Concert App using SQL Transactions

In [1]:
```
"""
Example1 TicketMaster's Concert transaction
Users can purchase tickets and get ticket refunds
"""

import sqlite3
import random

# Create a connection to an in-memory SQLite database
conn = sqlite3.connect(':memory:')
cursor = conn.cursor()

# Create the Concerts table
cursor.execute("""
CREATE TABLE Concerts (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  available_tickets INTEGER NOT NULL
);""")

# Add some sample concert data
concerts_data = [
    ('Concert 1', 10),
    ('Concert 2', 5),
    ('Concert 3', 15)
]

cursor.executemany("""
INSERT INTO Concerts (name, available_tickets) VALUES (?, ?)
""", concerts_data)

# Commit the changes
conn.commit()

def purchase_ticket(concert_id):
  print(f'Purchasing for {concert_id=}')
  with conn:
    # Begin a new transaction
    cursor.execute("""BEGIN;""")

    # Get the number of available_tickets for the specified concert
    cursor.execute("""
      SELECT available_tickets FROM Concerts WHERE id = ?;""", (con
    available_tickets = cursor.fetchone()[0]

    # Check if there are any tickets available for the specified co
    if available_tickets > 0:
      # Simulate payment processing to Visa/Stripe with a 50% chanc
      payment_successful = random.choice([True, False])
```

```
      # If payment is successful, update the number of available_ti
      if payment_successful:
        cursor.execute("""
        UPDATE Concerts SET available_tickets = available_tickets -
                        """, (concert_id,))
      else:
        # If payment fails, do not update available_tickets and pri
        print(f"Payment failed for concert_id {concert_id}. No tick

    # Commit the transaction
    conn.commit()

def refund_ticket(concert_id):
  print(f'Want refund for {concert_id=}')
  with conn:
    # Begin a new transaction
    cursor.execute("""BEGIN;""")

    # Increase the number of available_tickets for the specified co
    cursor.execute("""
    UPDATE Concerts SET available_tickets = available_tickets + 1 W
            """, (concert_id,))

    # Commit the transaction
    conn.commit()

# Simulate purchasing and refunding tickets
concert_id = random.randint(1, len(concerts_data))
purchase_ticket(concert_id)
concert_id = random.randint(1, len(concerts_data))
refund_ticket(concert_id)

# Print the final state of the concerts table
cursor.execute("SELECT * FROM Concerts")
print(cursor.fetchall())
```

```
Purchasing for concert_id=1
Want refund for concert_id=1
[(1, 'Concert 1', 10), (2, 'Concert 2', 5), (3, 'Concert 3', 15)]
```

In [3]:

```
# Create the ticket_purchase table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Tickets (
    id INTEGER PRIMARY KEY,
    user_id INTEGER NOT NULL,
    concert_id INTEGER NOT NULL,
    num_tickets INTEGER NOT NULL,
    purchase_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (concert_id) REFERENCES Concerts(id)
);
""")
cursor.execute("delete from Tickets;")

# Commit the changes
conn.commit()
```

In [5]:
```
"""
Example 1.2: Users can only purchase upto 6 tickets.
Here we see two additional patterns of code
- Run a query, get data, and decide if a value needs to be updated
- Run a query to modify another table
"""
def purchase_restricted(user_id, concert_id, num_tickets):
  print(f'Checking purchase restrictions for user_id={user_id}, con
  with conn:
    # Begin a new transaction
    cursor.execute("""BEGIN;""")

    # Get the number of tickets purchased by the user
    cursor.execute("""
    SELECT SUM(num_tickets) FROM Tickets WHERE user_id = ?;""", (us
    num_tickets_purchased = cursor.fetchone()[0]
    # Check if the user has reached the maximum ticket purchase lim
    if (num_tickets_purchased or 0) + num_tickets > 6:
      print("Purchase restriction: Maximum ticket purchase limit re
      return False
    else:
      # Update the number of available tickets for the concert
      cursor.execute("""
      UPDATE Concerts SET available_tickets = available_tickets - ?
      """, (num_tickets, concert_id))

      # Insert the ticket purchase entry into the Tickets table
      cursor.execute("""
      INSERT INTO Tickets (user_id, concert_id, num_tickets) VALUES
      """, (user_id, concert_id, num_tickets))
      # Commit the transaction
      conn.commit()
      return True

# Users trying to buy tickets
for (user, concert, numtix) in [(5, 2, 2), (5, 2, 1), (5, 2, 2), (6
  print('--->',  cursor.execute("""SELECT * FROM Tickets;""").fetch
  purchase_restricted(user, concert, numtix)
print('--->', cursor.execute("""SELECT * FROM Tickets;""").fetchall
```

```
---> []
Checking purchase restrictions for user_id=5, concert_id=2, num_tick
ets=2
---> [(1, 5, 2, 2, '2025-04-22 17:15:10')]
Checking purchase restrictions for user_id=5, concert_id=2, num_tick
ets=1
---> [(1, 5, 2, 2, '2025-04-22 17:15:10'), (2, 5, 2, 1, '2025-04-22
17:15:10')]
Checking purchase restrictions for user_id=5, concert_id=2, num_tick
ets=2
---> [(1, 5, 2, 2, '2025-04-22 17:15:10'), (2, 5, 2, 1, '2025-04-22
17:15:10'), (3, 5, 2, 2, '2025-04-22 17:15:10')]
Checking purchase restrictions for user_id=6, concert_id=2, num_tick
ets=2
---> [(1, 5, 2, 2, '2025-04-22 17:15:10'), (2, 5, 2, 1, '2025-04-22
17:15:10'), (3, 5, 2, 2, '2025-04-22 17:15:10'), (4, 6, 2, 2, '2025-
04-22 17:15:10')]
Checking purchase restrictions for user_id=6, concert_id=3, num_tick
ets=2
---> [(1, 5, 2, 2, '2025-04-22 17:15:10'), (2, 5, 2, 1, '2025-04-22
17:15:10'), (3, 5, 2, 2, '2025-04-22 17:15:10'), (4, 6, 2, 2, '2025-
04-22 17:15:10'), (5, 6, 3, 2, '2025-04-22 17:15:10')]
```

In [7]:

```
"""
Example 1.3: Find superfans who buy tickets to multiple concerts
Here we see an example of updates based on using more SQL logic
"""

# Create the Users table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Users (
    user_id INTEGER PRIMARY KEY,
    superfan INTEGER
);""")

# No superfans to start
cursor.execute("delete from Users;")
cursor.execute("""
INSERT INTO Users (user_id, superfan)
VALUES (5, 0), (6, 0);""")
conn.commit()

def find_superfans():
    print(f'Find super fans')
    with conn:
        # Begin a new transaction
        cursor.execute("""BEGIN;""")

        # Find users who buy tickets to >= 2 concerts and >=2 ticke
        cursor.execute("""
        UPDATE Users
        SET superfan = 1
        WHERE user_id IN (
          SELECT user_id
          FROM (
          SELECT user_id, COUNT(DISTINCT concert_id) AS distinct_co
```

```
            FROM Tickets
            GROUP BY user_id
            HAVING SUM(num_tickets) >= 2 AND COUNT(DISTINCT concert_i
            ) AS subquery
        );""")
        conn.commit()

find_superfans()
cursor.execute("SELECT * FROM Users;")
print('--->',  cursor.fetchall())
```

```
Find super fans
---> [(5, 0), (6, 1)]
```

In [ ]:
```
conn.close()
```

TO DO: Summarize in a few sentences what the code in this activity does.

This code spins up an in-memory SQLite DB and creates three tables—Concerts (to track shows and available tickets), Tickets (to record purchases), and Users (to flag superfans). It defines transactional Python functions to buy and refund tickets—updating availability and rolling back automatically on errors—and adds a variant that caps each user at six tickets per transaction. It then runs a query to mark anyone who's bought tickets to at least two different concerts as a superfan. Finally, the notebook simulates a few sample purchases and refunds, prints the final state of all tables, and closes the connection

In [ ]:
```

```