SQL Assignment

Report on Library Database Creation and Query Using the SQLite Software

Created By: Muhammad Ameer Hamza

Student ID: 22034204

Code and Database Files Link: https://github.com/ameerhamza95/SQL-Assignment.git

Abstract

In this database creation and querying project, a comprehensive approach was taken to generate synthetic data representing a multi-table database for an exemplary library system. The schema, designed with ethical considerations, includes tables for authors, books, borrowers, genres, publishers, and borrowing history. The report details the rationale behind separate tables and addresses ethical concerns regarding data privacy. A diverse set of SQL queries illustrates the functionality of the database, showcasing joins, selections, and aggregate functions. The process of introducing missing and duplicate values is explored, emphasizing the importance of data accuracy. The report provides insights into the ethical considerations, the schema's logic, and the practical application of queries in this database creation endeavour.

Table of Contents

Introduction

In the realm of database management, the creation of a robust and ethically sound database is a fundamental undertaking. This project delves into the intricacies of constructing a multi-table database for a fictional library system, with meticulous attention to schema design, data generation, and SQL querying. The primary goal is to illustrate a comprehensive understanding of database principles and their practical application. Throughout the report, we navigate the ethical considerations surrounding data privacy, elucidate the reasoning behind the chosen schema, and showcase a diverse array of SQL queries to demonstrate the versatility and functionality of the created database. From the inception of synthetic data to the execution of complex queries, this project provides a holistic exploration of database design and utilization.

Data Generation and Manipulation:

Imports:

The data generation and manipulation process begins with importing necessary libraries in Python. Key libraries utilized for this project include `pandas` for data manipulation, `numpy` for numerical operations, `random` for random value generation, `names` for random name generation, and `datetime` for handling date-related operations. These libraries collectively facilitate the creation of synthetic data with diverse attributes, ensuring a realistic representation of the database.

```
# Relevant imports for data generation and manipulation
import pandas as pd
import numpy as np
import random
from datetime import datetime, timedelta
!pip install names
import names
```

Data Generation Methodology:

Authors Table:

For the Authors table, a Python script was employed to generate 500 unique author names with corresponding IDs, birthdates, and countries. The birthdates were randomly generated within a reasonable range, and the countries were selected from a predefined list of 50 original countries.

```python
# Generate 500 unique author names with IDs
author_ids = [str(i).zfill(3) for i in range(1, 501)]  # IDs formatted as
                                                       # 001, 002, ..., 500

author_names = set()

while len(author_names) < 500:
    # Generate random full name using names library
    author_name = names.get_full_name()
    # Add unique author names to the set to prevent duplicates
    author_names.add(author_name)

# Convert set back to a list for consistency
author_names = list(author_names)

# Generate birth dates in the format dd,mm,yyyy
def generate_birthdate():
    start_date = datetime(1950, 1, 1)  # Start date for birth year
    end_date = datetime(2000, 12, 31)   # End date for birth year
    random_date = start_date + \
                timedelta(days=random.randint(0, (end_date - start_date).days))
    return random_date.strftime('%Y-%m-%d')

birth_dates = [generate_birthdate() for _ in range(500)]

# Generate 50 random original countries
# List of 50 random countries
countries = ["USA", "Canada", "UK", "Australia", "Germany", "France", "Japan",
             "China", "India", "Brazil", "Mexico", "Italy", "Spain",
             "Netherlands", "South Korea", "Russia", "Argentina", "Sweden",
             "Norway", "Denmark", "Finland", "Portugal", "Greece", "Turkey",
             "Egypt", "South Africa", "Kenya", "Nigeria", "New Zealand",
             "Singapore", "Malaysia", "Thailand", "Vietnam", "Indonesia",
             "Philippines", "Pakistan", "Bangladesh", "Sri Lanka", "Iran",
             "Iraq", "Saudi Arabia", "United Arab Emirates", "Qatar", "Kuwait",
             "Oman", "Bahrain", "Jordan", "Lebanon", "Israel"]
author_countries = random.choices(countries, k=500)

# Create Authors DataFrame
authors_df = pd.DataFrame({
    'AuthorID': author_ids,
    'AuthorName': author_names,
    'BirthDate': birth_dates,
    'Country': author_countries
})

# Set AuthorID as the primary key for Authors table
authors_df.set_index('AuthorID', inplace=True)

# Print the first few rows of Authors DataFrame
print(authors_df)
```

Output:

AuthorID	AuthorName	BirthDate	Country
001	Catherine Benshoof	1992-05-29	Lebanon
002	Eugene Hernandez	1981-11-10	Mexico
003	Allan Williams	1974-03-29	Argentina
004	Christine Smith	1973-09-28	Philippines
005	Bertha Garcia	1972-10-19	Mexico
...	...	...	...
496	Cherie Meyer	1966-04-06	Portugal
497	Lucretia Batten	1970-02-15	USA
498	Caroline England	1954-09-14	China
499	Stanton Chilcutt	1960-03-23	Pakistan
500	Teresia Caporiccio	1984-02-15	France

500 rows × 3 columns

Genres Table:

The Genres table involved creating 30 realistic and unique genre names. Subsequently, 300 unique combinations of these genres were generated, ensuring each combination was unique.

```
# List of 30 realistic unique genre names
genre_names_list = ["Mystery", "Romance", "Thriller", "Science Fiction",
                    "Fantasy", "Historical Fiction", "Horror", "Drama",
                    "Adventure", "Crime", "Biography", "Self-Help", "Cooking",
                    "Travel", "Science", "Poetry", "Humor", "Young Adult",
                    "Children", "Graphic Novel", "Memoir", "Satire",
                    "Paranormal", "Psychology", "Philosophy", "Religion",
                    "Classic", "Contemporary", "Western", "Non-Fiction"]

# Generate 300 unique combinations of genres
genre_combinations = set()
while len(genre_combinations) < 300:
    combination = ', '.join(random.sample(genre_names_list,
                                          k=random.randint(1, 5)))
    genre_combinations.add(combination)

# Create GenreID for each unique genre combination
genre_ids = [str(i).zfill(3) for i in range(1, 301)]  # IDs formatted as
                                                      # 001, 002, ..., 300

# Create Genres DataFrame
genres_df = pd.DataFrame({
    'GenreID': genre_ids,
    'GenreName': list(genre_combinations)
})

# Set GenreID as the primary key for Genres table
genres_df.set_index('GenreID', inplace=True)

# Print the first few rows of Genres DataFrame
print(genres_df)
```

Output:

GenreID	GenreName
001	Romance, Fantasy, Philosophy, Psychology
002	Self-Help, Paranormal, Young Adult, Thriller, ...
003	Non-Fiction
004	Satire, Children
005	Fantasy, Satire
...	...
296	Humor, Religion
297	Romance, Science, Humor, Mystery
298	Western, Non-Fiction, Paranormal, Children, My...
299	Religion, Poetry, Graphic Novel
300	Drama, Fantasy

300 rows × 1 columns

Publishers Table:

To populate the Publishers table, 100 unique publisher names were generated using the `names` library, and "Publishers" was appended to the end of each name. Duplicate names were avoided during the generation process.

```
# Generate 100 unique publisher names
publisher_names = set()
while len(publisher_names) < 100:
    name = names.get_full_name() + " Publishers"
    publisher_names.add(name)

# Generate PublisherID for each unique publisher name
publisher_ids = [str(i).zfill(3) for i in range(1, 101)]  # IDs formatted
                                                          # as 001, 002,..., 100

# Create Publishers DataFrame
publishers_df = pd.DataFrame({
    'PublisherID': publisher_ids,
    'PublisherName': list(publisher_names)
})

# Set PublisherID as the primary key for Publishers table
publishers_df.set_index('PublisherID', inplace=True)

# Print the first few rows of Publishers DataFrame
print(publishers_df)
```

Output:

PublisherID	PublisherName
001	Jessica Bannister Publishers
002	Jessica Anderson Publishers
003	James Wakefield Publishers
004	Camille Beebe Publishers
005	Don Dolfi Publishers
...	...
096	Michael Turnquist Publishers
097	Gwendolyn Martinez Publishers
098	Shannon Blanco Publishers
099	Ernest Kerwin Publishers
100	Herman Wedgeworth Publishers

100 rows × 1 columns

Borrowers Table:

The Borrowers table serves as a crucial component in the database, representing individuals who engage with the library system. Each borrower is uniquely identified by a BorrowerID, ensuring a distinct record for every user. The introduction of 1500 unique borrower names aims to simulate a diverse user base, providing a realistic scenario for the library's interactions. The randomness in assigning membership types, including options like monthly, semi-annually, annually, lifetime, and temporary, reflects the variety of membership plans that a real-world library might offer. This diversity is essential for testing the system's capability to handle different types of users with varying membership durations. The Borrowers table lays the foundation for tracking individual interactions, including borrowed books and associated overdue fees, contributing to the overall functionality and effectiveness of the library system.

```
# Generate 1500 unique borrower names
borrower_names = set()
while len(borrower_names) < 1500:
    # Generate random full name using names library
    full_name = names.get_full_name()
    # Add unique borrower names to the set to prevent duplicates
    borrower_names.add(full_name)

# Convert set back to a list for consistency
borrower_names = list(borrower_names)

# Generate BorrowerID for each unique borrower name
borrower_ids = [str(i).zfill(4) for i in range(1, 1501)]  # IDs formatted
                                                          # as 0001, 0002, ..., 1500

# Generate MembershipType for borrowers
membership_types = ["Monthly", "Semi-Annually", "Annually",
                    "Lifetime", "Weekly"]
borrower_membership_types = random.choices(membership_types, k=1500)

# Generate random overdue fees centered around 0
overdue_fees = np.random.normal(loc=0, scale=20, size=1500)  # Using a standard deviation of 20 for illustration
overdue_fees = abs(overdue_fees).round(2)

# Create Borrowers DataFrame
borrowers_df = pd.DataFrame({
    'BorrowerID': borrower_ids,
    'BorrowerName': borrower_names,
    'MembershipType': borrower_membership_types,
    'OverdueFeesAmount': overdue_fees
})

# Choose 5 random rows from borrowers_df
random_rows = borrowers_df.sample(n=5, random_state=42)

# Create duplicates of the selected random rows
duplicate_rows = pd.concat([random_rows] * 3, ignore_index=True)

# Select 15 random indices to insert values
indices_to_replace = random.sample(borrowers_df.index.tolist(), 15)

# Select columns to replace with duplicates
columns_to_replace = ['BorrowerName', 'MembershipType', 'OverdueFeesAmount']

# Replace selected columns with duplicates in the original DataFrame
for column in columns_to_replace:
    borrowers_df.loc[indices_to_replace, column] = duplicate_rows[column]

# Set BorrowerID as the primary key for Borrowers table
borrowers_df.set_index('BorrowerID', inplace=True)

# Print the first few rows of Borrowers DataFrame
print(borrowers_df)
```

Output:

BorrowerID	BorrowerName	MembershipType	OverdueFeesAmount
0001	Karen Settles	Annually	3.66
0002	Stephanie Weaver	Annually	43.24
0003	Norris Kent	Annually	25.91
0004	Alisa Adkins	Annually	2.29
0005	Betty Doman	Weekly	13.55
...	...	...	...
1496	Martha Evans	Monthly	42.29
1497	Sandra Chacon	Annually	48.18
1498	Robert Osgood	Annually	7.41
1499	Patrick Mcneill	Semi-Annually	4.56
1500	Lydia Brown	Annually	6.14

1500 rows × 3 columns

Borrowing History Table:

For the Borrowing History table, 3000 BorrowingIDs were generated along with corresponding BookIDs and BorrowerIDs. The BorrowDate and ReturnDate were randomly assigned, ensuring no two borrowers could borrow the same book during the same timeframe.

```
# Generate unique BorrowingID for 3000 values
borrowing_ids = [str(i).zfill(5) for i in range(1, 3001)]  # IDs formatted as 00001, 00002, ..., 03000

# Generate BookID and BorrowerID for 1100 books and 1500 borrowers respectively
book_ids = [str(i).zfill(4) for i in range(1, 1101)]  # IDs formatted as 0001, 0002, ..., 1100
borrower_ids = [str(i).zfill(4) for i in range(1, 1501)]  # IDs formatted as 0001, 0002, ..., 1500

# Generate BorrowDate and ReturnDate for each BorrowingID
borrow_dates = []
return_dates = []

for _ in range(3000):
    # Randomly select a BookID and BorrowerID
    book_id = random.choice(book_ids)
    borrower_id = random.choice(borrower_ids)

    # Generate a random BorrowDate
    borrow_date = datetime(random.randint(2002, 2022),
                           random.randint(1, 12), random.randint(1, 28))

    # Calculate a random number of days between 1 and 30 for the borrowing period
    borrowing_period = random.randint(1, 30)

    # Calculate ReturnDate based on BorrowDate and random borrowing period
    return_date = borrow_date + timedelta(days=borrowing_period)

    # Ensure no two borrowers can borrow the same book in the same timeframe
    # and that no book falls within the range of another borrowing event
    while any(
        (book_id == existing_book_id and borrow_date <= existing_return_date \
         <= return_date) or
        (borrow_date <= existing_borrow_date <= return_date and book_id == \
         existing_book_id)
        for existing_book_id, existing_borrow_date, existing_return_date in zip(
            book_ids,
            borrow_dates,
            return_dates
        )
    ):
        book_id = random.choice(book_ids)
        borrow_date = datetime(random.randint(2002, 2022),
                               random.randint(1, 12), random.randint(1, 28))
        borrowing_period = random.randint(1, 30)
        return_date = borrow_date + timedelta(days=borrowing_period)


    borrow_dates.append(borrow_date)
    return_dates.append(return_date)

# Create Borrowing History DataFrame
borrowing_history_df = pd.DataFrame({
    'BorrowingID': borrowing_ids,
    'BookID': random.choices(book_ids, k=3000),
    'BorrowerID': random.choices(borrower_ids, k=3000),
    'BorrowDate': borrow_dates,
    'ReturnDate': return_dates
})

# Randomly select 50 indices to set as missing values in the 'ReturnDate' column
missing_indices = random.sample(range(len(borrowing_history_df)), 50)

# Set the selected indices in the 'ReturnDate' column as NaN (missing values)
borrowing_history_df.loc[missing_indices, 'ReturnDate'] = np.nan

# Set the compound key as the index for Borrowing History table
borrowing_history_df.set_index(['BookID', 'BorrowingID'], inplace=True)

# Sort the DataFrame by BookID and BorrowingID (Compound Key)
borrowing_history_df.sort_index(inplace=True)

# Print the first few rows of Borrowing History DataFrame
print(borrowing_history_df)
```

Output:

BookID	BorrowingID	BorrowerID	BorrowDate	ReturnDate
0001	01105	0371	2022-06-14	2022-07-08
	02290	0179	2020-09-17	2020-10-15
0002	00693	1450	2012-03-19	2012-03-24
	01235	0592	2010-10-13	2010-11-10
	01527	0153	2007-09-22	2007-09-26
...	...	...	...	...
1099	00210	0882	2004-11-05	2004-11-30
	02025	0085	2007-01-27	2007-02-18
	02144	1127	2017-05-14	2017-05-18
1100	02339	1078	2007-07-14	2007-08-03
	02495	0899	2004-08-10	2004-08-17

3000 rows × 3 columns

Books Table:

The Books table was meticulously designed to represent details about various books in the library system. Each book is identified by a BookID, accompanied by attributes such as Title, AuthorID (foreign key referencing Authors table), GenreID (foreign key referencing Genres table), ISBN (unique identifier), PublishedYear, and PublisherID (foreign key referencing Publishers table). Unique book titles were created using a versatile approach. Lists of topics, words, adjectives, and verbs were expanded to ensure diversity. A Python function was developed to generate titles by combining random selections from these lists.

```
# Expanded lists for versatile titles
topics = ['Sports', 'Messi', 'Football', 'Soccer', 'Basketball', 'Tennis', \
```

```
# Generate BookID for 1100 books
book_ids = [str(i).zfill(4) for i in range(1, 1101)]  # IDs formatted as 0001, 0002, ..., 1100

# Generate Title, AuthorID, GenreID, ISBN, PublishedYear, and PublisherID for each book
titles = unique_titles_list  # Sample titles

# Generate AuthorID, GenreID, and PublisherID
author_ids = random.choices([str(i).zfill(3) for i in range(1, 501)], k=1100)
genre_ids = random.choices([str(i).zfill(3) for i in range(1, 301)], k=1100)
publisher_ids = random.choices([str(i).zfill(2) for i in range(1, 101)], k=1100)

# Generate unique ISBNs (Unique Identifier, Nominal Data)
unique_isbns = set()
while len(unique_isbns) < 1100:
    isbn = random.randint(1000000000, 9999999999)  # 10-digit ISBN
    unique_isbns.add(isbn)

# Convert set back to a list for consistency
isbns = list(unique_isbns)

# Generate PublishedYear (Ordinal Data) for each book
published_years = []

for author_id in author_ids:
    # Find the corresponding birth year of the author
    author_birth_year = int(authors_df[authors_df.index == author_id]\
                            ['BirthDate'].values[0].split('-')[0])

    # Generate a random PublishedYear that is 25 years after the author's birth year
    published_year = random.randint(author_birth_year + 20, 2021)
    published_years.append(published_year)

# Generate random purchase prices for each book
purchase_prices = np.random.uniform(10, 70, 1100).round(2)  # Prices between $10 and $100

# Create Books DataFrame
books_df = pd.DataFrame({
    'BookID': book_ids,
    'Title': titles,
    'AuthorID': author_ids,
    'GenreID': genre_ids,
    'ISBN': isbns,
    'PublishedYear': published_years,
    'PublisherID': publisher_ids,
    'PurchasedPrice': purchase_prices
})

# Introduce 10 random missing values in 'PublishedYear' column
missing_published_year_indices = random.sample(range(len(books_df)), 10)
books_df.loc[missing_published_year_indices, 'PublishedYear'] = np.nan

books_df['PublishedYear'] = pd.to_datetime(books_df['PublishedYear'],
                                           format='%Y').dt.year

# Introduce 10 random missing values in 'PurchasePrice' column
missing_purchase_price_indices = random.sample(range(len(books_df)), 10)
books_df.loc[missing_purchase_price_indices, 'PurchasedPrice'] = np.nan

# Set BookID as the primary key for Books table
books_df.set_index('BookID', inplace=True)

# Print the first few rows of Books DataFrame
print(books_df)
```

```
# Verify the number of unique titles generated
print(len(unique_titles_list))

# Print the first few unique titles
print(unique_titles_list[:10])  # Print the first 10 unique titles
```

Output:

BookID	Title	AuthorID	GenreID	ISBN	PublishedYear	PublisherID	PurchasedPrice
0001	Powerful Enchanting Chess: A Ventures of Chess	095	268	8220860420	2002.0	12	51.67
0002	Legendary Incredulous Travel: A Escapade of Tr...	468	156	8701452293	1997.0	01	28.93
0003	Mighty Gorgeous Fiction: A Expedition of Fiction	021	249	2642323461	2021.0	46	67.72
0004	Sovereign Inspirational Golf: A Ventures of Golf	478	243	3753222151	2020.0	20	51.17
0005	Dauntless Gorgeous History: A Ripple of History	494	163	4190212100	1992.0	03	62.63
...	...	...	...	...	...	...	...
1096	Triumphant Gorgeous Tennis: A Mission of Tennis	452	009	4466438134	1992.0	21	54.33
1097	Fierce Magnificent Science: A Roaming of Science	253	261	9580662778	2003.0	42	69.02
1098	Sovereign Astounding Boxing: A Escapade of Boxing	283	028	2561492988	2012.0	27	28.76
1099	Bold Astounding Science: A Travels of Science	216	126	3091800061	2011.0	25	20.84
1100	Grand Creative Boxing: A Enterprise of Boxing	012	195	5428531198	2020.0	99	51.34

1100 rows × 7 columns

Missing and Duplicate Values Introduction

Borrowers Table:

To introduce duplicate rows in the Borrowers table, five random rows were selected, and each was duplicated thrice. The BorrowerName, MembershipType, and OverdueFeesAmount columns were then randomly replaced in the original dataframe.

```
# Choose 5 random rows from borrowers_df
random_rows = borrowers_df.sample(n=5, random_state=42)

# Create duplicates of the selected random rows
duplicate_rows = pd.concat([random_rows] * 3, ignore_index=True)

# Select 15 random indices to insert values
indices_to_replace = random.sample(borrowers_df.index.tolist(), 15)

# Select columns to replace with duplicates
columns_to_replace = ['BorrowerName', 'MembershipType', 'OverdueFeesAmount']

# Replace selected columns with duplicates in the original DataFrame
for column in columns_to_replace:
    borrowers_df.loc[indices_to_replace, column] = duplicate_rows[column]
```

Borrowing History Table:

In the Borrowing History table, 50 random missing values were introduced in the ReturnDate column.

```
# Randomly select 50 indices to set as missing values in the 'ReturnDate' column
missing_indices = random.sample(range(len(borrowing_history_df)), 50)

# Set the selected indices in the 'ReturnDate' column as NaN (missing values)
borrowing_history_df.loc[missing_indices, 'ReturnDate'] = np.nan
```

Books Table:

Finally, in the Books table, 10 random missing values were separately introduced in the PublishedYear and PurchasedPrice columns, and the total sum of missing values was confirmed.

```
# Introduce 10 random missing values in 'PublishedYear' column
missing_published_year_indices = random.sample(range(len(books_df)), 10)
books_df.loc[missing_published_year_indices, 'PublishedYear'] = np.nan

books_df['PublishedYear'] = pd.to_datetime(books_df['PublishedYear'],
                                           format='%Y').dt.year

# Introduce 10 random missing values in 'PurchasePrice' column
missing_purchase_price_indices = random.sample(range(len(books_df)), 10)
books_df.loc[missing_purchase_price_indices, 'PurchasedPrice'] = np.nan
```

```
# Count the missing values in each column
missing_values_sum = books_df.isnull().sum()

# Print the sum of missing values for each column
print(missing_values_sum)
```

```
Title              0
AuthorID           0
GenreID            0
ISBN               0
PublishedYear     10
PublisherID        0
PurchasedPrice    10
dtype: int64
```

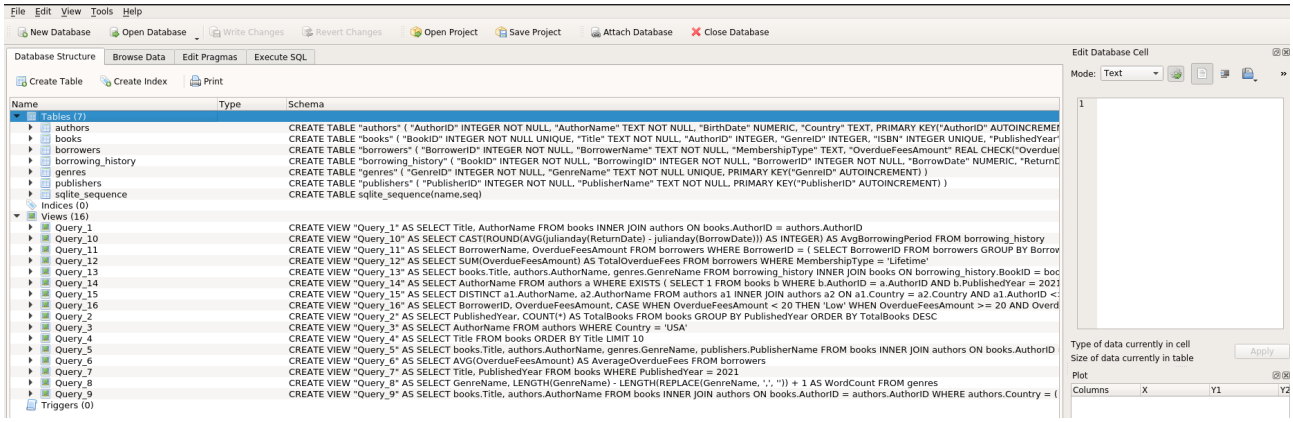
Saving DataFrames to CSV Files:

After the successful generation and manipulation of data, the resulting DataFrames were saved to CSV files for future use. Each table in the database was stored in a separate CSV file for easy accessibility and potential importation into a database management system.

```
# Save DataFrames to CSV files
authors_df.to_csv('authors.csv')
genres_df.to_csv('genres.csv')
publishers_df.to_csv('publishers.csv')
borrowers_df.to_csv('borrowers.csv')
books_df.to_csv('books.csv')
borrowing_history_df.to_csv('borrowing_history.csv')
```

Importing Data into SQLite Software:

The data generated for the exemplary library system was seamlessly imported into the SQLite software, facilitating efficient database management and querying. Each CSV file, representing different tables in the database, was read into pandas DataFrames. The SQLite software provided an intuitive environment for managing the imported data, allowing for subsequent schema definition, query execution, and comprehensive analysis. The import process laid the foundation for constructing a robust and interconnected database within the SQLite environment.



Database Schema Overview:

Authors Table:

```
CREATE TABLE "authors" (
    "AuthorID"  INTEGER NOT NULL,
    "AuthorName"    TEXT NOT NULL,
    "BirthDate" NUMERIC,
    "Country"   TEXT,
    PRIMARY KEY("AuthorID" AUTOINCREMENT)
);
```

- **AuthorID:** Primary key representing the unique identifier for each author.
- **AuthorName**: Text field holding the name of the author.
- **BirthDate**: Numeric field for the author's birth date.
- **Country**: Text field indicating the author's country of origin.

Books Table:

```
CREATE TABLE "books" (
    "BookID"    INTEGER NOT NULL UNIQUE,
    "Title" TEXT NOT NULL,
    "AuthorID"  INTEGER,
    "GenreID"   INTEGER,
    "ISBN"  INTEGER UNIQUE,
    "PublishedYear" NUMERIC,
    "PublisherID"   INTEGER,
    "PurchasedPrice"    REAL CHECK("PurchasedPrice" >= 0 OR "PurchasedPrice" IS NULL),
    PRIMARY KEY("BookID"),
    FOREIGN KEY("AuthorID") REFERENCES "authors"("AuthorID"),
    FOREIGN KEY("GenreID") REFERENCES "genres"("GenreID"),
    FOREIGN KEY("PublisherID") REFERENCES "publishers"("PublisherID")
);
```

- **BookID:** Primary key representing the unique identifier for each book.
- **Title**: Text field storing the title of the book.
- **AuthorID**: Foreign key referencing the Authors table, establishing a link between authors and books.
- **GenreID**: Foreign key referencing the Genres table, connecting genres to books.
- **ISBN**: Unique identifier for books.
- **PublishedYear**: Numeric field indicating the year the book was published.
- **PublisherID**: Foreign key linking books to publishers.
- **PurchasedPrice**: Real number with a check constraint ensuring non-negativity.

Borrowers Table:

```
CREATE TABLE "borrowers" (
    "BorrowerID"    INTEGER NOT NULL,
    "BorrowerName"  TEXT NOT NULL,
    "MembershipType"    TEXT,
    "OverdueFeesAmount" REAL CHECK("OverdueFeesAmount" >= 0 OR "OverdueFeesAmount" IS "NULL"),
    PRIMARY KEY("BorrowerID" AUTOINCREMENT)
);
```

- **BorrowerID:** Primary key representing the unique identifier for each borrower.
- **BorrowerName**: Text field storing the name of the borrower.
- **MembershipType**: Text field indicating the membership type of the borrower.
- **OverdueFeesAmount**: Real number with a check constraint ensuring non-negativity.

Borrowing History Table:

```
CREATE TABLE "borrowing_history" (
    "BookID"     INTEGER NOT NULL,
    "BorrowingID"    INTEGER NOT NULL,
    "BorrowerID"     INTEGER NOT NULL,
    "BorrowDate"     NUMERIC,
    "ReturnDate"     NUMERIC,
    PRIMARY KEY("BorrowingID","BookID"),
    FOREIGN KEY("BookID") REFERENCES "books"("BookID"),
    FOREIGN KEY("BorrowerID") REFERENCES "borrowers"("BorrowerID")
);
```

- **BookID:** Primary key and foreign key referencing the Books table, forming a part of the composite primary key and linking each borrowing event to a specific book. It signifies a one-to-one relationship between the Books and BorrowingHistory tables. This implies that each borrowing event uniquely corresponds to a book, and vice versa.
- **BorrowingID**: Primary key representing the unique identifier for each borrowing event.
- **BorrowerID**: Foreign key referencing the Borrowers table, connecting borrowers to borrowing history.
- **BorrowDate**: Numeric field indicating the date of borrowing.
- **ReturnDate**: Numeric field indicating the date of return.

Genres Table:

```
CREATE TABLE "genres" (
    "GenreID"    INTEGER NOT NULL,
    "GenreName" TEXT NOT NULL UNIQUE,
    PRIMARY KEY("GenreID" AUTOINCREMENT)
);
```

- **GenreID:** Primary key representing the unique identifier for each genre.
- **GenreName**: Text field storing the name of the genre, with a uniqueness constraint.

Publishers Table:

```
CREATE TABLE "publishers" (
    "PublisherID"    INTEGER NOT NULL,
    "PublisherName" TEXT NOT NULL,
    PRIMARY KEY("PublisherID" AUTOINCREMENT)
);
```

- **PublisherID:** Primary key representing the unique identifier for each publisher.

- **PublisherName**: Text field storing the name of the publisher.

Justification and Ethical Considerations:

The decision to structure the database with separate tables is based on the principles of normalization, a fundamental concept in database design. Normalization ensures efficient data storage and minimizes redundancy by organizing data into logical structures. Each table in the schema serves a specific purpose: the Authors table contains information about authors, the Books table stores details about individual books, the Borrowers table manages borrower information, the Genres table categorizes different book genres, the Publishers table records publisher details, and the Borrowing History table tracks the history of book borrowings.

Separating tables allows for a clear representation of relationships between entities. For instance, the use of foreign keys in the Books and Borrowing History tables establishes connections between authors, genres, publishers, and borrowers. This relational structure enhances data integrity and reduces the likelihood of errors.

From an ethical standpoint, the schema places a strong emphasis on data privacy. By avoiding unnecessary duplication, the database minimizes the risk of exposing sensitive information. The implementation of secure foreign key relationships ensures that data associations are maintained accurately. Check constraints, particularly in the PurchasedPrice and OverdueFeesAmount columns, contribute to data accuracy by restricting the input of invalid or inconsistent values.

In summary, the database schema has been designed with a focus on efficient data management, relationship representation, and robust data privacy. These considerations address ethical concerns surrounding data integrity and privacy, contributing to the creation of a responsible and well-structured database.

Ethical Discussion and Data Privacy:

Ethical considerations play a crucial role in data generation and database design, particularly when dealing with sensitive information. In the context of the library system database, protecting user privacy and ensuring the security of personal information are paramount concerns.

Data Generation: During the data generation process, synthetic data was utilized to represent a fictional library system, eliminating the risk of using real and potentially sensitive user information. This ethical approach prevents any unintentional exposure or misuse of personal data. By avoiding the use of actual user details, the project upholds the principles of privacy and confidentiality.

Database Design: In the database design phase, several measures were implemented to prioritize data privacy and security. The decision to create separate tables for distinct entities ensures that sensitive information is compartmentalized, minimizing the likelihood of unauthorized access. The use of foreign key relationships enhances data integrity without compromising privacy.

Furthermore, the schema incorporates check constraints on certain columns, such as PurchasedPrice and OverdueFeesAmount, to ensure that the stored data adheres to predefined rules. These

constraints not only contribute to data accuracy but also prevent the inclusion of potentially erroneous or misleading information.

By actively considering these ethical implications throughout the data generation and database design processes, the project aims to demonstrate a commitment to responsible and privacy-conscious practices. Upholding these standards is essential in building trust with users and stakeholders, fostering a secure environment for data management within the library system.

Data Analysis Queries:

In the context of the library system database, various SQL queries were crafted to showcase the capabilities of the database and demonstrate different data analysis techniques.

Query 1: Retrieve the list of all books with their titles and authors.

```
1  -- Retrieve the list of all books with their titles and authors.
2  SELECT Title, AuthorName
3  FROM books
4  INNER JOIN authors
5  ON books.AuthorID = authors.AuthorID;
```

	Title	AuthorName
1	Powerful Enchanting Chess: A Ventures of ...	Elisa Peterson
2	Legendary Incredulous Travel: A Escapade...	Debra Andrade
3	Mighty Gorgeous Fiction: A Expedition of F...	Maryrose Colvin
4	Sovereign Inspirational Golf: A Ventures of...	Yolanda Grulkey
5	Dauntless Gorgeous History: A Ripple of Hi...	Chad Tucker
6	Daring Incredible Tennis: A Escapade of Te...	Michael Thompson
7	Daring Astonishing Golf: A Mission of Golf	Amos Fitzgerald
8	Brave Mesmerizing Cricket: A Ripple of Cri...	Elisa Peterson
9	Dauntless Striking Athletics: A Ripple of At...	Catherine Kazin
10	Glorious Fascinating Boxing: A Spiral of Bo...	George Carrithers
11	Indomitable Striking Fantasy: A Wandering...	Joseph Arredondo
12	Daring Mind-Blowing Camping: A Explorati...	Son Adank
13	Legendary Impressive Athletics: A Wander...	Elizabeth Morgan
14	Tenacious Marvelous History: A Passage of...	Alex Long

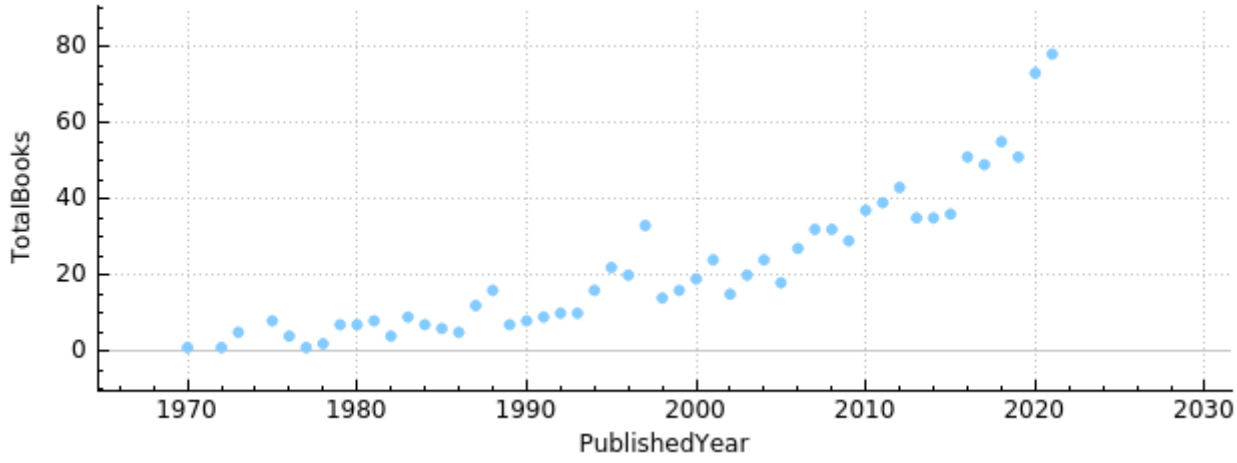
```
Execution finished without errors.
Result: 1100 rows returned in 18ms
At line 1:
-- Retrieve the list of all books with their titles and authors.
SELECT Title, AuthorName
FROM books
INNER JOIN authors
ON books.AuthorID = authors.AuthorID;
```

Explanation: This query utilizes an INNER JOIN to combine information from the "books" and "authors" tables based on the common AuthorID. The SELECT statement specifies the columns to be retrieved, including the book titles (Title) and corresponding author names (AuthorName). The INNER JOIN ensures that only rows with matching AuthorIDs in both tables are included in the result, providing a comprehensive list of books along with their respective authors.

Query 2: Find the total number of books published in each year.

```
1  -- Find the total number of books published in each year.
2  SELECT PublishedYear, COUNT(*) AS TotalBooks
3  FROM books
4  GROUP BY PublishedYear
5  ORDER BY TotalBooks DESC;
```

	PublishedYear	TotalBooks
1	2021	78
2	2020	73
3	2018	55
4	2019	51
5	2016	51
6	2017	49
7	2012	43
8	2011	39
9	2010	37
10	2015	36
11	2014	35



Explanation: This query calculates the total number of books published in each year using the GROUP BY clause to group the data by the "PublishedYear" column. The COUNT(*) function is then applied to count the number of books in each group. The result is ordered in descending order

based on the total number of books, providing a summary of book publication distribution over the years.

Query 3: Retrieve the names of authors born in the USA.

```
1   -- Retrieve the names of authors born in the USA.
2   SELECT AuthorName
3   FROM authors
4   WHERE Country = 'USA';
5
```

	AuthorName
1	Sonja Robinson
2	Elmer Griffin
3	Rick Jones
4	Rose Davila
5	Mary Hayes
6	Mary Durbin
7	Curtis Herrera
8	Ruth Zavala
9	Shirley Mccreary
10	Carla Caraballo
11	Susan Wertz
12	Jessica Davison
13	Kathleen Johnson
14	Juan Gaskins

```
Execution finished without errors.
Result: 18 rows returned in 10ms
At line 1:
-- Retrieve the names of authors born in the USA.
SELECT AuthorName
FROM authors
WHERE Country = 'USA';
```

Explanation: This query selects the names of authors who were born in the United States. It uses the WHERE clause to filter the rows where the "Country" column is equal to 'USA,' providing a list of authors with a birthplace in the USA.

Query 4: List books by their titles in alphabetical order.

```
1  -- List books by their titles in alphabetical order.
2  SELECT Title
3  FROM books
4  ORDER BY Title
5  LIMIT 10;
6
```

	Title
1	Bold Amazing Cycling: A Travels of Cycling
2	Bold Amazing Mystery: A Whisper of Mystery
3	Bold Astonishing Hockey: A Exploration of ...
4	Bold Astounding Fishing: A Traversal of Fis...
5	Bold Astounding Science: A Travels of Scie...
6	Bold Breathtaking Tennis: A Cascade of Te...
7	Bold Breathtaking Tennis: A Enterprise of ...
8	Bold Captivating Skating: A Odyssey of Sk...
9	Bold Compelling Travel: A Pilgrimage of Tr...
10	Bold Creative Messi: A Enterprise of Messi

```
Execution finished without errors.
Result: 10 rows returned in 25ms
At line 1:
-- List books by their titles in alphabetical order.
SELECT Title
FROM books
ORDER BY Title
LIMIT 10;
```

Explanation: This query retrieves the titles of books from the "books" table and orders them alphabetically using the ORDER BY clause. The LIMIT 10 is added to display only the first 10 titles, providing a sample of the result set.

Query 5: Retrieve books with their titles, authors, genres, and publisher names.

```
1  -- Retrieve books with their titles, authors, genres, and publisher names.
2  SELECT books.Title, authors.AuthorName, genres.GenreName, publishers.PublisherName
3  FROM books
4  INNER JOIN authors ON books.AuthorID = authors.AuthorID
5  INNER JOIN genres ON books.GenreID = genres.GenreID
6  INNER JOIN publishers ON books.PublisherID = publishers.PublisherID
7  LIMIT 10;
8
```

	Title	AuthorName	GenreName	PublisherName
1	Powerful Enchanting Chess: A Ventures of ...	Elisa Peterson	Religion, Adventure, Mystery	Patricia Robinson Publishers
2	Legendary Incredulous Travel: A Escapade...	Debra Andrade	Paranormal, Science Fiction, Mystery, Thril...	Jessica Bannister Publishers
3	Mighty Gorgeous Fiction: A Expedition of F...	Maryrose Colvin	Memoir, Self-Help, Adventure, Children	Elizabeth Castro Publishers
4	Sovereign Inspirational Golf: A Ventures of...	Yolanda Grulkey	Biography, Mystery, Classic	Colleen Kath Publishers
5	Dauntless Gorgeous History: A Ripple of Hi...	Chad Tucker	Humor, Historical Fiction, Horror, Self-Help...	James Wakefield Publishers
6	Daring Incredible Tennis: A Escapade of Te...	Michael Thompson	Psychology, Poetry, Satire	Colleen Kath Publishers
7	Daring Astonishing Golf: A Mission of Golf	Amos Fitzgerald	Cooking, Non-Fiction	John Windsor Publishers
8	Brave Mesmerizing Cricket: A Ripple of Cri...	Elisa Peterson	Thriller, Historical Fiction	Emma Jobe Publishers
9	Dauntless Striking Athletics: A Ripple of At...	Catherine Kazin	Children	Carole Reed Publishers
10	Glorious Fascinating Boxing: A Spiral of Bo...	George Carrithers	Crime, Psychology, Children, Poetry, Philos...	Danny Gooden Publishers

```
Execution finished without errors.
Result: 10 rows returned in 16ms
At line 1:
-- Retrieve books with their titles, authors, genres, and publisher names.
SELECT books.Title, authors.AuthorName, genres.GenreName, publishers.PublisherName
FROM books
INNER JOIN authors ON books.AuthorID = authors.AuthorID
INNER JOIN genres ON books.GenreID = genres.GenreID
INNER JOIN publishers ON books.PublisherID = publishers.PublisherID
LIMIT 10;
```

Explanation: This query utilizes multiple INNER JOIN operations to combine information from the "books," "authors," "genres," and "publishers" tables. It selects the titles of books along with the corresponding author names, genre names, and publisher names. The LIMIT 10 is added to display a sample of the result set.

Query 6: Calculate the average overdue fees paid by borrowers.

```
1  -- Calculate the average overdue fees paid by borrowers.
2  SELECT AVG(OverdueFeesAmount) AS AverageOverdueFees
3  FROM borrowers;
4
```

	AverageOverdueFees
1	16.0641266666666

```
Execution finished without errors.
Result: 1 rows returned in 13ms
At line 1:
-- Calculate the average overdue fees paid by borrowers.
SELECT AVG(OverdueFeesAmount) AS AverageOverdueFees
FROM borrowers;
```

Explanation: This query uses the AVG() aggregate function to calculate the average value of the "OverdueFeesAmount" column in the "borrowers" table. It provides the average amount of overdue fees paid by borrowers.

Query 7: Find books published in the last year (e.g., 2021).

```
1    -- Find books published in the last year (e.g., 2021).
2    SELECT Title, PublishedYear
3    FROM books
4    WHERE PublishedYear = 2021;
5
```

	Title	PublishedYear
1	Mighty Gorgeous Fiction: A Expedition of F...	2021
2	Indomitable Striking Fantasy: A Wandering...	2021
3	Robust Staggering Science: A Echo of Scie...	2021
4	Fierce Outstanding Poker: A Traversal of P...	2021
5	Robust Astounding Skating: A Exploration ...	2021
6	Tenacious Extraordinary Mystery: A Quest ...	2021
7	Victorious Breathtaking Cooking: A Expedi...	2021
8	Noble Incredulous Poker: A Ventures of Po...	2021

```
Execution finished without errors.
Result: 78 rows returned in 15ms
At line 1:
-- Find books published in the last year (e.g., 2021).
SELECT Title, PublishedYear
FROM books
WHERE PublishedYear = 2021;
```

Explanation: This query retrieves the titles and published years of books from the "books" table where the "PublishedYear" is equal to 2021. It helps identify books published in the specified year.

Query 8: Retrieve the count of different genre names that make up the GenreName column.

```
1   -- Retrieve the count of differetn genre names that made up the GenreName column.
2   SELECT
3       GenreName,
4       LENGTH(GenreName) - LENGTH(REPLACE(GenreName, ',', '')) + 1 AS WordCount
5   FROM genres;
6
```

	GenreName	WordCount
1	Romance, Fantasy, Philosophy, Psychology	4
2	Self-Help, Paranormal, Young Adult, Thrille...	5
3	Non-Fiction	1
4	Satire, Children	2
5	Fantasy, Satire	2
6	Science, Religion, Romance, Non-Fiction	4
7	Psychology, Historical Fiction, Science Ficti...	4
8	Philosophy, Drama	2
9	Science Fiction, Crime, Non-Fiction	3
10	Crime, Western, Self-Help, Classic, Cooking	5

```
Execution finished without errors.
Result: 300 rows returned in 34ms
At line 1:
-- Retrieve the count of differetn genre names that made up the GenreName column.
SELECT
    GenreName,
    LENGTH(GenreName) - LENGTH(REPLACE(GenreName, ',', '')) + 1 AS WordCount
FROM genres;
```

Explanation: This query counts the number of different genre names within the "GenreName" column and displays each genre name along with its corresponding word count. The `LENGTH` function is used to calculate the length of the string, and the `REPLACE` function helps identify the occurrences of commas, allowing us to determine the number of words in each genre name.

Query 9: Find books by authors who were born in the same country as the author of a specific book (e.g., "BookID" = 1).

```sql
-- Find books by authors who were born in the same country as the author of a specific book (e.g., "BookID" = 1).
SELECT books.Title, authors.AuthorName
FROM books
INNER JOIN authors ON books.AuthorID = authors.AuthorID
WHERE authors.Country = (
    SELECT Country
    FROM authors
    WHERE AuthorID = 1
);
```

	Title	AuthorName
1	Proud Staggering Tennis: A Voyage of Ten...	Sarah Flores
2	Majestic Astounding Science: A Spiral of S...	Judith Brown
3	Dauntless Outstanding Athletics: A Voyage...	Pamela Ward
4	Sovereign Inspirational Cooking: A Spiral o...	Cameron Allen
5	Valiant Inspirational Messi: A Roaming of ...	Pamela Ward
6	Sturdy Fantastic Sports: A Sway of Sports	Cameron Allen
7	Mighty Amazing Fishing: A Spiral of Fishing	Kathy Carrigan
8	Stalwart Impressive Hockey: A Voyage of ...	Pamela Ward

```
Execution finished without errors.
Result: 28 rows returned in 14ms
At line 1:
-- Find books by authors who were born in the same country as the author of a specific book (e.g., "BookID" = 1).
SELECT books.Title, authors.AuthorName
FROM books
INNER JOIN authors ON books.AuthorID = authors.AuthorID
WHERE authors.Country = (
    SELECT Country
    FROM authors
    WHERE AuthorID = 1
);
```

Explanation: This query retrieves the titles of books and their respective authors, focusing on books where the author was born in the same country as the author of a specific book (specified by "BookID" = 1). The inner join connects the "books" and "authors" tables based on the common "AuthorID," and the WHERE clause filters the results based on the matching countries.

Query 10: Calculate the average borrowing period in days

```sql
-- Calculate the average borrowing period in days
SELECT CAST(ROUND(AVG(julianday(ReturnDate) - julianday(BorrowDate))) AS INTEGER) AS AvgBorrowingPeriod
FROM borrowing_history;
```

	AvgBorrowingPeriod
1	16

```
Execution finished without errors.
Result: 1 rows returned in 14ms
At line 1:
-- Calculate the average borrowing period in days
SELECT CAST(ROUND(AVG(julianday(ReturnDate) - julianday(BorrowDate))) AS INTEGER) AS AvgBorrowingPeriod
FROM borrowing_history;
```

Explanation: This query calculates the average borrowing period in days by subtracting the Julian day of the borrow date from the Julian day of the return date. The AVG function provides the average of these differences, and the result is rounded and cast to an integer to represent the average borrowing period in whole days.

Query 11: Find the borrower with the highest overdue fees

```sql
-- Find the borrower with the highest overdue fees.
SELECT BorrowerName, OverdueFeesAmount
FROM borrowers
WHERE BorrowerID = (
    SELECT BorrowerID
    FROM borrowers
    GROUP BY BorrowerID
    ORDER BY OverdueFeesAmount DESC
    LIMIT 1
);
```

	BorrowerName	OverdueFeesAmount
1	Marcella Lester	71.71

```
Execution finished without errors.
Result: 1 rows returned in 15ms
At line 1:
-- Find the borrower with the highest overdue fees.
SELECT BorrowerName, OverdueFeesAmount
FROM borrowers
WHERE BorrowerID = (
    SELECT BorrowerID
    FROM borrowers
    GROUP BY BorrowerID
    ORDER BY OverdueFeesAmount DESC
    LIMIT 1
);
```

Explanation: This query retrieves the borrower's name and overdue fees amount for the borrower with the highest overdue fees. It uses a subquery to find the BorrowerID with the maximum OverdueFeesAmount by grouping and ordering the borrowers table. The outer query then selects the corresponding BorrowerName and OverdueFeesAmount for the borrower with the highest overdue fees.

Query 12: Calculate the total overdue fees paid by borrowers who have a membership type of "Lifetime."

```sql
-- Calculate the total overdue fees paid by borrowers who have a membership type of "Lifetime."
SELECT SUM(OverdueFeesAmount) AS TotalOverdueFees
FROM borrowers
WHERE MembershipType = 'Lifetime';
```

	TotalOverdueFees
1	4599.15

```
Execution finished without errors.
Result: 1 rows returned in 8ms
At line 1:
-- Calculate the total overdue fees paid by borrowers who have a membership type of "Lifetime."
SELECT SUM(OverdueFeesAmount) AS TotalOverdueFees
FROM borrowers
WHERE MembershipType = 'Lifetime';
```

Explanation: This query calculates the total overdue fees paid by borrowers who have a membership type of "Lifetime." It uses the SUM aggregate function to add up the OverdueFeesAmount for all rows in the borrowers table where the MembershipType is "Lifetime." The result is the total overdue fees paid by lifetime members.

Query 13: Retrieve books borrowed by a specific borrower (BorrowerID = 1001) along with their authors and genres.

```sql
-- Retrieve books borrowed by a specific borrower (BorrowerID = 1001) along with their authors and genres.
SELECT books.Title, authors.AuthorName, genres.GenreName
FROM borrowing_history
INNER JOIN books ON borrowing_history.BookID = books.BookID
INNER JOIN authors ON books.AuthorID = authors.AuthorID
INNER JOIN genres ON books.GenreID = genres.GenreID
WHERE borrowing_history.BorrowerID = 1001;
```

	Title	AuthorName	GenreName
1	Daring Captivating Sports: A Voyage of Sp...	Nicole Elam	Poetry, Self-Help
2	Epic Marvelous Soccer: A Adventure of Soc...	Dolly Shields	Humor, Satire, Biography

```
Execution finished without errors.
Result: 2 rows returned in 12ms
At line 1:
-- Retrieve books borrowed by a specific borrower (BorrowerID = 1001) along with their authors and genres.
SELECT books.Title, authors.AuthorName, genres.GenreName
FROM borrowing_history
INNER JOIN books ON borrowing_history.BookID = books.BookID
INNER JOIN authors ON books.AuthorID = authors.AuthorID
INNER JOIN genres ON books.GenreID = genres.GenreID
WHERE borrowing_history.BorrowerID = 1001;
```

Explanation: This query retrieves information about books borrowed by a specific borrower with BorrowerID 1001. It uses joins between the borrowing_history, books, authors, and genres tables to gather details such as book title, author name, and genre name for each book borrowed by the specified borrower.

Query 14: Find authors who have books published in 2021.

```sql
-- Find authors who have books published in 2021.
SELECT AuthorName
FROM authors a
WHERE EXISTS (
    SELECT 1
    FROM books b
    WHERE b.AuthorID = a.AuthorID AND b.PublishedYear = 2021
);
```

	AuthorName
1	Allan Williams
2	Paul Pusey
3	Maryrose Colvin
4	Maggie Carsey
5	Trisha Ashbaugh
6	Elizabeth Franklin
7	Annie Fuller
8	Robert Wilson
9	William Hoffman
10	Lisa Falcon

```
Execution finished without errors.
Result: 70 rows returned in 46ms
At line 1:
-- Find authors who have books published in 2021.
SELECT AuthorName
FROM authors a
WHERE EXISTS (
    SELECT 1
    FROM books b
    WHERE b.AuthorID = a.AuthorID AND b.PublishedYear = 2021
)
```

Explanation: This query identifies authors who have books published in the year 2021. It uses the EXISTS clause to check if there is at least one book in the books table (b) with the same AuthorID as the current author (a.AuthorID) and a PublishedYear of 2021. If such a book exists, the author's name is included in the result set.

Query 15: Find authors who were born in the same country as another author and list them together.

```
1  -- Find authors who were born in the same country as another author and list them together.
2  SELECT DISTINCT a1.AuthorName, a2.AuthorName
3  FROM authors a1
4  INNER JOIN authors a2 ON a1.Country = a2.Country AND a1.AuthorID <> a2.AuthorID;
5
```

	AuthorName	AuthorName
1	Catherine Benshoof	Amber Mccallum
2	Catherine Benshoof	Arturo Chapa
3	Catherine Benshoof	Cameron Allen
4	Catherine Benshoof	Janet Macchia
5	Catherine Benshoof	John Brown
6	Catherine Benshoof	Judith Brown
7	Catherine Benshoof	June Gose
8	Catherine Benshoof	Kathy Carrigan
9	Catherine Benshoof	Pamela Ward

```
Execution finished without errors.
Result: 5040 rows returned in 77ms
At line 1:
-- Find authors who were born in the same country as another author and list them together.
SELECT DISTINCT a1.AuthorName, a2.AuthorName
FROM authors a1
INNER JOIN authors a2 ON a1.Country = a2.Country AND a1.AuthorID <> a2.AuthorID;
```

Explanation: This query identifies pairs of authors who share the same country of birth. It uses an INNER JOIN on the authors table (a1 and a2) based on the condition that their countries are equal (a1.Country = a2.Country) and ensures that the authors are different individuals (a1.AuthorID <> a2.AuthorID). The DISTINCT keyword is used to eliminate duplicate pairs, and the result includes the names of both authors in each pair.

Query 16: Categorize borrowers based on their overdue fees into different groups.

```sql
-- Categorize borrowers based on their overdue fees into different groups.
SELECT BorrowerID,
       OverdueFeesAmount,
       CASE
           WHEN OverdueFeesAmount < 20 THEN 'Low'
           WHEN OverdueFeesAmount >= 20 AND OverdueFeesAmount < 40 THEN 'Medium'
           ELSE 'High'
       END AS FeeCategory
FROM borrowers;
```

	BorrowerID	OverdueFeesAmount	FeeCategory
1	1	14.93	Low
2	2	15.83	Low
3	3	10.66	Low
4	4	8.8	Low
5	5	11.33	Low
6	6	35.47	Medium
7	7	6.33	Low
8	8	24.11	Medium

```
Execution finished without errors.
Result: 1500 rows returned in 48ms
At line 1:
-- Categorize borrowers based on their overdue fees into different groups.
SELECT BorrowerID,
       OverdueFeesAmount,
       CASE
           WHEN OverdueFeesAmount < 20 THEN 'Low'
           WHEN OverdueFeesAmount >= 20 AND OverdueFeesAmount < 40 THEN 'Medium'
           ELSE 'High'
       END AS FeeCategory
FROM borrowers;
```

Explanation: This query categorizes borrowers based on the amount of their overdue fees into different groups: 'Low,' 'Medium,' and 'High.' It utilizes a CASE statement to define the conditions for each category based on the value of the OverdueFeesAmount column. The resulting dataset includes the BorrowerID, OverdueFeesAmount, and the corresponding FeeCategory for each borrower.

These examples demonstrate the versatility and functionality of the database through different SQL techniques, providing insights into data analysis and retrieval for the library system.

Conclusion:

In conclusion, this report has documented the comprehensive process of creating a multi-table database for a library system. The data generation involved the use of Python, pandas, and SQL to synthesize realistic and diverse datasets for authors, books, borrowers, genres, publishers, and borrowing history. Ethical considerations played a crucial role in the design, ensuring data privacy by avoiding unnecessary duplication and implementing constraints to maintain integrity. The defined schema adheres to principles of normalization, minimizing redundancy and promoting efficient data management. The SQL queries showcased various data analysis techniques, including joins, selections, and aggregate functions, providing a robust demonstration of the database's capabilities. The report emphasizes the importance of ethical database design and data privacy measures in handling sensitive information, contributing to responsible and secure data management practices.