']))

        # Insert values into movie_writer table
        cursor.execute("INSERT INTO movie_writer (wID, writer) VALUES (%s, %s)", (wID, row['writer']))

        # Insert values into movie_production table
        cursor.execute("INSERT INTO movie_production (pID, mID, cID, sID, dID, wID) VALUES (%s, %s, %s, %s, %s, %s)",
                       (pID, mID, cID, sID, dID, wID))

        # Insert values into movie_general table
        cursor.execute("INSERT INTO movie_general (gID, pID, votes, score, genre, year, country, released, runtime, rating, budget, gross) \
                       VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)",
                       (gID, pID, row['votes'], row['score'], row['genre'], row['year'], row['country'],
                        row['released'], row['runtime'], row['rating'], row['budget'], row['gross']))

        # Commit the changes for each row
        connection.commit()

    # Print a success message if no errors occurred
    print("Data inserted successfully.")

# Exception block to handle any errors that occur during execution
except Error as e:
    # Print the error message
    print(f"Error: {e}")

```

Figure 7: Rest of the code

```python
# Finally block to ensure cursor and connection are closed regardless of errors
finally:
    # Check if the connection is still open
    if connection.is_connected():
        # Close the cursor object
        cursor.close()
        # Close the database connection
        connection.close()
        # Print a message indicating that the database connection is closed
        print("Database connection closed.")
```

Figure 8: The end of the code

SQL Statements to Show Correct Implementation

There are 5 types of information needed to be obtained about the data in this database, and they are:

1. The average, minimum and maximum budget of drama movies.
2. The minimum budget for each genre.
3. The average score of each rating (ignore not rated movies).
4. The average runtime of all movies between the years 2014 and 2016.
5. The number of movies released before 2000.

The results for these are obtained in MySQL Workbench by the following SQL statements in Figures 9 through 13.

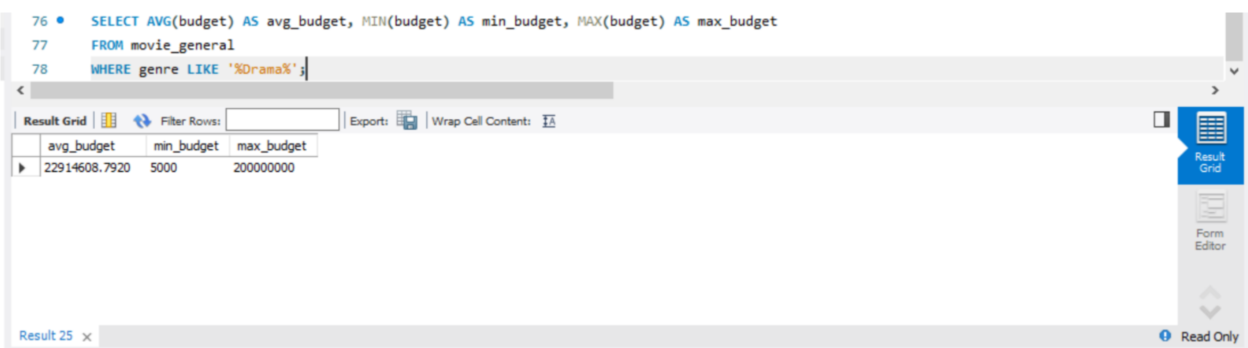

Figure 9: Average, minimum, and maximum budget

```sql
SELECT genre, MIN(budget) AS min_budget
FROM movie_general
GROUP BY genre;
```

genre	min_budget
Action	7000
Comedy	3000
Horror	15000
Drama	5000
Crime	6000
Biography	111000
Adventure	500000
Animation	70000
Fantasy	1000000
Mystery	6900000
Thriller	2500000
Family	10500000
Sci-Fi	370000
Musical	NULL
Western	10000000
Romance	10000000
Music	NULL
History	323562
Sport	NULL

Figure 10: Minimum budget

```sql
SELECT rating, AVG(score) AS avg_score
FROM movie_general
WHERE rating != 'Not Rated'
GROUP BY rating;
```

rating	avg_score
R	6.446347403255376
PG-13	6.287215909497305
PG	6.223562303156898
TV-PG	6.9399998664855955
G	6.590196081236297
TV-MA	7.02222220102946
NC-17	6.547826041346011
Unrated	6.617307699643648
X	6.900000095367432
Approved	3.4000000953674316
TV-14	6.300000190734863

Figure 11: Average score

```sql
SELECT AVG(runtime) AS avg_runtime
FROM movie_general
WHERE year BETWEEN 2014 AND 2016;
```

avg_runtime
110.6150

Figure 12: Average runtime

```sql
SELECT COUNT(*) AS num_movies
FROM movie_general
WHERE year < 2000;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

num_movies
3643

Result 29

Read Only

Figure 13: Number of movies before the year 2000

Also, the results obtained by these statements in Visual Studio, using Python, will be presented in Figures 14 through 18.

```python
# 1. SELECT AVG(budget) AS avg_budget, MIN(budget) AS min_budget, MAX(budget) AS max_budget
#    FROM movie_general
#    WHERE genre LIKE '%Drama%';
cursor.execute("""
    SELECT AVG(budget) AS avg_budget, MIN(budget) AS min_budget, MAX(budget) AS max_budget
    FROM movie_general
    WHERE genre LIKE '%Drama%';
""")
result = cursor.fetchone()
print("Average Budget:", result[0])
print("Minimum Budget:", result[1])
print("Maximum Budget:", result[2])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Average Score: 6.300000190734863
Average Runtime: 110.6150
Number of Movies released before 2000: 3643
PS D:\Study\Database Design and Implementation\Projects> python -u "d:\Study\Database Design and Implementation\Projects\creating tables.py"
Average Budget: 22914608.7920
Minimum Budget: 5000
Maximum Budget: 200000000
PS D:\Study\Database Design and Implementation\Projects>
```

Figure 14: Average, minimum, and maximum budget

```python
# 2. SELECT genre, MIN(budget) AS min_budget
#    FROM movie_general
#    GROUP BY genre;
cursor.execute("""
    SELECT genre, MIN(budget) AS min_budget
    FROM movie_general
    GROUP BY genre;
""")
results = cursor.fetchall()
for row in results:
    print("Genre:", row[0], ", Minimum Budget:", row[1])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Study\Database Design and Implementation\Projects> python -u "d:\Study\Database Design and Implementation\Projects\creating tables.py"
Genre: Action , Minimum Budget: 7000
Genre: Comedy , Minimum Budget: 3000
Genre: Horror , Minimum Budget: 15000
Genre: Drama , Minimum Budget: 5000
Genre: Crime , Minimum Budget: 6000
Genre: Biography , Minimum Budget: 111000
Genre: Adventure , Minimum Budget: 500000
Genre: Animation , Minimum Budget: 70000
Genre: Fantasy , Minimum Budget: 1000000
Genre: Mystery , Minimum Budget: 6900000
Genre: Thriller , Minimum Budget: 2500000
Genre: Family , Minimum Budget: 10500000
Genre: Sci-Fi , Minimum Budget: 370000
Genre: Musical , Minimum Budget: None
Genre: Western , Minimum Budget: 10000000
Genre: Romance , Minimum Budget: 10000000
Genre: Music , Minimum Budget: None
Genre: History , Minimum Budget: 323562
Genre: Sport , Minimum Budget: None
PS D:\Study\Database Design and Implementation\Projects>
```

Figure 15: Minimum budget

```python
# 3. SELECT rating, AVG(score) AS avg_score
#    FROM movie_general
#    WHERE rating != 'Not Rated'
#    GROUP BY rating;
cursor.execute("""
    SELECT rating, AVG(score) AS avg_score
    FROM movie_general
    WHERE rating != 'Not Rated'
    GROUP BY rating;
""")
results = cursor.fetchall()
for row in results:
    print("Rating:", row[0], ", Average Score:", row[1])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Study\Database Design and Implementation\Projects> python -u "d:\Study\Database Design and Implementation\Projects\creating tables.py"
Rating: R , Average Score: 6.446347403255376
Rating: PG-13 , Average Score: 6.287215909497305
Rating: PG , Average Score: 6.223562303156898
Rating: TV-PG , Average Score: 6.9399998664855955
Rating: G , Average Score: 6.590196081236297
Rating: TV-MA , Average Score: 7.02222220102946
Rating: NC-17 , Average Score: 6.547826041346011
Rating: Unrated , Average Score: 6.617307699643648
Rating: X , Average Score: 6.900000095367432
Rating: Approved , Average Score: 3.4000000953674316
Rating: TV-14 , Average Score: 6.300000190734863
PS D:\Study\Database Design and Implementation\Projects>
```

Figure 16: Average score

```
# 4. SELECT AVG(runtime) AS avg_runtime
#    FROM movie_general
#    WHERE year BETWEEN 2014 AND 2016;
cursor.execute("""
    SELECT AVG(runtime) AS avg_runtime
    FROM movie_general
    WHERE year BETWEEN 2014 AND 2016;
""")
result = cursor.fetchone()
print("Average Runtime:", result[0])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Study\Database Design and Implementation\Projects> python -u "d:\Study\Database Design and Implementation\Projects\creating tables.py"
Average Runtime: 110.6150
PS D:\Study\Database Design and Implementation\Projects>
```

Figure 17: Average runtime

```
# 5. SELECT COUNT(*) AS num_movies
#    FROM movie_general
#    WHERE year < 2000;
cursor.execute("""
    SELECT COUNT(*) AS num_movies
    FROM movie_general
    WHERE year < 2000;
""")
result = cursor.fetchone()
print("Number of Movies released before 2000:", result[0])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Study\Database Design and Implementation\Projects> python -u "d:\Study\Database Design and Implementation\Projects\creating tables.py"
Number of Movies released before 2000: 3643
PS D:\Study\Database Design and Implementation\Projects>
```

Figure 18: Number of movies before the year 2000

Reflective Segment

This section aims to provide an in-depth analysis of the challenges encountered by the designer throughout the project. Additionally, any improvements that can be made to the design will be evaluated and their impact on the overall outcome.

Challenges in the design:

Throughout the design process, the initial obstacle was to standardize the database schema. At first, the "movie_production" and "movie_general" tables had a shared primary key, which was the "movie ID" attribute of the "movie_name" table. This allowed users to effortlessly access all the information for each movie in other tables via the "movie ID" attribute as a foreign key. However, this approach had the potential for data anomalies and made maintaining referential integrity difficult. Additionally, it could create complications for database operations, such as updates and deletions. As a result, the design was modified to its current version, where each table has its own primary key that is not a foreign key. This guarantees the same output without any issues.

Another issue arose in the Python code for inserting values into tables. Initially, the logic was to extract all values from a particular column in the CSV file and insert them into the relevant table after assigning IDs if necessary. The code would then repeat the process with the next column. However, this approach was not efficient, as MySQL Workbench automatically rearranges rows in alphabetical order. Consequently, when moving to insert other columns, incorrect values could be assigned from one table to another, given that the ID was randomly generated rather than sequenced. Furthermore, the values would differ every time the code ran. Therefore, the logic was revised to navigate through the CSV file row by row, extract values and store them in variables before inserting them into the tables, and then move to the next row.

Improvements to the design:

An enhancement that could be made to the code involved in utilizing Python functions and the "random" library to generate distinct IDs. Although this method is effective and comprehensible, it may appear slightly complex when compared to an alternative technique that utilizes the "AUTO_INCREMENT" SQL statement during table creation. The latter approach is more straightforward and eliminates the necessity of creating a specific function for each ID.

To handle 'nan' values in a CSV file and ensure their proper storage in a database, we utilized the "where()" function from the Pandas library. This function effectively transformed 'nan' values to "None". While this approach generally works well with most databases, there exists a simpler and more direct method. This method involves assigning a default value of "0" to all 'nan' values. However, this method has its drawbacks as it may lead to biased analysis in certain scenarios by concealing missing data. In our particular database, this method would have sufficed since no cell in any table holds the value "0".

References

Geeks for Geeks (2023) *Python | Pandas DataFrame.where().* Available from: https://www.geeksforgeeks.org/python-pandas-dataframe-where/ [Accessed 16 April 2024].