
# Final Project - Music Label Data Base System

**IBT Department**

*Profesor:*

*Mehdi Pirahandeh*

Students:

| | |
|---|---|
| Vu Nguyen Bao Ngoc | ID:12225193 |
| Thiri Moe Htet | ID:12214682 |
| Chaw Thiri San | ID:12225272 |
| Luis Felipe Depardon Jasso | ID:12235233 |

**Date:**

2022/12/03

# Music Label Database System
# Introduction

Nowadays music have a big impact in humans daily live thats why our team is interested in developing a database system for a music company to introduce ourselves in this industry.

The Music Label Project Database System is a carefully designed solution to meet the changing needs of a modern music label. In a time when technology is changing how we do things and people's preferences are always evolving, managing the information related to the music industry is crucial for the success and lasting presence of a company.

Having an effective database system is really important for a music label. The music industry involves many elements like artists, albums, songs, contracts, events, and sales, all interconnected and dependent on each other. A strong database system helps make operations smoother, ensures the information is correct, and aids in making important decisions.

During this project, we'll be creating different queries that we've learned about in class. These queries represent different situations the company might face in its daily operations. These queries will help us get important information, making the company's work easier.

In our music label simulation, the key player is the artist—be it a band, singer, or musician, each treated as an 'artist' in our system. Artists create many songs and albums. So, albums are connected to a specific artist, and songs are tied to both a particular album and artist.

On the business side, we're keen on tracking album sales for revenue statistics. We use a 'Sale' table linked to a specific album to keep tabs on how albums are sold.

Beyond music, our database covers events and contracts related to artists, both linked to our main 'Artist' table. Events include things like concerts, and contracts capture important details about an artist's commitments.

In resume, our database pursues to commit the following:
-Track artist contracts.
-Track the albums and singles an artist released
-Track the sales of an album.
-Track the events of an artist.

# I. Tables

## -Artist

- ArtistID (Primary Key)
  - ArtistName
  - StageName
  - Role
  - DateOfBirth
  - Country
  - PhoneNum
  - Mail

## -Albums

- AlbumID (Primary Key)
  - ArtistID (Foreign Key referencing Artist table)
  - Title
  - ReleaseDate
  - Genre
  - TotalTracks
  - Duration

## -Songs

- SongID (Primary Key)
  - SongTitle
  - Duration
  - ReleaseDate
  - Genre
  - AlbumID (Foreign Key referencing Album table)
  - ArtistID (Foreign Key referencing Artist table)
  - ProducerID (Foreign Key referencing Artist table)

## -Events

- EventID (Primary Key)
  - EventType (concert, release party, etc.)
  - EventDate
  - Location
  - ArtistID (Foreign Key referencing Artist table)

## -Contracts

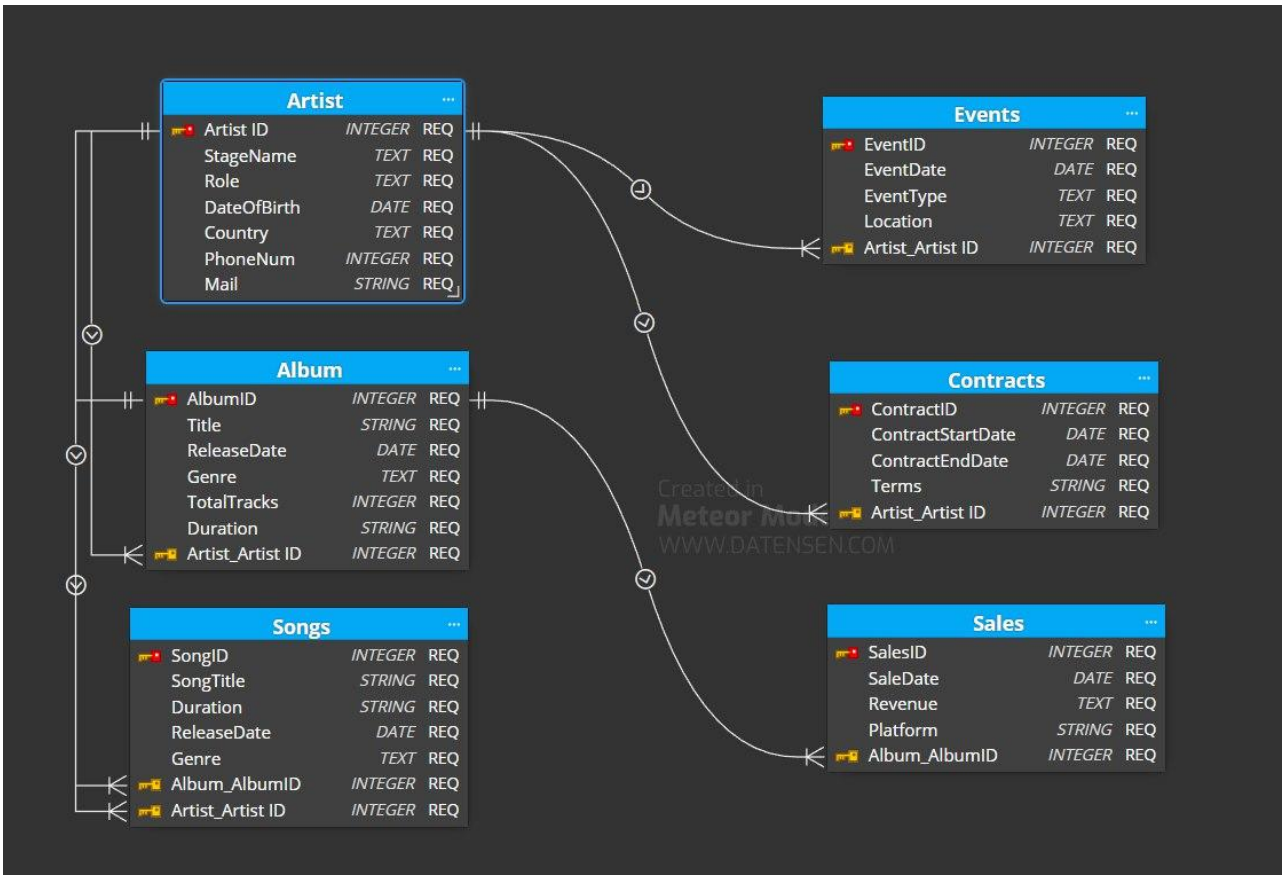
- ContractID (Primary Key)
  - ArtistID (Foreign Key referencing Artist table)
  - ContractStartDate
  - ContractEndDate
  - Terms (details of the contract)

## -Sales

- SaleID (Primary Key)
  - AlbumID (Foreign Key referencing Album table)
  - SaleDate
  - Revenue
  - Platform

# ERD Diagram

Vu Nguyen Bao Ngoc - 12225193




**Structural Overview:**
This ERD showcases the relationships between six entities in our database: Artists, Albums, Songs, Events, Contracts, and Sales, and also associations and dependencies.

**Data Integrity:**
Throughout the design process, data integrity is ensured by using appropriate data types, constraints, and avoiding redundancy, fostering a more efficient and maintainable database structure.

**Entity Attributes and Constraints:**

- **Artist:**
  - Artist ID (Integer, Primary Key, Not Null)
  - Stage Name, Role, Date of Birth, Country (Text, Not Null)

    - Phone Number (Integer, Not Null)
    - Email (String, Not Null)
- **Album:**
    - Album ID (Integer, Primary Key, Not Null)
    - Title, Release Date, Genre (String/Text, Not Null)
    - Total Tracks (Integer, Not Null)
    - Duration (String, Not Null)
    - Artist ID (Integer, Not Null)
- **Songs:**
    - Song ID (Integer, Primary Key, Not Null)
    - Song Title, Duration (String, Not Null)
    - Release Date, Genre (String/Text, Not Null)
    - Album ID, Artist ID (Integer, Not Null)
- **Events:**
    - Event ID (Integer, Primary Key, Not Null)
    - Event Date, Type, Location (String/Text, Not Null)
    - Artist ID (Integer, Not Null)
- **Contracts:**
    - Contract ID (Integer, Primary Key, Not Null)
    - Contract Start Date, End Date, Terms (String, Not Null)
    - Artist ID (Integer, Not Null)
- **Sales:**
    - Sales ID (Integer, Primary Key, Not Null)
    - Sale Date, Revenue, Platform (String/Text, Not Null)
    - Album ID (Integer, Not Null)

# II. Data Insertion

# Part 1

Author : Chaw Thiri San
ID : 12225272

## Objective:

This script aims to generate and insert fake data into three tables - Artist, Album, and Song - within an SQLite database named "Music-Label.db." This simulated data is intended for later testing using intermediate and advanced SQLs.

## Preparation

To insert varied data into the tables, the primary intention was to use the “faker” library to get randomized results. However, due to the difficulties in the analysis of data in later queries, the “random” module was decided to use instead of the “faker”.

## Code Explanation:

- Firstly, a connection is established to the SQLite database using the *SQLite3 module*. A cursor is created to execute SQL commands.
- A loop is used to insert 20 rows of fake data into the Artist table. The data includes *11 columns: ArtistID, ArtistName, Role, DateOfBirth, Country, Email, Phone, and 4 timestamp fields*. The values are generated using a combination of iteration variables and random functions.
- The other tables are also filled using a similar manner.
- * For the duration column, to get as close as possible to the real-world data, the length of the song is randomized between 120 to 300s.
- * For the data format, to get a fixed format, we use 2 digits for every date. So, in the case of 1st to 9th days, 0x format has been applied.
- After inserting data into all three tables, the changes are committed to the database using the conn.commit() statement.
- The **Pandas library** reads data from each table into DataFrames (artist_df, album_df, and song_df). The content of each table is then printed in a tabular format.
- Finally, the script closes the database connection.
- P.S. Please reference the collab file for the actual codes

# Part 2

**Thiri Moe Htet (12214682)**

## Data Insertion Part 2: Events, Contracts, and Sales Tables

This script establishes a connection with a SQLite database, adds fictitious data to the "Event," "Contract," and "Sale" tables, prints the data, and then ends the database connection. The ArtistIDs are chosen from entries that already exist in the 'Artist' table, and the fake data contains random values for a number of columns.

### Importing Necessary Modules

```
import sqlite3

import random
```

'sqlite3': This module offers a disk-based database that is lightweight and doesn't require a separate server process. It also enables users to access the database using a SQL query language variation that isn't commonly used.

'random': There are functions in this module for creating random numbers.

### Connecting to the SQLite Database

```
conn = sqlite3.connect('Music-Label.db')

cursor = conn.cursor()
```

'sqlite3.connect('Music-Label.db')': 'Music-Label.db' is the SQLite database file that is connected to by this line.

'conn.cursor()': A cursor object is created by this line to execute SQL commands.

### Fetching Existing ArtistIDs from the Artist Table

```
# Get existing ArtistIDs from the Artist table

cursor.execute('SELECT ArtistID FROM Artist')

artist_ids = [row[0] for row in cursor.fetchall()]
```

'cursor.execute('SELECT ArtistID FROM Artist')': An SQL query is run on this line in order to select every ArtistID from the 'Artist' table.

'cursor.fetchall()': By doing this, the whole result set's rows are retrieved.

'artist_ids = [row[0] for row in cursor.fetchall()]': The ArtistIDs retrieved from the 'Artist' table are created into a list (artist_ids) by this line.

### Inserting Data into Event, Contract, & Sale Tables

```
# Insert 20 rows of data into the Event table
for i in range(20):
    cursor.execute('''
        INSERT INTO Event (EventID, ArtistID, EventType, EventDate,
Location, CreatedOn, CreatedBy, ChangedOn, ChangedBy)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
    ''', (
        i + 1,
            random.choice(artist_ids),   # Randomly select an existing
ArtistID
        f"EventType{i}",
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        f"Location{i}",
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100),
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100)
    ))


# Insert 20 rows of data into the Contract table
for i in range(20):
    cursor.execute('''
       INSERT INTO Contract (ContractID, ArtistID, ContractStartDate,
ContractEndDate, Terms, CreatedOn, CreatedBy, ChangedOn, ChangedBy)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
    ''', (
        i + 1,
            random.choice(artist_ids),   # Randomly select an existing
ArtistID
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        f"{random.randint(2000, 2024)}-12-{i+1:02d}",
```

```
        f"Terms{i}",
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100),
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100)
    ))

# Insert 20 rows of data into the Sale table
for i in range(20):
    cursor.execute('''
      INSERT INTO Sale (SaleID, AlbumID, SaleDate, Revenue, Platform,
CreatedOn, CreatedBy, ChangedOn, ChangedBy)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
    ''', (
        i + 1,
          random.randint(1, 20),   # Assuming AlbumID is present in the
Album table
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.uniform(100, 1000000),
        f"Platform{i}",
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100),
        f"{random.randint(2000, 2024)}-01-{i+1:02d}",
        random.randint(1, 100)
    ))
```

Each table has 20 rows of fictitious data added by these loops.

‘random.choice(artist_ids)’ in Event & Contract Tables: chooses an existing ArtistID at random from the list.

‘VALUES(?,?,?,…)’: matches the SQL query placeholders, and the resulting tuple gives values that need to be inserted.

### Committing Changes & Printing Data

```
# Commit changes
conn.commit()


# Print all data from the Event table
cursor.execute('SELECT * FROM Event')
print("\nData from Event table:")
for row in cursor.fetchall():
    print(row)


# Print all data from the Contract table
cursor.execute('SELECT * FROM Contract')
print("\nData from Contract table:")
for row in cursor.fetchall():
    print(row)


# Print all data from the Sale table
cursor.execute('SELECT * FROM Sale')
print("\nData from Sale table:")
for row in cursor.fetchall():
    print(row)
```

Run a SELECT query, then output the data that was obtained. The database modifications are committed.

### Closing the Database Connection

```
# Close the connection
conn.close()
```

The SQLite database connection is terminated by this line.

# III. Queries

**Thiri Moe Htet (12214682)**

**Query to Calculate Total Activities & Revenues generated by Each Artist**

The 'Music-Label.db' SQLite database uses a variety of advanced SQL capabilities, including subqueries, window functions, group by clause, JOIN operations, Common Table Expressions (CTE), and GROUP BY clauses to run a complicated SQL query. Overall, this script opens a connection to a SQLite database, runs an intricate SQL query across several tables, and uses Pandas to show the results.

### Import necessary libraries

```
# Import necessary libraries

import sqlite3

import pandas as pd
```

A module to communicate with SQLite databases as sqlite3.

Pandas is a data manipulation and analysis package.

### Define a function to execute a SQL query and display the results using Pandas

```
# Define a function to execute a SQL query and display the results
using Pandas

def execute_sql_query():

    # Connect to the SQLite database named 'Music-Label.db'

    conn = sqlite3.connect('Music-Label.db')



    # Create a cursor object to execute SQL queries

    cursor = conn.cursor()
```

### Define the SQL query using Common Table Expressions (CTE), complex joins, window functions, and intermediate SQL operations

```
 # Define the SQL query using Common Table Expressions (CTE), complex
joins, window functions,

    # and intermediate SQL operations (joins, group by, having,
subqueries, aggregate functions)
```

```
    sql_query = '''

    -- Using Common Table Expressions (CTE) for readability and
simplifying the query structure



    -- CTE: RankedSales - Window function to rank sales within each
album based on revenue

    WITH RankedSales AS (

        SELECT

            SaleID,

            AlbumID,

            Revenue,

            ROW_NUMBER() OVER (PARTITION BY AlbumID ORDER BY Revenue
DESC) AS SaleRank

        FROM

            Sale

    ),



    -- CTE: TopSales - Selecting only the top sale for each album

    TopSales AS (

        SELECT

            SaleID,

            AlbumID,

            Revenue

        FROM

            RankedSales

        WHERE

            SaleRank = 1

    )



    -- Main Query combining multiple tables and advanced SQL features

    SELECT
```

```
        A.ArtistID,
        A.ArtistName,
        COUNT(DISTINCT AL.AlbumID) AS TotalAlbums,
        COUNT(DISTINCT S.SongID) AS TotalSongs,
        COUNT(DISTINCT E.EventID) AS TotalEvents,
        COUNT(DISTINCT C.ContractID) AS TotalContracts,
        AVG(SL.Revenue) AS AvgRevenuePerSale,
        MAX(TS.Revenue) AS MaxRevenueFromSingleSale
    FROM
        Artist A
    LEFT JOIN
        Album AL ON A.ArtistID = AL.ArtistID
    LEFT JOIN
        Song S ON A.ArtistID = S.ArtistID
    LEFT JOIN
        Event E ON A.ArtistID = E.ArtistID
    LEFT JOIN
        Contract C ON A.ArtistID = C.ArtistID
    LEFT JOIN (
        -- Subquery to find the average revenue per sale
        SELECT
            AL.AlbumID,
            AVG(SL.Revenue) AS Revenue
        FROM
            Album AL
        LEFT JOIN
            Sale SL ON AL.AlbumID = SL.AlbumID
        GROUP BY
            AL.AlbumID
```

```
    ) SL ON AL.AlbumID = SL.AlbumID
    LEFT JOIN TopSales TS ON AL.AlbumID = TS.AlbumID
    GROUP BY
        A.ArtistID, A.ArtistName
    HAVING
        TotalAlbums > 0
    ORDER BY
        AvgRevenuePerSale DESC;
    '''
```

**Execute the SQL query using the cursor**

```
    # Execute the SQL query using the cursor
    cursor.execute(sql_query)
```

**Fetch all results from the executed query into a Pandas DataFrame**

```
    # Fetch all results from the executed query into a Pandas DataFrame
    df = pd.read_sql_query(sql_query, conn)
```

**Display the Pandas DataFrame**

```
    # Display the Pandas DataFrame
    print(df)
```

**Close the connection to the database**

```
    # Close the connection to the database
    conn.close()
```

**Call the function to execute the SQL query and display the results**

```
# Call the function to execute the SQL query and display the results
execute_sql_query()
```

## Query to Filter Songs

Using dynamic SQL construction, this script enables you to filter songs from a music database according to dynamic constraints like artist ID, genre, and minimum duration.

### Importing necessary libraries

```
import sqlite3

import pandas as pd
```

sqlite3: A library for working with SQLite databases.

pandas: An analysis and manipulation library for data.

### Defining the function get_filtered_songs

```
def get_filtered_songs(artist_name=None, genre=None,
min_duration=None):
```

This function filters songs according to artist_id, genre, and min_duration, which are optional arguments.

### Constructing the base SQL query

```
# Construct the base SQL query with dynamic conditions using CASE
statements

    sql_query = '''

    SELECT

        S.SongID,

        S.SongTitle,

        S.Duration,

        S.ReleaseDate,

        S.Genre,

        A.ArtistName

    FROM

        Song S
```

```
    LEFT JOIN
        Artist A ON S.ArtistID = A.ArtistID
    WHERE
        1 = 1
    '''
```

creates the base SQL query by joining the Song and Artist tables on the left using the chosen fields.

**Adding dynamic conditions using CASE statements**

```
    # Add dynamic conditions based on user inputs using CASE statements
    if artist_name:
        sql_query += f" AND A.ArtistName = '{artist_name}'"

    if genre:
        sql_query += f" AND S.Genre = '{genre}'"

    if min_duration:
        sql_query += f" AND S.Duration >= {min_duration}"
```

depending on the given arguments, appends conditions to the SQL query.

**Function calls to test functionality**

```
# Call the function with different parameters to test its functionality
# Retrieve all songs
get_filtered_songs()

# Retrieve songs by a specific artist
get_filtered_songs(artist_name='Artist8')
```

```
# Retrieve songs of a specific genre

get_filtered_songs(genre='Genre1')



# Retrieve songs with a minimum duration of 200 seconds

get_filtered_songs(min_duration=200)
```

To demonstrate the capabilities of the function, call it with various functions.

**Luis Felipe Depardon Jasso (12235233)**

**Intermediate SQL**

- Classifying songs by duration.

```
SELECT SongTitle,
CASE
WHEN Duration < 120 THEN 'Short'
WHEN Duration >= 120 AND Duration <= 180 THEN 'Medium'
ELSE 'Long'
END AS DurationCategory
FROM Song;
```

- Select the column SongTitle give us the song name.
- Case clause helps us add the 3 different conditions we will use to classify the songs between Short, Medium and Long.
- From selects the song table.
- We aim to classify the songs by range.

- Total number of songs and average duration of each genre.

```
SELECT Genre, COUNT(SongID) AS TotalSongs, AVG(Duration) AS AvgDuration
FROM Song
GROUP BY Genre;
```

- Select the Genre column and apply an aggregate function to duration and song and at the same time renaming them to TotalSongs and AvgDuration respectively.
- Select the song table
- Group By the Genre.
- The GROUP BY clause in this query enables the use of aggregate functions by grouping rows based on the common values in the Genre column. This allows the application of aggregate functions such as counting the total number of songs (TotalSongs) and calculating the average duration (AvgDuration) within each distinct genre.
- We aim to retrieve song data statistics.

- Genres with an average duration above 120 sec.

```
SELECT Genre, AVG(Duration) AS AvgAlbumDuration
FROM Album
GROUP BY Genre
HAVING AvgAlbumDuration > 120;
```

- Select Genre column and apply an aggregated function to Duration at the same time rename it as AvgAlbumDuration.
- From the album table.
- Grouped the data we are going to use the aggregate function by the Genre.
- Using the Having clause allows us to apply a condition to the aggregate function in this case be more than 120.
- we aim to retrieve information.

**Advanced SQL**

- Rank genres based on their revenue generated by the album.

```
SELECT Genre, SUM(Revenue) AS TotalRevenue,
RANK() OVER (ORDER BY SUM(Revenue) DESC) AS GenreRank
FROM Album
JOIN Sale ON Album.AlbumID = Sale.AlbumID
GROUP BY Genre;
```

- The SQL query selects the Genre column and applies the aggregate function SUM to the Revenue column. Additionally
- the RANK() window function is used to assign a rank to each genre based on the total revenue, with higher sales ranked at the top.
- Select the album table where data is retrieved.
- join the tables so the sale data can be retrieved.
- group the data by genre so the sum aggregated function works.
- With this query we aim to rank the information.

- Retrieve the information about album artist and songs filtered by usa country.

```
SELECT Artist.ArtistName, Album.AlbumTitle, Song.SongTitle, Artist.Country
FROM Artist
JOIN Album ON Artist.ArtistID = Album.ArtistID
JOIN Song ON Album.AlbumID = Song.AlbumID
WHERE Artist.Country = 'Country0';
```

- Select the artistName, AlbumTitle, SongTitle and Country.
- We will only use the Artist table and we retrieve the other data performing inner joins.
- First join performed with the artist table with the album table by the ArtistID in both tables.
- Second join between song and album tables by AlbumID in both tables.

- Using Where to state the condition in which the country have to be equal to the one we want to retrieve data.
- With this we aim to retrieve data with a condition.

- find albums released after a certain date.

```
WITH RecentAlbums AS (
SELECT AlbumTitle, ReleaseDate
FROM Album
WHERE ReleaseDate > '2022-01-01'
)
SELECT *
FROM RecentAlbums;
```

- In this query, a Common Table Expression (CTE) named RecentAlbums is created to store albums with a release date after '2022-01-01'.
- The query inside the CTE selects the AlbumTitle and ReleaseDate from the Album table and lastly adds the date condition.
- The main query then selects all columns from the RecentAlbums CTE.

**Vu Nguyen Bao Ngoc (12225193)**

**Intermediate SQL**

1) **Contracts ending in 2020**

```
# 1. Contracts ending in 2020
cursor.execute('''
    SELECT * FROM Contract
    WHERE ContractEndDate BETWEEN '2020-01-01' AND '2020-12-31'
''')
contracts_ending_2020 = cursor.fetchall()
print("Contracts ending in 2020:")
print(contracts_ending_2020)
```

- **SELECT:** Retrieve columns from the Contract table.
- **FROM:** Specify the table to retrieve data.
- **WHERE:** Filter rows based on a condition
- **BETWEEN:** Checks if a value lies within a specified range
- **FETCHALL:** retrieve all the remaining rows of a query result as a list of tuples

### 2) Contracts starting in 2024

```
# 2. Contracts starting in 2024
cursor.execute('''
    SELECT * FROM Contract
    WHERE ContractStartDate BETWEEN '2024-01-01' AND '2024-12-31'
''')
contracts_starting_2024 = cursor.fetchall()
print("\nContracts starting in 2024:")
print(contracts_starting_2024)
```

- **SELECT:** Retrieve columns from the Contract table.
- **FROM:** Specify the table to retrieve data.
- **WHERE:** Filter rows based on a condition
- **BETWEEN:** Checks if a value lies within a specified range
- **FETCHALL:** retrieve all the remaining rows of a query result as a list of tuples

### 3) Least sold album per artist

```
# 3. Least sold album per artist
cursor.execute('''
    SELECT ArtistID, MIN(SaleCount) AS MinSales
    FROM (
        SELECT Album.ArtistID, Album.AlbumID, COUNT(Sale.AlbumID) AS SaleCount
        FROM Album
        LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
        GROUP BY Album.AlbumID
    ) AS AlbumSales
    GROUP BY ArtistID
''')
least_sold_albums = cursor.fetchall()
print("Least sold album per artist:")
print(least_sold_albums)
```

- **MIN:** Calculate and retrieve the smallest value
- **LEFT JOIN:** Combine data from the Album and Sale tables.
- **COUNT:** Count the occurrences of Sale.AlbumID.
- **GROUP BY:** Group the results by AlbumID and then by ArtistID.

4) **Rank the artists according to the sales of their albums**

```
# 4. Rank the artists according to the sales of their albums
cursor.execute('''
    SELECT ArtistID, SUM(Revenue) AS TotalRevenue
    FROM Album
    JOIN Sale ON Album.AlbumID = Sale.AlbumID
    GROUP BY ArtistID
    ORDER BY TotalRevenue DESC
''')
artists_ranked_by_sales = cursor.fetchall()
print("Artists ranked by sales of their albums:")
print(artists_ranked_by_sales)
```

- **JOIN:** Join the Album and Sale tables based on matching AlbumID.
- **GROUP BY:** Group the results by ArtistID to calculate the total revenue for each artist
- **ORDER BY:** Sort the result set in descending order based on TotalRevenue

5) **CTE (Common Table Expression) to update another artist into the company**

```
# 5. CTE (Common Table Expression) to update another artist into the company
update_data = ('New Name', 'New Role', 1)
query = '''
    UPDATE Artist
    SET ArtistName = ?, Role = ?
    WHERE ArtistID = ?
'''

cursor.execute(query, update_data)

# Fetch and display the updated data
cursor.execute('SELECT * FROM Artist WHERE ArtistID = ?', (1,))
updated_artist = cursor.fetchone()
print("Artist updated:")
print(updated_artist)
```

- **UPDATE:** Modify data in the Artist table
- **SET:** Set new values in the UPDATE statement.
- **WHERE:** Specify the source table or CTE to use in the UPDATE.

### 6) Combine artist and contract data

```
# 6. Combine artist and contract data
cursor.execute('''
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY Artist.ArtistID) AS RowNum
    FROM Artist
    LEFT JOIN Contract ON Artist.ArtistID = Contract.ArtistID
''')
combined_data = cursor.fetchall()
print("Combined data from Artist and Contract tables with RowNum:")
for row in combined_data:
    print(row)
```

- **ROW_NUMBER() OVER (ORDER BY Artist.ArtistID)** order the rows based on the ArtistID in ascending order and assign a row number to each row.
- **LEFT JOIN:** Join the tables based on matching ArtistID from the Artist table to the Contract table.

## Chaw Thiri San(12225272)

## Part 2: Intermediate and Advanced Queries

Intermediate Queries

- Total Revenue generated by each artist
- The top 10 BEST SELLING ALBUMS
- Artists having albums with less than 3 songs

1.Total Revenue Generated by Each Artist:

- The first query calculates the total revenue generated by each artist by performing a **left join** on the Artist table with the Album and Sale tables.
- The **SUM aggregate function** is used to calculate the total revenue for each artist, and the **GROUP BY** clause is used to group the results by ArtistID and ArtistName. The final result is **ordered by** ArtistID.

```
cursor.execute('''
  SELECT Artist.ArtistID, Artist.ArtistName, SUM(Sale.Revenue) AS TotalRevenue
    FROM Artist LEFT JOIN Album ON Artist.ArtistID = Album.ArtistID
    LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    GROUP BY Artist.ArtistID, Artist.ArtistName
    ORDER BY Artist.ArtistID;

''')
```

2. Top 10 Best-Selling Albums:

- The second query finds the top 10 best-selling albums based on sale revenue.
- It utilizes a **join** between the Album, Artist, and Sale tables.
- The **SUM aggregate function** is applied to calculate the total revenue for each album, and the results are **grouped by** AlbumID.
- The final output is ordered by total revenue in **descending order**, and the **LIMIT 10** clause is used to retrieve only the top 10 results.

```
cursor.execute('''
    SELECT
    Album.AlbumTitle AS AlbumTitle, Artist.ArtistName AS ArtistName,
    SUM(Sale.Revenue) AS TotalRevenue FROM Album
    JOIN Artist ON Album.ArtistID = Artist.ArtistID
    LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    GROUP BY Album.AlbumID ORDER BY TotalRevenue DESC LIMIT 10;

''')
```

3. Artists with Albums Having Less Than 3 Songs:

- The third query identifies artists who have albums with less than three songs.
- It uses two **JOIN** keywords to connect Artist, Album, and Song tables.
- The **COUNT aggregate function** is used to count the number of songs for each album, and the **GROUP BY and HAVING clauses** are employed to filter albums with less than three songs.
- The final result is **ordered by** ArtistName and AlbumTitle.

```
cursor.execute('''
     SELECT ArtistName, AlbumTitle, COUNT(*) AS TotalSongs FROM Artist
      JOIN Album ON Artist.ArtistID = Album.ArtistID
      JOIN Song ON Album.AlbumID = Song.AlbumID
    WHERE Album.AlbumID IN (
        SELECT AlbumID FROM Song GROUP BY AlbumID HAVING COUNT(*) < 3)
    GROUP BY ArtistName, AlbumTitle
    ORDER BY ArtistName, AlbumTitle;
''')
```

Advance Query

1. Percentage Contribution of Invidual Album in an Artist Total Profits
2. Ranking the artist based on their income
3. Delete artists who contracts have expired before 2020

**1) Percentage Contribution of Invidual Album in an Artist Total Profits**

- The following query calculates the total revenue and percentage contribution of each album associated with a specific artist. It's especially useful for analyzing the financial performance of individual albums within an artist's career.
- Artist9 has been chosen for analysis in this example.
- A SQL query is executed to retrieve the total revenue generated by the specified artist. The **SUM aggregate function** is used to calculate the total revenue from album sales of that artist.
- The fetched total revenue result is then stored in the total_revenue variable.
- Another SQL query is executed to retrieve individual album revenue and the percentage contribution of each album to the total revenue of the artist.
- **100 * SUM(Sale.Revenue) / ? AS RevenuePercentage:** calculates the percentage contribution of each album to the total revenue. The ? is a placeholder for a parameter that will be provided later. The result is aliased as RevenuePercentage.
- The results are ordered by album title.
- Then, the final answer is fetched into a Pandas DataFrame (result_df) and printed to the console.
- A bar chart is generated using Matplotlib to visually represent the percentage contribution of each album to the artist's total revenue.

```
# Execute the SQL query to get total revenue for the artist
cursor.execute('''
    SELECT SUM(Sale.Revenue) AS TotalRevenue FROM Album LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    WHERE Album.ArtistID = (SELECT ArtistID FROM Artist WHERE ArtistName = ?)
''', (artist_name,))

# Fetch the total revenue
total_revenue_result = cursor.fetchone()
total_revenue = total_revenue_result[0] if total_revenue_result else 0

# Execute the SQL query to get individual album revenue and percentage
cursor.execute(f'''
    SELECT Album.AlbumTitle, SUM(Sale.Revenue) AS AlbumRevenue, 100 * SUM(Sale.Revenue) / ? AS RevenuePercentage
    FROM Album
    LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    WHERE Album.ArtistID = (SELECT ArtistID FROM Artist WHERE ArtistName = ?)
    GROUP BY Album.AlbumID, Album.AlbumTitle
    ORDER BY Album.AlbumTitle;
''', (total_revenue, artist_name))
```

## 2) Ranking the artist based on their income

- The script begins by executing a **complex SQL query using a Common Table Expression (CTE) and Window Function**.
- This query ranks artists based on their total sale revenue and categorizes them into revenue classes.
- The **CTE named ArtistRevenue** aggregates sales revenue for each artist and calculates the rank using the ROW_NUMBER window function.
- The main query selects the artist's name, total revenue, and assigns a revenue class based on predefined thresholds (Class A, B, or C).
- The result is **ordered by** the calculated revenue rank.

```
cursor.execute('''
    SELECT SUM(Sale.Revenue) AS TotalRevenue FROM Album LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    WHERE Album.ArtistID = (SELECT ArtistID FROM Artist WHERE ArtistName = ?)
''', (artist_name,))
```

```
cursor.execute(f'''
    SELECT Album.AlbumTitle, SUM(Sale.Revenue) AS AlbumRevenue, 100 * SUM(Sale.Revenue) / ? AS RevenuePercentage
    FROM Album
    LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
    WHERE Album.ArtistID = (SELECT ArtistID FROM Artist WHERE ArtistName = ?)
    GROUP BY Album.AlbumID, Album.AlbumTitle
    ORDER BY Album.AlbumTitle;
''', (total_revenue, artist_name))
```

### 3) Delete artists who contracts have expired before 2020

- The third advanced SQL aims  to delete artists whose contracts have expired before the year 2020.
- The script retrieves the count of artists before the deletion operation for reference.
- It then performs a **deletion operation** using a **DELETE** clause, removing artists whose contract end date is before January 1, 2020.
- The count of artists after the deletion is fetched and the changes are committed to the database.
- The script includes **error-handling mechanisms** to catch and print any exceptions that might occur during execution. In case of an error, it **rolls back** the changes to maintain data integrity.
- Finally, the database connection is closed.

```
try:
    # 2. Ranking artist based on their sale revenue
    cursor.execute('''
        WITH ArtistRevenue AS (
        SELECT Artist.ArtistID, ArtistName, COALESCE(SUM(Sale.Revenue), 0) AS TotalRevenue,
        ROW_NUMBER() OVER (ORDER BY COALESCE(SUM(Sale.Revenue), 0) DESC) AS RevenueRank FROM Artist
        LEFT JOIN Album ON Artist.ArtistID = Album.ArtistID
        LEFT JOIN Sale ON Album.AlbumID = Sale.AlbumID
        GROUP BY Artist.ArtistID, ArtistName
        )

        SELECT ArtistName, TotalRevenue,
        CASE
            WHEN TotalRevenue >= 100000 THEN 'Class A'
            WHEN TotalRevenue >= 50000 THEN 'Class B'
            ELSE 'Class C'
        END AS RevenueClass FROM
        ArtistRevenue ORDER BY     RevenueRank;

    ''')
```

```
except Exception as e:
    print(f"Error: {e}")
    conn.rollback()  # Roll back changes if an error occurs
finally:
    conn.close()
```

# IV. Github Repositories:

-Luis Felipe Depardon Jasso:
https://github.com/A01770043/DBManagementPractice/blob/e1aca299db35165d0638bff0208d1d608fac73d4/DataBase_Project.ipynb

- Vu Nguyen Bao Ngoc - 12225193: ERD Diagram
https://github.com/alexngocvu/dbclass/blob/main/Finals_Music%20Label_ERD%20Diagram.dmm

-Chaw Thiri San : 12225272
https://github.com/chaw-thiri/Music_Label_Database/tree/main

- Thiri Moe Htet
https://github.com/Ellie2499