HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN I

BÀI TẬP LỚN

MÔN HỌC: CƠ SỞ DỮ LIỆU PHÂN TÁN

NHÓM 10 - NMH 10

Giảng viên hướng dẫn: Kim Ngọc Bách

STT	Họ tên	Mã sinh viên
1	Phan Thị Hồng Huế	B22DCCN356
2	Phạm Thị Hương Nhài	B22DCCN574
3	Hà Mạnh Dũng	B22DCCN125

Hà Nội 2025

MỤC LỤC

LỜI CẢM ƠN

Trước tiên, nhóm chúng em xin gửi lời cảm ơn chân thành và sâu sắc đến thầy Kim Ngọc Bách – người đã tận tình hướng dẫn, truyền đạt kiến thức và tạo điều kiện thuận lợi để chúng em hoàn thành bài tập lớn này. Sự tận tâm và đồng hành của thầy là nguồn động lực to lớn giúp chúng em vượt qua những khó khăn trong suốt quá trình thực hiện bài tập lớn môn học.

Chúng em cũng xin gửi lời cảm ơn đến các thầy cô trong Khoa đã xây dựng chương trình học đầy tính thực tiễn, giúp chúng em có cơ hội vận dụng kiến thức vào dự án cụ thể, từ đó tích lũy được thêm nhiều kinh nghiệm quý báu cho bản thân.

Một lời cảm ơn đặc biệt xin được gửi đến các thành viên trong nhóm – những người bạn đồng hành tuyệt vời. Trong suốt thời gian làm việc, dù có những lúc áp lực, bất đồng ý kiến hay mệt mỏi, nhưng tất cả đều cùng nhau nỗ lực, chia sẻ, động viên và hỗ trợ lẫn nhau để hoàn thành bài tập với tinh thần trách nhiệm và sự gắn kết cao nhất. Nhờ sự đồng lòng ấy, bài làm không chỉ là kết quả học tập mà còn là minh chứng cho tinh thần làm việc nhóm, cho sự trưởng thành của mỗi cá nhân trong tập thể.

Cuối cùng, tuy đã cố gắng hết sức, nhưng do thời gian và kiến thức còn hạn chế, bài làm không tránh khỏi những thiếu sót. Nhóm rất mong nhận được sự góp ý quý báu từ thầy để hoàn thiện hơn trong các dự án tiếp theo.

Trân trọng cảm ơn!

NỘI DUNG

I. Mô tả bài toán

1. Mô tả và mục tiêu

Bài tập lớn này yêu cầu sinh viên triển khai một hệ thống mô phỏng các phương pháp phân mảnh ngang (horizontal partitioning) trên một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở, cụ thể là PostgreSQL hoặc MySQL. Mục tiêu của bài toán là xây dựng một tập hợp các hàm Python nhằm quản lý dữ liệu đánh giá phim (ratings), bao gồm các bước: tải dữ liệu đầu vào, thực hiện phân mảnh dữ liệu theo hai phương pháp khác nhau, và chèn các bản ghi mới vào đúng phân mảnh tương ứng. Hai kỹ thuật phân mảnh được triển khai trong bài là phân mảnh theo dải giá trị (Range Partitioning) và phân mảnh kiểu vòng tròn (Round Robin Partitioning). Ngoài việc triển khai các hàm thực hiện phân mảnh, sinh viên còn phải phát triển các hàm hỗ trợ việc chèn dữ liệu mới sao cho đảm bảo dữ liệu được đưa vào đúng phân mảnh đã thiết lập ban đầu, từ đó đảm bảo tính nhất quán và đúng đắn của hệ thống dữ liệu phân mảnh. Qua bài tập này, sinh viên sẽ hiểu rõ hơn về nguyên lý và cơ chế hoạt động của phân mảnh ngang trong cơ sở dữ liệu, đồng thời nâng cao kỹ năng lập trình xử lý dữ liệu với Python và tương tác với hệ quản trị cơ sở dữ liệu.

2. Dữ liệu sử dụng

- Dữ liệu đầu vào là tập dữ liệu đánh giá phim được thu thập từ trang web MovieLens (http://movielens.org).
    + **Tên tệp:** ratings.dat
    + **Mô tả:** Chứa 10 triệu đánh giá và 100.000 thẻ áp dụng cho 10.000 bộ phim bởi 72.000 người dùng.
- Mỗi dòng trong tệp ratings.dat đại diện cho một đánh giá của một người dùng với một bộ phim, và có định dạng như sau: UserID::MovieID::Rating::Timestamp
    - **UserID:** (integer) Định danh duy nhất cho người dùng.
    - **MovieID:** (integer) Định danh duy nhất cho bộ phim.
    - **Rating:** (float) Giá trị đánh giá trên thang điểm 5 sao, có thể chia nửa sao (ví dụ: 0.5, 1.0, ..., 5.0).
    - **Timestamp:** (bigint) Số giây kể từ nửa đêm UTC ngày 1 tháng 1 năm 1970.

II. Triển khai bài toán

1. Công nghệ sử dụng

- **Ngôn ngữ lập trình:** Python 3.12.x (theo cấu hình máy ảo/máy chấm điểm).
- **Hệ quản trị cơ sở dữ liệu:** PostgreSQL.
- **Thư viện Python:** psycopg2 (để kết nối và tương tác với PostgreSQL).
- **Hàm hỗ trợ:** Sử dụng các hàm và hằng số được cung cấp/định nghĩa trong testHelper.py và Interface.py của thầy, cùng với các hàm tiện ích chúng em đã tự xây dựng.

2. Lược đồ cơ sở dữ liệu

- **Bảng gốc (Main Table):** ratings
  - **userid (INT):** Định danh người dùng.
  - **movieid (INT):** Định danh phim.
  - **rating (REAL):** Điểm đánh giá (0.0 đến 5.0).
  - Cột timestamp và các cột extra (từ định dạng tệp thô) sẽ được loại bỏ sau khi tải dữ liệu.
- **Các bảng phân mảnh theo dải (Range Partitions):**
  - Được đặt tên theo định dạng: range_part<index>, với index bắt đầu từ 0 (ví dụ: range_part0, range_part1,...).
  - **Gồm có:** userid (INT), movieid (INT), rating (REAL).
- **Các bảng phân mảnh Round Robin (Round Robin Partitions):**
  - Được đặt tên theo định dạng: rrobin_part<index>, với index bắt đầu từ 0 (ví dụ: rrobin_part0, rrobin_part1,...).
  - **Gồm có:** userid (INT), movieid (INT), rating (REAL).

3. Chi tiết triển khai

3.1. Xử lý dữ liệu, gọi các hàm tích hợp

3.1.1 Hàm getopenconnection

Hàm **getopenconnection** đóng vai trò là nền tảng cho mọi hoạt động tương tác với cơ sở dữ liệu PostgreSQL. Mục đích chính của nó là thiết lập và trả về một đối tượng kết nối mới tới cơ sở dữ liệu. Hàm này nhận các tham số như tên người dùng, mật khẩu, tên cơ sở dữ liệu, địa chỉ máy chủ và cổng, với tất cả đều có các giá trị mặc định được cấu hình sẵn. Khi được gọi, hàm sẽ cố gắng sử dụng psycopg2.connect() để tạo kết nối với các tham số đã cung cấp. Nếu có bất kỳ lỗi nào xảy ra trong quá trình kết nối (ví dụ: thông tin đăng nhập không chính xác, cơ sở dữ liệu không tồn tại hoặc máy chủ không khả dụng), hàm sẽ bắt ngoại lệ psycopg2.Error, in ra thông báo lỗi chi tiết và sau đó ném lại (raise) ngoại lệ đó. Việc ném lại ngoại lệ là rất quan trọng, cho phép các phần khác của chương trình biết về sự cố kết nối và thực hiện các hành động xử lý lỗi phù hợp.

```
def getopenconnection(
    user=DB_USER_PG_DEFAULT,
    password=DB_PASS_PG_DEFAULT,
    dbname=DATABASE_NAME,
    host=DB_HOST_PG_DEFAULT,
    port=DB_PORT_PG_DEFAULT):
    try:
        conn = psycopg2.connect(dbname=dbname, user=user, password=password, host=host, port=port)
        return conn
    except psycopg2.Error as e:
        print(f"Lỗi kết nối đến PostgreSQL: {e}")
        raise
```

Hình 3.1.1. Mã hàm getopenconnection()

3.1.2. Hàm _execute_query_pg_with_provided_conn

Hàm **_execute_query_pg_with_provided_conn** là một tiện ích nội bộ thiết yếu, được thiết kế để thực thi mọi truy vấn SQL trên một kết nối

PostgreSQL đã được thiết lập. Chức năng cốt lõi của nó là tập trung hóa logic thực thi truy vấn, đảm bảo sử dụng tham số hóa truy vấn để ngăn chặn các lỗ hổng bảo mật như SQL Injection. Hàm này chấp nhận đối tượng kết nối cơ sở dữ liệu (conn), chuỗi truy vấn SQL (query), các tham số tùy chọn cho truy vấn (params), và một tùy chọn fetch ('one', 'all', hoặc None) để chỉ định cách lấy kết quả.

Cách hoạt động của hàm bắt đầu bằng việc kiểm tra tính hợp lệ của kết nối; nếu kết nối không hợp lệ hoặc đã đóng, nó sẽ ném ra lỗi psycopg2.InterfaceError. Để đảm bảo an toàn và giải phóng tài nguyên hiệu quả, hàm sử dụng câu lệnh with conn.cursor() as cur: để tạo và quản lý con trỏ. Sau khi truy vấn được thực thi bằng cur.execute(query, params), hàm sẽ lấy kết quả dựa trên giá trị của fetch. Một điểm mạnh đáng kể của hàm này là khả năng xử lý lỗi SQL. Khi một lỗi psycopg2.Error xảy ra, hàm sẽ in thông báo lỗi chi tiết, bao gồm truy vấn và các tham số liên quan. Đặc biệt, nếu kết nối đang trong một giao dịch và gặp lỗi, hàm sẽ cố gắng thực hiện conn.rollback() để hoàn tác các thay đổi chưa được cam kết, từ đó duy trì tính nhất quán của dữ liệu. Cuối cùng, ngoại lệ sẽ được ném lại để phần gọi có thể xử lý sâu hơn.

```
def _execute_query_pg_with_provided_conn(conn, query, params=None, fetch=False):
    if not conn or conn.closed:
        raise psycopg2.InterfaceError("Kết nối không hợp lệ hoặc đã đóng trong _execute_query_pg_with_provided_conn.")
    result = None
    try:
        with conn.cursor() as cur:
            cur.execute(query, params)
            if fetch == 'one':
                result = cur.fetchone()
            elif fetch == 'all':
                result = cur.fetchall()
    except psycopg2.Error as e:
        print(f"Lỗi SQL trong _execute_query_pg_with_provided_conn: {e}")
        print(f"  Truy vấn thất bại: {query}")
        if params: print(f"  Tham số: {params}")
        if not conn.autocommit and conn.status == psycopg2.extensions.STATUS_IN_ERROR:
            print("  Đang cố gắng rollback do lỗi...")
            try:
                conn.rollback()
            except psycopg2.Error as rb_err:
                print(f"  Lỗi trong quá trình rollback: {rb_err}")
        raise
    return result
```

Hình 3.1.2. Mã hàm _execute_query_pg_with_provided_conn()

3.1.3. Hàm create_db_if_not_exists

Hàm **create_db_if_not_exists** đảm bảo sự tồn tại của một cơ sở dữ liệu PostgreSQL cụ thể. Mục đích của nó là kiểm tra xem cơ sở dữ liệu đã có chưa; nếu chưa, hàm sẽ tạo mới, còn nếu đã tồn tại, nó sẽ bỏ qua quá trình tạo.

Cách hoạt động của hàm này bao gồm việc thiết lập một kết nối tạm thời đến cơ sở dữ liệu mặc định với quyền superuser. Sau đó, kết nối này được đặt vào chế độ autocommit, đảm bảo rằng lệnh CREATE DATABASE sẽ được thực thi ngay lập tức mà không cần thêm bước xác nhận thủ công. Hàm tiếp tục bằng cách truy vấn bảng hệ thống pg_database để kiểm tra sự hiện diện của tên cơ sở dữ liệu mong muốn. Nếu cơ sở dữ liệu chưa tồn tại, nó sẽ thực thi lệnh CREATE DATABASE với tên cơ sở dữ liệu được xử lý an toàn bằng

psycopg2.extensions.quote_ident để tránh các vấn đề liên quan đến ký tự đặc biệt hoặc từ khóa SQL. Cuối cùng, hàm bao gồm cơ chế xử lý lỗi và đảm bảo kết nối tạm thời được đóng gọn gàng trong khối finally, giải phóng tài nguyên hệ thống.

```
def create_db_if_not_exists(dbname):
    print(f"Đảm bảo cơ sở dữ liệu '{dbname}' tồn tại...")
    conn_default = None
    try:
        conn_default = getopenconnection(dbname='postgres')
        conn_default.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
        cur = conn_default.cursor()

        cur.execute('SELECT 1 FROM pg_database WHERE datname = %s;', (dbname,))
        exists = cur.fetchone()

        if not exists:
            cur.execute(f'CREATE DATABASE {psycopg2.extensions.quote_ident(dbname, cur)}')
            print(f"Cơ sở dữ liệu '{dbname}' đã được tạo thành công.")
        else:
            print(f"Cơ sở dữ liệu '{dbname}' đã tồn tại. Bỏ qua việc tạo.")
    except psycopg2.Error as e:
        print(f"Lỗi khi tạo hoặc kiểm tra cơ sở dữ liệu '{dbname}': {e}")
        raise
    finally:
        if conn_default:
            conn_default.close()
```

Hình 3.1.3. Mã hàm create_db_if_not_exits()

3.1.4. Hàm _count_partitions_with_prefix

Hàm **_count_partitions_with_prefix** được thiết kế để đếm chính xác số lượng bảng trong schema public của cơ sở dữ liệu hiện tại mà tên của chúng bắt đầu bằng một tiền tố cụ thể. Mục đích chính của nó là kiểm tra số lượng bảng phân vùng đã được tạo, giúp các hàm khác (như rangeinsert và roundrobininsert) có thể hoạt động hiệu quả. Hàm này nhận vào hai tham số: openconnection, là đối tượng kết nối cơ sở dữ liệu đã mở, và prefix_to_match, một chuỗi tiền tố (ví dụ: "rrobin_part") dùng để tìm kiếm các bảng.

Về cách hoạt động, đầu tiên, hàm kiểm tra tính hợp lệ của kết nối. Sau đó, nó thực thi một truy vấn SQL

```
query = "SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = 'public' AND table_name LIKE %s;"
```

để tìm kiếm các bảng có tên khớp với tiền tố. Để đảm bảo khớp chính xác với cách PostgreSQL lưu trữ tên bảng (thường là chữ thường), tiền tố được chuyển đổi thành chữ thường (.lower()) trước khi so sánh. Cuối cùng, hàm sử dụng **_execute_query_pg_with_provided_conn** để thực thi truy vấn và lấy về số lượng bảng tìm được.

```python
def _count_partitions_with_prefix(openconnection, prefix_to_match):
    conn = openconnection
    if not conn or conn.closed:
        print("Lỗi trong _count_partitions_with_prefix: Kết nối không hợp lệ.")
        return 0
    query = "SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = 'public' AND table_name LIKE %s;"
    try:
        count = _execute_query_pg_with_provided_conn(conn, query, (f"{prefix_to_match}%",), fetch='one')[0]
    except psycopg2.Error as e:
        print(f"Lỗi SQL khi đếm phân vùng (tiền tố: '{prefix_to_match}%'): {e}")
        count = 0
    return count
```

Hình 3.1.4. Mã hàm _count_partitions_with_prefix()

3.1.5. Hàm create_partition_table

Hàm **create_partition_table** có mục đích quan trọng là tạo một bảng mới với cấu trúc schema chuẩn (UserID, MovieID, Rating) nếu bảng đó chưa tồn tại. Đây là một hàm tiện ích được thiết kế để đảm bảo rằng tất cả các bảng con, đặc biệt là các bảng phân vùng được tạo ra bởi các chiến lược roundrobinpartition hay rangepartition, đều có cùng một cấu trúc dữ liệu nhất quán. Hàm nhận vào hai tham số: một cursor cơ sở dữ liệu để thực thi lệnh SQL và table_name là tên của bảng cần tạo.

Về cách hoạt động, hàm sẽ chuyển đổi table_name thành chữ thường (.lower()) để phù hợp với quy ước của PostgreSQL. Sau đó, nó thực thi lệnh SQL CREATE TABLE IF NOT EXISTS {sql_table_name} (...). Mệnh đề IF NOT EXISTS là một tính năng hữu ích, giúp hàm không gây ra lỗi nếu bảng với tên đã cho đã tồn tại, mà thay vào đó chỉ đơn giản là bỏ qua thao tác tạo. Schema của bảng được định nghĩa rõ ràng với ba cột UserID (INT), MovieID (INT), và Rating (FLOAT), sử dụng các hằng số được khai báo sẵn để tăng cường khả năng đọc và dễ bảo trì mã.

```python
def create_partition_table(cursor, table_name):
    sql_table_name = table_name.lower()
    cursor.execute(f'''
        CREATE TABLE IF NOT EXISTS {sql_table_name} (
            {USER_ID_COLNAME} INT,
            {MOVIE_ID_COLNAME} INT,
            {RATING_COLNAME} FLOAT
        );
    ''')
```

Hình 3.1.5. Mã hàm create_partition_table()

3.2. LoadRatings()

Hàm **loadratings()** chịu trách nhiệm tải dữ liệu từ một tệp tin được chỉ định vào một bảng PostgreSQL, cụ thể ở đây là bảng ratings.

```python
def loadratings(ratingsTableName, ratingsFilePath, openconnection):
    with openconnection.cursor() as cur:
        cur.execute(f'''
            CREATE TABLE IF NOT EXISTS {ratingsTableName} (
                UserID INT,
                MovieID INT,
                Rating FLOAT
            );
        ''')
        with open(ratingsFilePath, 'r') as f:
            for line in f:
                parts = line.strip().split("::")
                if len(parts) < 3: continue
                userid, movieid, rating = int(parts[0]), int(parts[1]), float(parts[2])
                cur.execute(f'''
                    INSERT INTO {ratingsTableName} (UserID, MovieID, Rating)
                    VALUES (%s, %s, %s);
                ''', (userid, movieid, rating))
        openconnection.commit()
```

Hình 3.2.a: Phần toàn bộ mã hàm loadings()

Hàm này có 2 nhiệm vụ chính:

Đảm bảo bảng đích tồn tại: Nó kiểm tra xem một bảng với tên ratingsTableName đã tồn tại trong cơ sở dữ liệu PostgreSQL của bạn hay chưa. Nếu chưa, nó sẽ tạo bảng đó với cấu trúc đã định nghĩa (UserID, MovieID, Rating).

Đọc dữ liệu từ tệp và chèn vào bảng: Sau đó, nó đọc từng dòng từ tệp ratingsFilePath, phân tích các dữ liệu liên quan (UserID, MovieID, Rating), và chèn dữ liệu này dưới dạng một hàng mới vào bảng cơ sở dữ liệu đích.

Phân tích rõ hơn về cách hoạt động hàm loadratings():

Bước 1: Tạo một công cụ giao tiếp với database

```python
def loadratings(ratingsTableName, ratingsFilePath, openconnection):
    with openconnection.cursor() as cur:
```

Hình 3.2.b: Phần tạo kết nối với database

openconnection là đối tượng kết nối database của bạn. Để thực hiện các lệnh SQL, bạn cần tạo một con trỏ (cursor) từ kết nối này. cú pháp with ... as ... đảm bảo rằng con trỏ (cur) sẽ được tự động đóng và giải phóng tài nguyên khi khối code này kết thúc, dù có lỗi xảy ra hay không.

```python
cur.execute(f'''
    CREATE TABLE IF NOT EXISTS {ratingsTableName} (
        UserID INT,
        MovieID INT,
        Rating FLOAT
    );
''')
```

Hình 3.2.c: Phần tạo bảng nếu bảng chưa tồn tại

cur.execute(...): Là phương thức chính để gửi một lệnh SQL đến database thông qua con trỏ.

f''' ... ''': Là một f-string của Python. Nó cho phép dễ dàng chèn giá trị của biến Python vào chuỗi SQL. Ở đây, {ratingsTableName} sẽ được thay thế bằng tên bảng.

Lệnh SQL CREATE TABLE IF NOT EXISTS {ratingsTableName} (UserID INT, MovieID INT, Rating FLOAT) dùng để thiết lập cấu trúc lưu trữ dữ liệu trong database. CREATE TABLE ra lệnh tạo một bảng mới. Mệnh đề IF NOT EXISTS đảm bảo rằng nếu bảng với ratingsTableName đã tồn tại, hệ thống sẽ bỏ qua việc tạo bảng đó mà không báo lỗi, giúp code chạy mượt mà khi được thực thi nhiều lần. Cuối cùng, phần (UserID INT, MovieID INT, Rating FLOAT) định nghĩa ba cột chính của bảng: UserID và MovieID sẽ lưu trữ mã định danh dạng số nguyên, còn Rating sẽ chứa điểm đánh giá dưới dạng số thực.

```
with open(ratingsFilePath, 'r') as f:
    for line in f:
        parts = line.strip().split("::")
        if len(parts) < 3: continue
        userid, movieid, rating = int(parts[0]), int(parts[1]), float(parts[2])
        cur.execute(f'''
            INSERT INTO {ratingsTableName} (UserID, MovieID, Rating)
            VALUES (%s, %s, %s);
        ''', (userid, movieid, rating))
```

Hình 3.2.d: Phần ghi dữ liệu từ tệp chỉ định vào bảng

with open(ratingsFilePath, 'r') as f:: Hàm mở tệp dữ liệu được chỉ định bởi ratingsFilePath ở chế độ đọc ('r'). Cũng sử dụng with, đảm bảo tệp sẽ được đóng tự động khi không còn sử dụng, ngay cả khi có lỗi.

for line in f:: Vòng lặp này sẽ đọc tệp tin từng dòng một. Mỗi dòng được gán vào biến line.

parts = line.strip().split("::"): line.strip(): loại bỏ bất kỳ khoảng trắng nào (bao gồm cả ký tự xuống dòng \n) ở đầu và cuối dòng. .Split("::"): Chia chuỗi dòng thành một danh sách các chuỗi con (parts) dựa trên dấu phân cách "::". Ví dụ, dòng "1::123::4.5::978300760\n" sẽ trở thành ['1', '123', '4.5', '978300760'].

if len(parts) < 3: continue: Đây là một bước kiểm tra cơ bản về định dạng dữ liệu. Tệp ratings.dat thường có ít nhất 3 phần (UserID, MovieID, Rating). Nếu một dòng không đủ 3 phần, nó có thể là dòng trống hoặc định dạng sai. Lệnh continue sẽ bỏ qua phần còn lại của vòng lặp cho dòng hiện tại và chuyển ngay sang xử lý dòng tiếp theo.

userid, movieid, rating = int(parts[0]), int(parts[1]), float(parts[2]): Các giá trị từ danh sách parts được giải nén vào các biến userid, movieid, rating.

cur.execute(f''' INSERT INTO {ratingsTableName} (...) VALUES (%s, %s, %s); ''', (userid, movieid, rating)): Đây là lệnh SQL để chèn dữ liệu của một hàng vào bảng.

- **%s, %s, %s:** Đây là các placeholder (phần giữ chỗ) tiêu chuẩn của psycopg2. Chúng đại diện cho các giá trị sẽ được chèn vào.
- **(userid, movieid, rating):** Đây là một tuple chứa các giá trị thực tế của userid, movieid, rating. psycopg2 sẽ tự động và an toàn thay thế các giá trị này vào các placeholder %s theo thứ tự.
- Tại sao lại dùng %s mà không chèn trực tiếp giá trị vào f-string?: Đây là một thực hành rất quan trọng và an toàn để:
- Ngăn chặn SQL Injection: Tránh việc kẻ xấu đưa mã SQL độc hại vào dữ liệu của hệ thống và database thực thi nó.
- Xử lý kiểu dữ liệu: psycopg2 sẽ tự động chuyển đổi kiểu dữ liệu từ Python sang kiểu phù hợp trong PostgreSQL.

openconnection.commit(): Trong database, các lệnh thay đổi dữ liệu (như CREATE TABLE, INSERT) thường được nhóm lại thành một giao dịch (transaction). Các thay đổi này chỉ thực sự được lưu vĩnh viễn vào database khi commit.

3.3. Range_Partition()

- Hàm rangepartition() thực hiện phân mảnh ngang trên một bảng cơ sở dữ liệu dựa trên thuộc tính Rating. Dữ liệu sau khi phân mảnh sẽ được chia thành các bảng con dựa trên các dải giá trị đồng đều cột Rating.

- **def rangepartition(ratingsTable, numberOfPartitions, openconnection):**
  + **ratingsTable (str):** Tên bảng gốc mà ta muốn phân mảnh. Hàm sẽ đọc dữ liệu của bảng này.
  + **numberOfPartitions (int):** Số lượng phân mảnh cần tạo. Hàm sẽ tạo N bảng con, mỗi bảng chứa một dải giá trị Rating cụ thể.
  + **openconnection (psycopg2.connection):** Dùng để kết nối đến cơ sở dữ liệu PostgreSQL đã được mở.

- **Khởi tạo, định nghĩa lại biến:**

```
def rangepartition(ratingsTableName, numberOfPartitions, openconnection):
    actual_base_table_name = ratingsTableName.lower()
    n = numberOfPartitions
```

  + **actual_base_table_name = ratingsTableName.lower():** chuyển đổi tên bảng gốc sang chữ thường => Tránh các lỗi không khớp tên bảng do sự khác biệt về chữ hoa/thường giữa Python và PostgreSQL.
  + **n = numberOfPartitions:** gán **numberOfPartitions** vào biến **n** để mã nguồn gọn hơn và dễ đọc hơn trong suốt hàm**.**

- **Kiểm tra dữ liệu đầu vào:**

```
if n <= 0:
    raise ValueError("numberOfPartitions must be greater than 0 for range partitioning.")
print(f"Partitioning table '{actual_base_table_name}' into {n} range partitions (PostgreSQL)...")
conn = openconnection
if not conn or conn.closed:
    raise Exception("Range_Partition: Invalid or closed database connection provided.")
```

+ Kiểm tra điều kiện: nếu số lượng phân mảnh n <= 0 hàm sẽ tả ra ValueError để báo lỗi và ngăn chặn các thao tác không hợp lệ.
  => Yêu cầu số lượng phân mảnh phải lớn hơn 0.
+ Thỏa mãn điều kiện thì sẽ in ra một dòng thông báo cho người dùng biết quá trình phân mảnh đang bắt đầu và với các tham số tương ứng như nào.
+ Gán **conn = openconnection** giúp nhất quán khi sử dụng trong hàm.
+ Kiểm tra xem **conn** có tồn tại hoặc bị đóng hay không. Nếu kết nối không hợp lệ, hàm sẽ trả về Exception đảm bảo rằng mọi thao tác database sau đó đều được thực hiện trên một kết nối đang hoạt động.

-

- **Tính toán Bước Dải:**

```
range_step = (MAX_RATING_CONST - MIN_RATING_CONST) / n
if range_step == 0 and n > 0:
    raise ValueError("Range step is zero. Check MIN_RATING_CONST, MAX_RATING_CONST, or numberOfPartitions.")
previous_upper_bound = MIN_RATING_CONST
```

+ **range_step:** Tính toán độ rộng mỗi giải địa chỉ Rating mà 1 phân mảnh sẽ chứa.
+ Sử dụng các hằng số **MAX_RATING_CONST, MIN_RATING_CONST** : => đảm bảo sự nhất quán và dễ cấu hình. Phép chia này sẽ luôn ra số thực.
+ **range_step = 0** khi và chỉ khi **MAX_RATING_CONST = MIN_RATING_CONST (5.0 - 0.0 = 5.0 > 0** => không xảy ra trường hợp 2 hằng số bằng nhau được) hoặc số lượng phân mảnh n quá lớn gây nên lỗi làm tròn => range_step gần như 0.
  => Việc kiểm tra này sẽ giúp xác định được vấn đề với việc thiết lập các hằng số hoặc n.

- **Khởi tạo previous_upper_bound và khối Try-except:**
  + **previous_upper_bound = MIN_RATING_CONST =>** để theo dõi giới hạn trên của dải Rating của phân mảnh trước đó, giúp đảm bảo tính rời rạc của dữ liệu. Khởi tạo bằng giá trị **MIN_RATING_CONST** vì đây là giá trị bắt đầu của phân mảnh đầu tiên.
  + Khối **try … except..** : bao bọc toàn bộ logic chính của hàm. Bất kỳ lỗi nào xảy ra trong **try** sẽ bị bắt lại bởi **except**.
  + Trong khối **except**, hàm sẽ in thông báo lỗi, cố gắng **rollback** giao dịch (nếu không ở chế độ autocommit), và sau đó **raise** lại lỗi để hàm gọi bên ngoài (**AssignmentTester.py**) có thể xử lý. Điều này đảm bảo tính bền vững của ứng dụng.

+ **with conn.cursor() as cur:** lấy 1 đối tượng con trỏ **cur** từ kết nối => đảm bảo con trỏ sẽ tự động được đóng khi khối with kết thúc dù có lỗi hay không => giải phóng tài nguyên database 1 cách hiệu quả.

-
- **Vòng lặp Tạo và điền dữ liệu cho từng phân mảnh.**

```
for i in range(n):
    table_name = f"{RANGE_TABLE_PREFIX}{i}"
    _execute_query_pg_with_provided_conn(conn, f"DROP TABLE IF EXISTS {table_name} CASCADE;")
    create_partition_table(cur, table_name)
    current_upper = MIN_RATING_CONST + (i + 1) * range_step
    if i == n - 1:
        current_upper = MAX_RATING_CONST
    condition = ""
    if i == 0:
        condition = f"{RATING_COLNAME} >= %s AND {RATING_COLNAME} <= %s"
        params = (MIN_RATING_CONST, current_upper)
    else:
        condition = f"{RATING_COLNAME} > %s AND {RATING_COLNAME} <= %s"
        params = (previous_upper_bound, current_upper)
    insert_sql = f"""
    INSERT INTO {table_name} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
    SELECT {USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME} FROM {actual_base_table_name}
    WHERE {condition};
    """
    _execute_query_pg_with_provided_conn(conn, insert_sql, params)
    previous_upper_bound = current_upper
    print(
        f"Created and populated partition '{table_name}' (Ratings: {previous_upper_bound - range_step if i > 0 else MIN_RATING_CONST:.2f} to {current_upper:.2f}).")
```

+ Vòng lặp for chạy n lần, mỗi lần cho một phân mảnh từ 0 đến n-1.
+ Tạo tên bảng phân mảnh động, theo đúng quy ước **range_part<index>** Sử dụng f-string giúp việc tạo bảng dễ dàng và rõ ràng hơn.
+ **DROP TABLE IF EXISTS {table_name} CASCADE:** xóa các bảng phân mảnh cũ nếu có. **CASCADE** để xóa các đối tượng phụ thuộc (ví dụ: index, view) liên quan đến bảng, đảm bảo quá trình luôn sạch sẽ.
+ **create_partition_table(cur, table_name):** Gọi hàm hỗ trợ để tạo bảng phân mảnh với lược đồ (**userid INT, movieid INT, rating REAL**). Hàm này nhận con trỏ **cur** để thực thi lệnh trong cùng một giao dịch.
+ Xử lý dải giá trị và tính đầy đủ / rời rạc;
    + Tính giới hạn trên của dải hiện tại: **current_upper = MIN_RATING_CONST + (i + 1) * range_step.**
      Ví dụ: nếu range_step là 1.0, i = 0 => current_upper là 1.0;
      i = 1 thì current_step là 2.0.
    + **if i == n - 1: current_upper = MAX_RATING_CONST** => giải quyết sai số dấu phẩy động bằng cách đảm bảo phân mảnh cuối cùng luôn bao gồm chính xác thẳng bằng MAX_RATING_CONST (5.0) để xử lý lỗi làm tròn và đảm bảo tính đầy đủ của phân mảnh.
    + Điều kiện WHERE - đảm bảo tính rời rạc: Đảm bảo mỗi bản ghi Rating chỉ thuộc về một và chỉ một phân mảnh.
      Phân mảnh đầu tiên: i == 0: Sử dụng >= cho giới hạn dưới (MIN_RATING_CONST) và <= cho giới hạn trên (current_upper). Dùng >= để đảm bảo gồm giá trị nhỏ nhất của toàn dải (0.0). Dùng <= để bao gồm giá trị ranh giới trên nếu có.
      Các phân mảnh sau: else: Sử dụng > cho giới hạn dưới (previous_upper_bound) và <= cho giới hạn trên (current_upper)

=> Đảm bảo tính rời rạc của phân mảnh. Một bản ghi Rating nằm chính xác tại ranh giới giữa hai dải sẽ chỉ được gán vào một và chỉ một phân mảnh (phân mảnh có dấu <=), không bị trùng lặp.
VD: range_step = 2.5, n = 2 => một bản ghi có Rating = 2.5 sẽ rơi vào range_part0.

+ Tham số hóa truy vấn (%s) : kết hợp truyền params => thực hiện bảo mật tốt nhất ngăn chặn tấn công SQL Injection và đảm bảo xử lý kiểu dữ liệu chính xác. psycopy2 sẽ tự động xử lý việc thoát các ký tự đặc biệt và chuyển đổi kiểu dữ liệu, giúp truy vấn an toàn và đáng tin cậy hơn.
+ **insert_sql = f""" INSERT INTO ... SELECT ... WHERE {condition}; """:** Câu lệnh này để chèn dữ liệu vào bảng phân mảnh. Chọn **usedid, movieid, rating** từ bảng gốc đã thỏa mãn **condition** đã tính toán và chèn chúng vào table_name.. Việc sử dụng SELECT trực tiếp từ bảng gốc đảm bảo rằng tất cả dữ liệu gốc được xem xét cho quá trình phân mảnh..
+ **_execute_query_pg_with_provided_conn(conn, insert_sql, params):**thực thi câu lệnh INSERT… SELECT Hàm hỗ trợ thực thi truy vấn an toàn.
+ **previous_upper_bound = current_upper;** Cập nhật giới hạn trên của dải hiện tại để nó trở thành giới hạn dưới của dải tiếp theo.
+ Cung cấp thông tin chi tiết về từng phân mảnh được tạo và dải Rating của nó, rất hữu ích cho việc gỡ lỗi và báo cáo. => in qua câu lệnh **print()**.

- **Hoàn tất Giao dịch:**

```
        if not conn.autocommit:
            conn.commit()
    print(f"Range partitioning completed successfully! {n} partitions created.")
except Exception as e:
    print(f"Error during Range_Partition: {e}")
    if not conn.autocommit and conn and not conn.closed and conn.status == psycopg2.extensions.STATUS_IN_ERROR:
        try:
            conn.rollback()
        except psycopg2.Error:
            pass
    raise
```

+ **if not conn.autocommit: conn.commit():** Sau khi vòng lặp hoàn thành (tức là tất cả các bảng phân mảnh đã được tạo và điền dữ liệu), **conn.commit()** sẽ được gọi. **commit()** xác nhận tất cả các thay đổi đã được thực hiện trong khối try này (từ đầu hàm đến thời điểm hiện tại) vào cơ sở dữ liệu một cách vĩnh viễn.
+ Hàm được bao bọc trong khối try-except Nếu có lỗi xảy ra, nó sẽ in thông báo, cố gắng conn.rollback() (để hủy bỏ các thay đổi và đưa database về trạng thái nhất quán) nếu không ở chế độ autocommit, và sau đó raise lại lỗi.
  => Cả hai cùng đảm bảo tính nguyên tử (Atomicity) của giao dịch. Tức là, toàn bộ quá trình phân mảnh (tạo N bảng và điền dữ liệu) sẽ hoặc

thành công hoàn toàn (tất cả các thay đổi được lưu) hoặc thất bại hoàn toàn (không có thay đổi nào được lưu), ngăn chặn tình trạng database bị lỗi hoặc không nhất quán do chỉ một phần của quá trình thành công.

Kết luận: Hàm **rangepartition** là nền tảng của chiến lược phân mảnh dải giá trị thực hiện việc chuyển đổi một bảng dữ liệu lớn thành nhiều bảng con được tổ chức theo các dải giá trị của thuộc tính Rating.

- **Đảm bảo tính đầy đủ (Completeness) và tính rời rạc (Disjointness):** Hàm sử dụng logic chặt chẽ với các điều kiện WHERE (>=, <=, >) và xử lý đặc biệt cho điểm cuối của dải (MAX_RATING_CONST) để đảm bảo mọi bản ghi đều được phân mảnh chính xác và chỉ duy nhất vào một phân mảnh.
- **Quản lý giao dịch mạnh mẽ:** Toàn bộ quá trình tạo và điền dữ liệu vào các phân mảnh được bao bọc trong một giao dịch. commit() khi thành công và rollback() khi có lỗi đảm bảo **tính nguyên tử (Atomicity)**, tức là hoặc tất cả các thay đổi đều được lưu, hoặc không có thay đổi nào được lưu, duy trì tính nhất quán của cơ sở dữ liệu.
- **An toàn và hiệu quả:** Việc sử dụng tham số hóa truy vấn (%s) không chỉ cải thiện bảo mật chống SQL Injection mà còn giúp việc thực thi truy vấn hiệu quả hơn. Đồng thời, việc xóa các bảng cũ bằng DROP TABLE ... CASCADE đảm bảo môi trường sạch sẽ cho mỗi lần chạy.

3.4. RoundRobin_Partition()

Hàm **roundrobinpartition** được thiết kế để phân chia dữ liệu từ một bảng đánh giá (ratings table) lớn thành nhiều bảng nhỏ hơn theo cơ chế "**round-robin**". Cơ chế này đảm bảo rằng các hàng dữ liệu được phân phối đều đặn giữa các bảng đích, mỗi bảng nhận một phần dữ liệu tuần tự.

```
def roundrobinpartition(ratingsTableName, numberOfPartitions, openconnection):
    cursor = openconnection.cursor()
    for i in range(numberOfPartitions):
        table_name = RROBIN_TABLE_PREFIX + str(i)
        cursor.execute("DROP TABLE IF EXISTS " + table_name)
        cursor.execute("CREATE TABLE " + table_name + " (UserID INT, MovieID INT, Rating FLOAT);")
    cursor.execute("SELECT UserID, MovieID, Rating FROM " + ratingsTableName)
    rows = cursor.fetchall()
    for index, row in enumerate(rows):
        target_partition = index % numberOfPartitions
        target_table = RROBIN_TABLE_PREFIX + str(target_partition)
        cursor.execute("INSERT INTO " + target_table + " (UserID, MovieID, Rating) VALUES (%s, %s, %s)", row)
    openconnection.commit()
```

Hình 3.4.a: Toàn bộ mã hàm roundrobinpartition()

Mục đích chính của hàm **roundrobinpartition** là:

- **Phân vùng dữ liệu (Data Partitioning):** Chia một bảng cơ sở dữ liệu lớn thành nhiều bảng nhỏ hơn.

- **Tối ưu hóa hiệu suất:** Việc phân vùng có thể giúp cải thiện hiệu suất truy vấn, quản lý dữ liệu, và bảo trì cơ sở dữ liệu, đặc biệt với các bảng có kích thước rất lớn.
- **Cơ chế Round-Robin:** Dữ liệu được phân phối lần lượt theo thứ tự, đảm bảo mỗi bảng đích nhận một hàng dữ liệu sau mỗi numberOfPartitions hàng từ bảng gốc.

Hàm này nhận ba tham số đầu vào:

- **ratingsTableName**: (Kiểu chuỗi) Tên của bảng nguồn chứa dữ liệu đánh giá gốc. Bảng này dự kiến có các cột UserID, MovieID, và Rating.
- **numberOfPartitions**: (Kiểu số nguyên) Số lượng bảng con (phân vùng) mà đề bài muốn tạo và phân chia dữ liệu vào.
- **openconnection**: (Đối tượng kết nối) Một đối tượng kết nối cơ sở dữ liệu đã được mở sẵn. Hàm sẽ sử dụng kết nối này để thực hiện các thao tác SQL.

Phân tích cách hoạt động của hàm

a. **Khởi Tạo Con Trỏ:**

cursor = openconnection.cursor(): Một con trỏ cơ sở dữ liệu được tạo từ đối tượng kết nối. Con trỏ này dùng để thực thi các lệnh SQL.

```
def roundrobinpartition(ratingsTableName, numberOfPartitions, openconnection):
    cursor = openconnection.cursor()
```

Hình 3.4.b: Phần khởi tạo con trỏ cursor

b. **Chuẩn Bị Các Bảng Phân Vùng:**

```
for i in range(numberOfPartitions):
    table_name = RROBIN_TABLE_PREFIX + str(i)
    cursor.execute("DROP TABLE IF EXISTS " + table_name)
    cursor.execute("CREATE TABLE " + table_name + " (UserID INT, MovieID INT, Rating FLOAT);")
```

Hình 3.4.c: Phần chuẩn bị các bảng phân vùng

- **for i in range(numberOfPartitions)**: Hàm lặp qua một số lần bằng numberOfPartitions để tạo ra các bảng phân vùng.
- **table_name = RROBIN_TABLE_PREFIX + str(i)**: Tên của mỗi bảng phân vùng được tạo bằng cách nối một tiền tố (giả định RROBIN_TABLE_PREFIX là một hằng số được định nghĩa ở đâu đó, ví dụ: "rrobin_part") với chỉ số của vòng lặp (0, 1, 2, ...).
- **cursor.execute("DROP TABLE IF EXISTS " + table_name):** Lệnh SQL này xóa bảng phân vùng nếu nó đã tồn tại. Điều này đảm bảo rằng mỗi khi hàm được chạy, nó sẽ bắt đầu với các bảng phân vùng trống và mới, tránh dữ liệu cũ hoặc xung đột schema.
- **cursor.execute("CREATE TABLE " + table_name + " (UserID INT, MovieID INT, Rating FLOAT);"):** Lệnh SQL này tạo một bảng mới với tên

đã tạo. Bảng này có ba cột: UserID (kiểu số nguyên), MovieID (kiểu số nguyên), và Rating (kiểu số thực).

c. Truy Vấn Dữ Liệu Từ Bảng Nguồn:

```
cursor.execute("SELECT UserID, MovieID, Rating FROM " + ratingsTableName)
rows = cursor.fetchall()
```

Hình 3.4.d: Phần truy vấn dữ liệu từ bảng nguồn (theo đề bài là bảng Ratings)

- **cursor.execute("SELECT UserID, MovieID, Rating FROM " + ratingsTableName)**: Hàm thực thi một truy vấn SELECT để lấy tất cả các hàng (với các cột UserID, MovieID, Rating) từ bảng ratingsTableName (bảng gốc).
- **rows = cursor.fetchall():** Kết quả của truy vấn (tất cả các hàng) được lấy và lưu vào biến rows dưới dạng một danh sách các tuple. Mỗi tuple đại diện cho một hàng dữ liệu.

d. Phân Chia và Chèn Dữ Liệu (Round-Robin):

```
for index, row in enumerate(rows):
    target_partition = index % numberOfPartitions
    target_table = RROBIN_TABLE_PREFIX + str(target_partition)
    cursor.execute("INSERT INTO " + target_table + " (UserID, MovieID, Rating) VALUES (%s, %s, %s)", row)
```

Hình 3.4.e: Phần phân chia và chèn dữ liệu

- **for index, row in enumerate(rows)**: Hàm lặp qua từng hàng (row) trong danh sách rows, cùng với **chỉ số** (index) của hàng đó. Hàm enumerate() rất hữu ích ở đây vì nó cung cấp cả giá trị và vị trí của phần tử.
- **target_partition = index % numberOfPartitions:** Đây là cốt lõi của cơ chế round-robin. Toán tử modulo (%) được sử dụng để xác định bảng phân vùng đích cho hàng hiện tại.
    - Phép toán index % numberOfPartitions sẽ trả về phần dư khi index chia cho numberOfPartitions. Phần dư này sẽ luôn nằm trong khoảng từ 0 đến numberOfPartitions - 1. Điều này đảm bảo rằng các hàng được phân phối đều đặn và tuần tự vào các bảng con.
    - **Ví dụ:**
        - Giả sử numberOfPartitions = 3.
        - **Hàng 0:** 0 % 3 = 0 Chèn vào rrobin_part0
        - **Hàng 1:** 1 % 3 = 1 → Chèn vào rrobin_part1
        - **Hàng 2:** 2 % 3 = 2 → Chèn vào rrobin_part2
        - **Hàng 3:** 3 % 3 = 0 → Chèn vào rrobin_part0 (quay vòng)
        - **Hàng 4:** 4 % 3 = 1 → Chèn vào rrobin_part1
        - **Hàng 5:** 5 % 3 = 2 → Chèn vào rrobin_part2
        - Và cứ thế tiếp tục.
- **target_table = RROBIN_TABLE_PREFIX + str(target_partition):** Xây dựng tên bảng đích dựa trên chỉ số phân vùng đã tính toán.

- **cursor.execute("INSERT INTO " + target_table + " (UserID, MovieID, Rating) VALUES (%s, %s, %s)", row):** Lệnh SQL này chèn hàng hiện tại (row) vào bảng phân vùng đích đã xác định. %s là các placeholder sẽ được thay thế bằng các giá trị trong tuple row (ví dụ: (1, 101, 4.5)).

e. **openconnection.commit():**

- **openconnection.commit():** Sau khi tất cả các hàng đã được xử lý và chèn vào các bảng phân vùng tương ứng, lệnh commit() được gọi trên đối tượng kết nối. Lệnh này lưu các thay đổi vĩnh viễn vào cơ sở dữ liệu. Nếu không có commit(), các thay đổi sẽ không được lưu và sẽ bị mất khi kết nối đóng hoặc chương trình kết thúc.

4. Các Lệnh SQL Được Sử Dụng

Hàm này sử dụng các lệnh SQL cơ bản sau:

- DROP TABLE IF EXISTS [table_name]: Dùng để xóa một bảng hiện có. IF EXISTS ngăn chặn lỗi nếu bảng không tồn tại.
- CREATE TABLE [table_name] (Column1 DataType, Column2 DataType, ...): Dùng để tạo một bảng mới với các cột và kiểu dữ liệu được chỉ định.
- SELECT Column1, Column2, ... FROM [table_name]: Dùng để truy xuất dữ liệu từ một hoặc nhiều cột của một bảng.
- INSERT INTO [table_name] (Column1, Column2, ...) VALUES (Value1, Value2, ...): Dùng để chèn một hàng dữ liệu mới vào một bảng.

Ví dụ mô phỏng cơ chế hoạt động hàm roundrobinpartition

Giả sử ta có một bảng ratings_main với dữ liệu sau:

UserID	MovieID	Rating
1	101	4.0
2	102	3.5
3	103	5.0
4	104	2.0
5	105	4.5
6	106	3.0
7	107	4.0

Và gọi roundrobinpartition("ratings_main", 3, openconnection).

Hàm sẽ thực hiện các bước sau:

1. **Tạo/Xóa các bảng con:**

- DROP TABLE IF EXISTS rrobin_part0; CREATE TABLE rrobin_part0 (UserID INT, MovieID INT, Rating FLOAT);
- DROP TABLE IF EXISTS rrobin_part1; CREATE TABLE rrobin_part1 (UserID INT, MovieID INT, Rating FLOAT);
- DROP TABLE IF EXISTS rrobin_part2; CREATE TABLE rrobin_part2 (UserID INT, MovieID INT, Rating FLOAT);

2. **Đọc dữ liệu từ ratings_main:**
   - rows sẽ chứa: [(1, 101, 4.0), (2, 102, 3.5), (3, 103, 5.0), (4, 104, 2.0), (5, 105, 4.5), (6, 106, 3.0), (7, 107, 4.0)]

3. **Phân phối dữ liệu:**
   - **Hàng 1:** (1, 101, 4.0) (index 0) → 0 % 3 = 0 → INSERT INTO rrobin_part0
   - **Hàng 2:** (2, 102, 3.5) (index 1) → 1 % 3 = 1 → INSERT INTO rrobin_part1
   - **Hàng 3:** (3, 103, 5.0) (index 2) → 2 % 3 = 2 → INSERT INTO rrobin_part2
   - **Hàng 4:** (4, 104, 2.0) (index 3) → 3 % 3 = 0 → INSERT INTO rrobin_part0
   - **Hàng 5:** (5, 105, 4.5) (index 4) → 4 % 3 = 1 → INSERT INTO rrobin_part1
   - **Hàng 6:** (6, 106, 3.0) (index 5) → 5 % 3 = 2 → INSERT INTO rrobin_part2
   - **Hàng 7:** (7, 107, 4.0) (index 6) → 6 % 3 = 0 → INSERT INTO rrobin_part0

Sau khi hoàn tất, các bảng con sẽ có dữ liệu như sau:

Bảng rrobin_part0:

UserID	MovieID	Rating
1	101	4.0
4	104	2.0
7	107	4.0

Bảng rrobin_part1:

UserID	MovieID	Rating
2	102	3.5
5	105	4.5

Bảng rrobin_part2:

UserID	MovieID	Rating
3	103	5.0
6	106	3.0

3.5. RoundRobin_Insert()

1. **Tên hàm: roundrobininsert**

```
def roundrobininsert(ratingsTableName, userid, movieid, rating, openconnection):
```

2. **Mục đích của hàm:**

   Hàm roundrobininsert được sử dụng để:

   - Ghi một bản ghi đánh giá phim (bao gồm: UserID, MovieID, Rating) vào **bảng chính** chứa toàn bộ dữ liệu đánh giá.
   - Đồng thời, chèn bản ghi đó vào một **bảng phân vùng phụ** theo cơ chế **Round-Robin (vòng tròn)**.
   - Mục tiêu là đảm bảo **phân phối đều** dữ liệu vào các bảng phân vùng phụ, từ đó **tối ưu hóa hiệu suất truy vấn** và **giảm tải** cho bảng chính.

3. **Tham số**

Tên tham số	Kiểu dữ liệu	Ý nghĩa
ratingsTableName	str	Tên bảng chính chứa dữ liệu đánh giá
userid	int	ID người dùng thực hiện đánh giá
movieid	int	ID bộ phim được đánh giá
rating	float	Số điểm người dùng đánh giá cho bộ phim
openconnection	psycopg2 connection	Đối tượng kết nối đến cơ sở dữ liệu PostgreSQL

4. **Cách thức hoạt động**

Bước 1: Mở con trỏ cơ sở dữ liệu: Sử dụng openconnection.cursor() để khởi tạo con trỏ truy vấn.

```
cursor = openconnection.cursor()
```

Bước 2: Thêm bản ghi vào bảng chính: Sử dụng câu lệnh INSERT INTO để lưu bản ghi đánh giá vào bảng chính ratingsTableName.

```
cursor.execute(
    f"INSERT INTO {ratingsTableName} (UserID, MovieID, Rating) VALUES (%s, %s, %s);",
    (userid, movieid, rating)
)
```

Bước 3: Lấy số lượng bảng phân vùng round-robin hiện có: Truy vấn pg_tables để đếm số bảng có tên bắt đầu bằng tiền tố RROBIN_TABLE_PREFIX.

```
cursor.execute(
    f"SELECT COUNT(*) FROM pg_tables WHERE tablename LIKE '{RROBIN_TABLE_PREFIX.lower()}%';"
)
num_partitions = cursor.fetchone()[0]
```

Bước 4: Kiểm tra tồn tại phân vùng: Nếu chưa có phân vùng nào thì chỉ ghi vào bảng chính và thoát.

```
if num_partitions == 0:
    openconnection.commit()
    return
```

Bước 5: Tính tổng số bản ghi hiện có trong tất cả các bảng phân vùng: Duyệt từng bảng phân vùng (ví dụ: rrobin0, rrobin1, ...) để tính tổng số bản ghi hiện có.

```
total_rows = 0
for i in range(num_partitions):
    part_table = f"{RROBIN_TABLE_PREFIX}{i}"
    cursor.execute(f"SELECT COUNT(*) FROM {part_table};")
    count = cursor.fetchone()[0]
    total_rows += count
```

Bước 6: Tính chỉ số phân vùng tiếp theo để ghi dữ liệu: Áp dụng công thức: next_partition_index = total_rows % num_partitions.

```
next_partition_index = total_rows % num_partitions
```

Bước 7: Chèn bản ghi vào bảng phân vùng tương ứng: Dựa vào chỉ số phân vùng, tạo tên bảng cần ghi (rrobinX) và chèn bản ghi vào đó.

```
target_partition_table = f"{RROBIN_TABLE_PREFIX}{next_partition_index}"
cursor.execute(
    f"INSERT INTO {target_partition_table} (UserID, MovieID, Rating) VALUES (%s, %s, %s);",
    (userid, movieid, rating)
)
```

Bước 8: Xác nhận và lưu thay đổi: Gọi openconnection.commit() để xác nhận và lưu tất cả thay đổi vào CSDL.

```
openconnection.commit()
```

5. **Các lệnh SQL được sử dụng**
   - INSERT INTO ... VALUES (...) – Thêm một bản ghi mới vào bảng chính và bảng phân vùng.
   - SELECT COUNT(*) FROM ... – Đếm số lượng bản ghi trong bảng.
   - SELECT COUNT(*) FROM pg_tables WHERE tablename LIKE ... – Đếm số bảng phân vùng hiện có bằng cách truy vấn hệ thống bảng của PostgreSQL.

6. **Giả định và lưu ý**
   - Biến toàn cục RROBIN_TABLE_PREFIX đã được định nghĩa trước (ví dụ: "rrobin").
   - Các bảng phân vùng phải được đặt tên theo định dạng chuẩn: rrobin0, rrobin1, rrobin2, ...
   - Hệ thống phân vùng phải được khởi tạo từ trước.
   - Hàm **không xử lý ngoại lệ** như: bảng không tồn tại, lỗi định dạng dữ liệu,...
   - Việc phân vùng dựa trên tổng số bản ghi để đảm bảo phân phối đều giữa các bảng.

7. **Trường hợp sử dụng**

   Khi hệ thống cần lưu trữ đánh giá phim với số lượng lớn và muốn **phân phối đều dữ liệu** vào nhiều bảng nhỏ hơn để tối ưu hóa hiệu năng truy vấn và xử lý. Ứng dụng phù hợp với các hệ thống phân tích dữ liệu lớn hoặc hệ thống khuyến nghị phim.

8. **Ví dụ cụ thể**

   Giả sử:
   - Bảng chính: ratings
   - Có 3 bảng phân vùng: rrobin0, rrobin1, rrobin2

- Biến RROBIN_TABLE_PREFIX = "rrobin"

Gọi hàm:

```
roundrobininsert("ratings", 12, 300, 4.5, openconnection)
```

Hệ thống thực hiện:

1. Ghi bản ghi (12, 300, 4.5) vào bảng ratings.
2. Đếm tổng số bản ghi đang có trong rrobin0, rrobin1, rrobin2.
3. Giả sử tổng cộng có 7 bản ghi, hệ thống tính: 7 % 3 = 1.
4. Bản ghi tiếp theo sẽ được chèn vào rrobin1.
5. Lưu thay đổi với commit().

3.6. Range_Insert()

- Hàm **rangeinsert()** có nhiệm vụ chèn 1 bản ghi dữ liệu mới (UserID, MovieID, Rating) vào bảng CSDL đã được phân mảnh theo dải giá trị trước đó. Bản ghi sẽ được phép chèn cả vào bảng dữ liệu gốc và bảng dữ liệu đã phân mảnh tương ứng. Khi chèn vào bảng gốc để duy trì một cái nhìn toàn diện, tổng quan về dữ liệu, giúp các truy vấn tổng hợp, không phân mảnh vẫn được hoạt động. Chèn vào phân mảnh để hưởng những lợi ích từ hiệu suất truy vấn trên các phân mảnh nhỏ hơn.
- **def rangeinsert(ratingsTableName, userid, movieid, rating, openconnection):**
  + **ratingsTableName: (str)** Tên của bảng gốc (ratings) muốn chèn bản ghi vào.
  + **useid (int):** ID người dùng.
  + **moviedid (int) :** ID bộ phim.
  + **rating (float):** Giá trị đánh giá. Đây là khóa phân mảnh để xác định phân mảnh dài.
  + **openconnection (psycopg2.connection):** Đối tượng kết nối đến CSDL, được cung cấp bên ngoài, không tự mở hay đóng kết nối.

- **Khởi tạo và kiểm tra đầu vào:**

```
def rangeinsert(ratingsTableName, userid, movieid, rating, openconnection):
    actual_base_table_name = ratingsTableName.lower()
    MovieID = movieid
    RatingVal = float(rating)
    UserID = userid
    print(f"Performing Range Insert: UserID={UserID}, MovieID={MovieID}, Rating={RatingVal}")
    conn = openconnection
    if not conn or conn.closed:
        raise Exception("Range_Insert: Invalid or closed database connection provided.")
    try:
```

Minh họa code

  + **actual_base_table_name = ratingsTableName.lower():** Chuyển đổi tên bảng gốc sang chữ thường để đảm bảo tương thích với PostgreSQL.

+ Gán lại các tham số **moviedid, rating, userid** vào các biến cục bộ tương ứng **MovieID, RatingVal, UserID,** đảm bảo giá trị rating luôn là kiểu float.
+ Gán giá trị **conn = openconnection,** đảm bảo conn là hợp lệ và chưa đóng. Nếu không hàm sẽ trả về Exception.

- **Chèn vào bảng gốc**

```
try:
    with conn.cursor() as cur:
        insert_original_sql = f'INSERT INTO {actual_base_table_name} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME}) VALUES (%s, %s, %s);'
        _execute_query_pg_with_provided_conn(conn, insert_original_sql, (UserID, MovieID, RatingVal))
        print(f"Inserted into main table '{actual_base_table_name}'.")
```

Minh họa code

+ Bản ghi được chèn vào bảng gốc **actual_base_table_name:** duy trì một cái nhìn tổng thể về dữ liệu trong một bảng duy nhất, ngay cả khi nó đã phân mảnh.

- **Phân tích chi tiết phần phân mảnh mục tiêu:** xác định rõ bản ghi mới được thêm vào đúng bảng phân mảnh dải giá trị.

```
num_partitions = _count_partitions_with_prefix(conn, RANGE_TABLE_PREFIX)
if num_partitions == 0:
    print("No range partitions found. Skipping insert into partition.")
    if not conn.autocommit: conn.commit()
    return
target_part_table_name = None
if num_partitions == 1:
    target_part_table_name = f"{RANGE_TABLE_PREFIX}0"
    if not (MIN_RATING_CONST <= RatingVal <= MAX_RATING_CONST):
        print(
            f"Warning: Rating {RatingVal} is outside the defined range [{MIN_RATING_CONST}, {MAX_RATING_CONST}]. Skipping insert into partition.")
        if not conn.autocommit: conn.commit()
        return
else:
    range_step = (MAX_RATING_CONST - MIN_RATING_CONST) / num_partitions
    if range_step == 0:
        raise ValueError("Range step is zero during Range_Insert. Check partition configuration.")
    determined_index = -1
    current_lower_bound = MIN_RATING_CONST
    for i in range(num_partitions):
        current_upper_bound = MIN_RATING_CONST + (i + 1) * range_step
        if i == num_partitions - 1:
            current_upper_bound = MAX_RATING_CONST
        if i == 0:
            if RatingVal >= current_lower_bound and RatingVal <= current_upper_bound:
                determined_index = i
                break
        else:
            if RatingVal > current_lower_bound and RatingVal <= current_upper_bound:
                determined_index = i
                break
        current_lower_bound = current_upper_bound
    if determined_index != -1:
        target_part_table_name = f"{RANGE_TABLE_PREFIX}{determined_index}"
    else:
        print(
            f"Warning: Rating {RatingVal} does not fall into any defined range partition. Skipping insert into partition.")
        if not conn.autocommit: conn.commit()
        return
```

Minh họa code

+ **num_partitions=_count_partitions_with_prefix(conn, RANGE_TABLE_PREFIX):** Khởi tạo **num_partitions** để lấy số lượng phân mảnh dải đang tồn tại.
+ Kiểm tra **if num_partitions = 0:** Kiểm tra nếu chưa có phân mảnh dải nào được tạo, sẽ có một thông báo được in ra qua câu lệnh **print**. Và dù không có

phân mảnh nào thì thao tác chèn vẫn sẽ được thực hiện, tức là thao tác chèn vào bảng gốc đã hoàn tất và cần được **commit** => đảm bảo bảng ghi vẫn được lưu vào bảng chính ngay cả khi không có phân mảnh nào chèn vào. Kết thúc thóa hàm sau khi chèn vào bảng gốc(nếu có) và không chèn phân mảnh.

+ **target_part_table_name;** Khởi tạo None là biến để lưu tên bảng phân mảnh mục tiêu.
+ Nếu **num_partitions == 1**. Tức là nếu chỉ có 1 phân mảnh, bản ghi sẽ được chèn vào range_part0.
+ Ngoài ra **if not (MIN_RATING_CONST <= RatingVal <= MAX_RATING_CONST):** dùng để điền kiện kiệu RatingVal có nằm trong dải chung hay không:
    + Nếu nằm ngoài thì sẽ có một cảnh báo được in ra.
    + commit() bản chèn vào bảng gốc.
    + Thoát hàm không chèn vào phân mảnh => giúp xử lý các bản ghi dị thường khi chỉ có 1 phân mảnh.
+ **else**: Khi có nhiều hơn 1 phân mảnh:
    + Ta cần phải tính lại độ rộng dải giống như trong rangepartition: **range_step** => đảm bảo range_step khớp với cấu hình của phân mảnh hiện tại.
    + **determined_index = -1:** Biến để lưu chỉ mục của phân mảnh tìm được.
    + **current_lower_bound = MIN_RATING_CONST:** Biến để theo dõi giới hạn dưới của dải hiện tại trong vòng lặp.
    + **for i in range(num_partitions):** Duyệt qua từng phân mảnh để tìm dải phù hợp.
    + **current_upper_bound = MIN_RATING_CONST + (i + 1) * range_step:** Tính giới hạn trên của dải i.
    + **if i == num_partitions - 1: current_upper_bound = MAX_RATING_CONST:** xử lý sai số dấu phẩy động đảm bảo giá trị 5.0 luôn được bao gồm trong phân mảnh cuối cùng. => đảm bảo tính đầy đủ cho phân mảnh cuối cùng, giống như trong rangepartition.
+ **Logic xác định dải**: **if i == 0: … else: …**
    + Logic này giống hệt logic trong **rangepartition**. => Đảm bảo bản ghi mới được chèn vào cùng một phân mảnh mà nó sẽ thuộc về nếu được phân mảnh từ đầu. Nếu logic này khác, sẽ dẫn đến vi phạm tính rời rạc hoặc đầy đủ.
    + **if RatingVal >= current_lower_bound and RatingVal <= current_upper_bound (** với **i == 0):** Kiểm tra nếu Rating nằm trong dải đầu tiên, bao gồm cả hai đầu mút.
    + **if RatingVal > current_lower_bound and RatingVal <= current_upper_bound: (else):** Kiểm tra nếu Rating nằm trong các dải tiếp theo. Sử dụng > cho giới hạn dưới của các dải sau để đảm bảo tính

rời rạc. Bản ghi RatingVal tại điểm ranh giới sẽ chỉ thuộc về phân mảnh có điều kiện <=.

    + **determined_index = i; break** Nếu tìm thấy dải khớp, lưu chỉ mục và thoát vòng lặp.

+ **current_lower_bound = current_upper_bound** Cập nhật giới hạn dưới cho lần lặp tiếp theo.
+ **if determined_index != -1:** Kiểm tra xem có thấy phân mảnh mục tiêu hay không
    + target_part_table_name = f"{RANGE_TABLE_PREFIX}{determined_index}": Xây dựng tên bảng phân mảnh mục tiêu.
    + else: Nếu không tìm thấy in ra cảnh báo, commit bản chèn vào bảng gốc. Sau đó return.

- **Chèn vào phân mảnh mục tiêu**

```
if target_part_table_name:
    print(f"Data will be inserted into partition '{target_part_table_name}' for Rating {RatingVal}.")
    insert_partition_sql = f'INSERT INTO {target_part_table_name} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME}) VALUES (%s, %s, %s);'
    _execute_query_pg_with_provided_conn(conn, insert_partition_sql, (UserID, MovieID, RatingVal))
    print(f"Successfully inserted into partition '{target_part_table_name}'.")

if not conn.autocommit:
    conn.commit()
```

Minh họa code

- Nếu **target_part_table_name** đã được xác định, bản ghi sẽ được chèn vào bảng phân mảnh đó. Sẽ có thông báo mỗi khi chèn 1 phân mảnh cụ thể.
- **insert_partition_sql = …** xây dựng câu lệnh INSERT cho bảng phân mảnh.
- **_execute_query_pg_with_provided_conn(() t**hực thi truy vấn chèn vào phân mảnh, sử dụng các truy vấn tham số hóa.
- Sau khi hoàn tất sẽ có thông báo xác nhận chèn thành thông vào phân mảnh.
- **if not conn.autocommit: conn.commit(): :** Sau khi cả thao tác chèn vào bảng gốc và thao tác chèn vào phân mảnh đã hoàn tất, **commit()** được gọi để lưu vĩnh viễn tất cả các thay đổi của giao dịch vào cơ sở dữ liệu. Điều này đảm bảo **tính nguyên tử (Atomicity)** của thao tác chèn: hoặc cả hai thao tác đều thành công và được lưu, hoặc cả hai đều bị hủy bỏ nếu có lỗi.

```
except Exception as e:
    print(f"Error during Range_Insert: {e}")
    if not conn.autocommit and conn and not conn.closed and conn.status == psycopg2.extensions.STATUS_IN_ERROR:
        try:
            conn.rollback()
        except psycopg2.Error:
            pass
    raise
```

Xử lý ngoại lệ

- except Exception as e: Bắt bất kỳ ngoại lệ nào xảy ra trong hàm.
    + print() In thông báo lỗi.
    + **if not conn.autocommit and conn and not conn.closed and conn.status == psycopg2.extensions.STATUS_IN_ERROR:**: Logic rollback mạnh mẽ.
        ■ Chỉ rollback nếu kết nối không ở chế độ autocommit (nơi mỗi lệnh được tự động commit).
        ■ Đảm bảo kết nối vẫn còn hoạt động và đang ở trạng thái lỗi (STATUS_IN_ERROR).
- try...except psycopg2.Error:: Bắt lỗi có thể xảy ra trong chính quá trình rollback.
- raise: Ném lại ngoại lệ. =>Thông báo cho hàm gọi (tester của thầy) rằng thao tác chèn đã thất bại.

Hàm **rangeinsert** đóng vai trò then chốt trong việc duy trì tính nhất quán và hiệu quả của hệ thống cơ sở dữ liệu đã được phân mảnh theo dải giá trị. Nó không chỉ chèn một bản ghi mới vào bảng gốc để đảm bảo dữ liệu toàn vẹn, mà còn xác định và chèn bản ghi đó vào đúng bảng phân mảnh dải giá trị mà nó thuộc về.
Việc triển khai hàm này đặc biệt chú trọng đến:

- **Tính chính xác:** Sử dụng cùng logic xác định dải giá trị (>=, <=, >) và xử lý các trường hợp biên (MAX_RATING_CONST) như hàm rangepartition, đảm bảo bản ghi mới được đặt vào đúng phân mảnh theo quy tắc đầy đủ và rời rạc.
- **Tính nhất quán:** Thực hiện cả hai thao tác chèn (vào bảng gốc và vào phân mảnh) trong cùng một giao dịch, đảm bảo tính nguyên tử (Atomicity) của thao tác. Nếu có lỗi, mọi thay đổi sẽ được rollback, giữ cho database ở trạng thái nhất quán.
- **Xử lý lỗi và dữ liệu ngoài dải:** Cung cấp các thông báo cảnh báo và xử lý linh hoạt cho các bản ghi có giá trị Rating không nằm trong bất kỳ dải phân mảnh đã định nghĩa nào.
- **Bảo mật:** Sử dụng tham số hóa truy vấn để ngăn chặn các rủi ro bảo mật như SQL Injection.

Nhìn chung, **rangeinsert** là một thành phần thiết yếu, đảm bảo rằng dữ liệu mới được tích hợp một cách chính xác và an toàn vào cấu trúc cơ sở dữ liệu đã được phân mảnh, tối ưu hóa cho các truy vấn dựa trên dải giá trị.'

3.7. Kiểm thử kết quả (FILE DATA: ratings.dat)

a. **Kết quả sau chạy hàm LoadRatings():**

```
A database named "dds_assgn1" already exists
loadratings function pass!
```

ratings

1-500 of 501+ Tx: Auto DDI

WHERE ORDER BY

	userid	movieid	rating
1	1	122	5
2	1	185	5
3	1	231	5
4	1	292	5
5	1	316	5
6	1	329	5
7	1	355	5
8	1	356	5
9	1	362	5
10	1	364	5
11	1	370	5
12	1	377	5
13	1	420	5
14	1	466	5
15	1	480	5
16	1	520	5
17	1	539	5

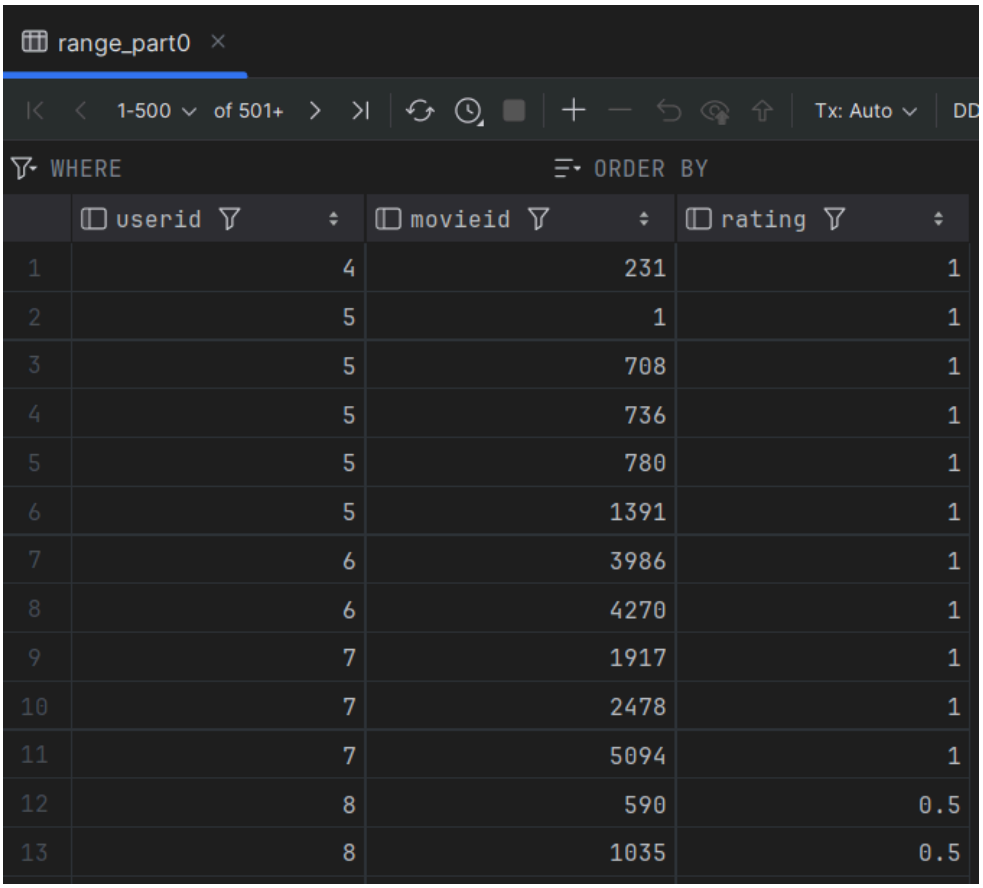
Dữ liệu gốc sau khi được xử lý

b. Kết quả sau chạy hàm Range_Partition():

```
A database named "dds_assgn1" already exists
loadratings function pass!
Partitioning table 'ratings' into 5 range partitions (PostgreSQL)...
Created and populated partition 'range_part0' (Ratings: 0.00 to 1.00).
Created and populated partition 'range_part1' (Ratings: 1.00 to 2.00).
Created and populated partition 'range_part2' (Ratings: 2.00 to 3.00).
Created and populated partition 'range_part3' (Ratings: 3.00 to 4.00).
Created and populated partition 'range_part4' (Ratings: 4.00 to 5.00).
Range partitioning completed successfully! 5 partitions created.
rangepartition function pass!
```

Các thông báo khi thực hiện phân mảnh theo dải giá trị của hàm rangepartition()

:

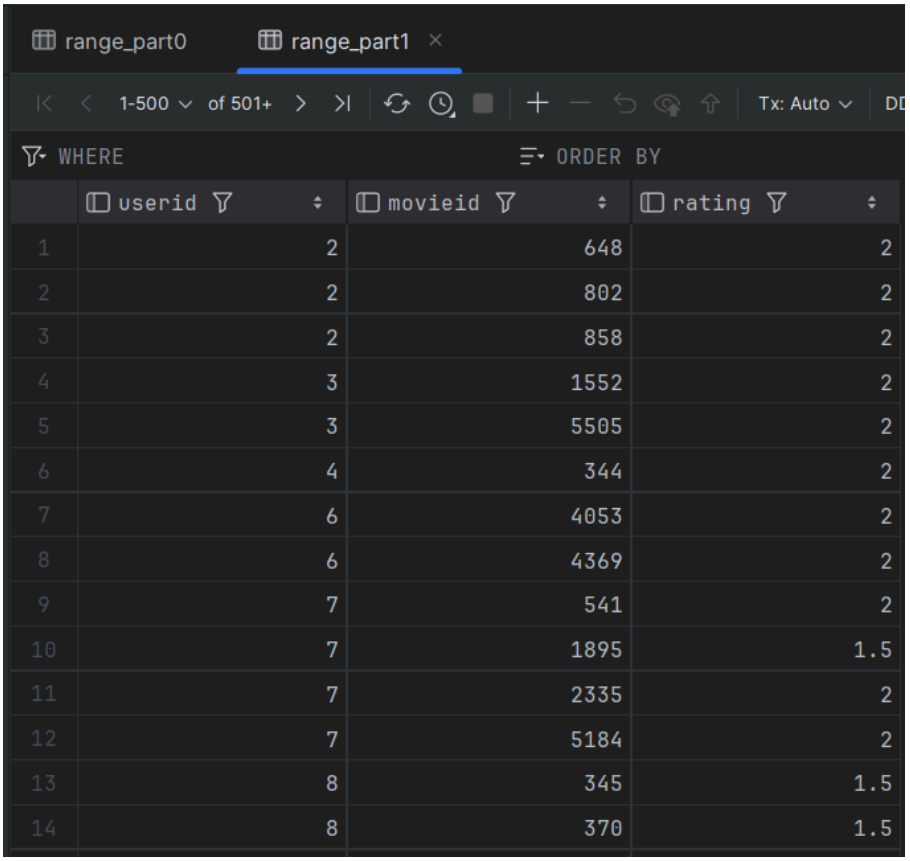

range_part0

1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

	userid	movieid	rating
1	4	231	1
2	5	1	1
3	5	708	1
4	5	736	1
5	5	780	1
6	5	1391	1
7	6	3986	1
8	6	4270	1
9	7	1917	1
10	7	2478	1
11	7	5094	1
12	8	590	0.5
13	8	1035	0.5

Bảng range_part0 phân mảnh trong dải rangting giá trị 0 - 1


range_part0 range_part1

1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

	userid	movieid	rating
1	2	648	2
2	2	802	2
3	2	858	2
4	3	1552	2
5	3	5505	2
6	4	344	2
7	6	4053	2
8	6	4369	2
9	7	541	2
10	7	1895	1.5
11	7	2335	2
12	7	5184	2
13	8	345	1.5
14	8	370	1.5

Bảng range_part1 phân mảnh trong dải rangting giá trị 1-2

range_part0 range_part1 range_part2

1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

	userid	movieid	rating
1	2	151	3
2	2	376	3
3	2	539	3
4	2	719	3
5	2	733	3
6	2	736	3
7	2	780	3
8	2	786	3
9	2	1049	3
10	2	1073	3
11	2	1356	3
12	2	1391	3
13	2	1544	3
14	3	1288	3
15	3	5299	3
16	3	6287	3

Bảng range_part2 phân mảnh trong dải rangting giá trị 2-3

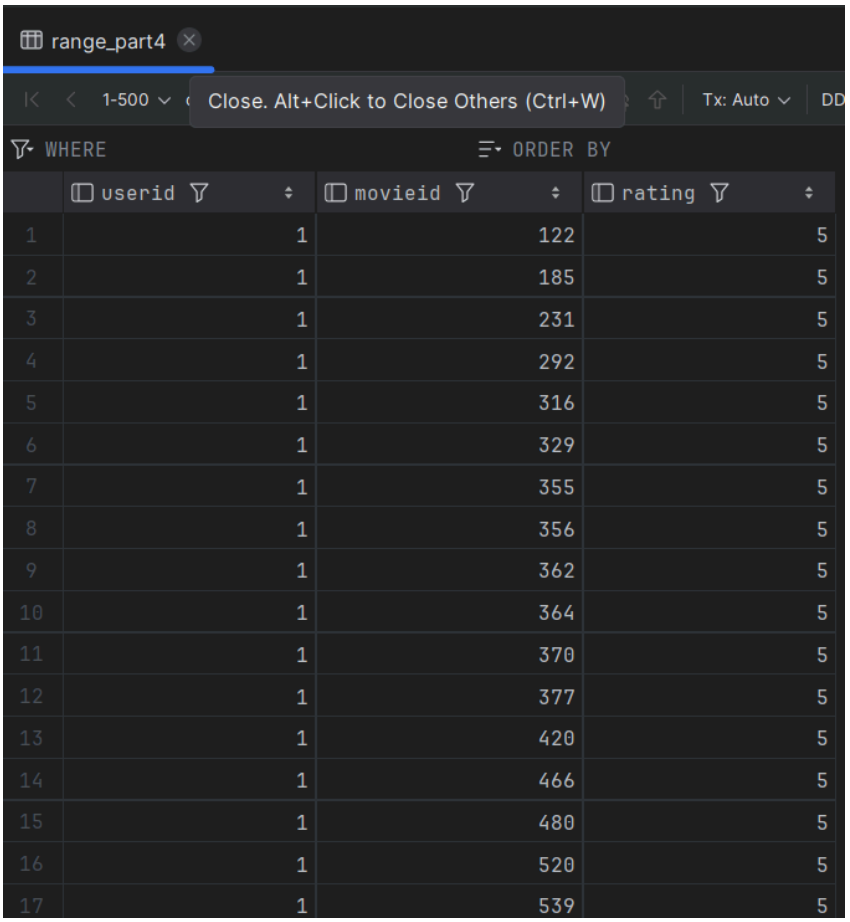
range_part0 range_part1 range_part2 range_part3

1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

	userid	movieid	rating
1	2	1210	4
2	3	590	3.5
3	3	1148	4
4	3	1246	4
5	3	1252	4
6	3	1276	3.5
7	3	1408	3.5
8	3	3408	4
9	3	4535	4
10	3	4677	4
11	3	5952	3.5
12	3	6377	4
13	3	7153	4
14	3	7155	3.5

Bảng range_part3 phân mảnh trong dải rangting giá trị 3-4



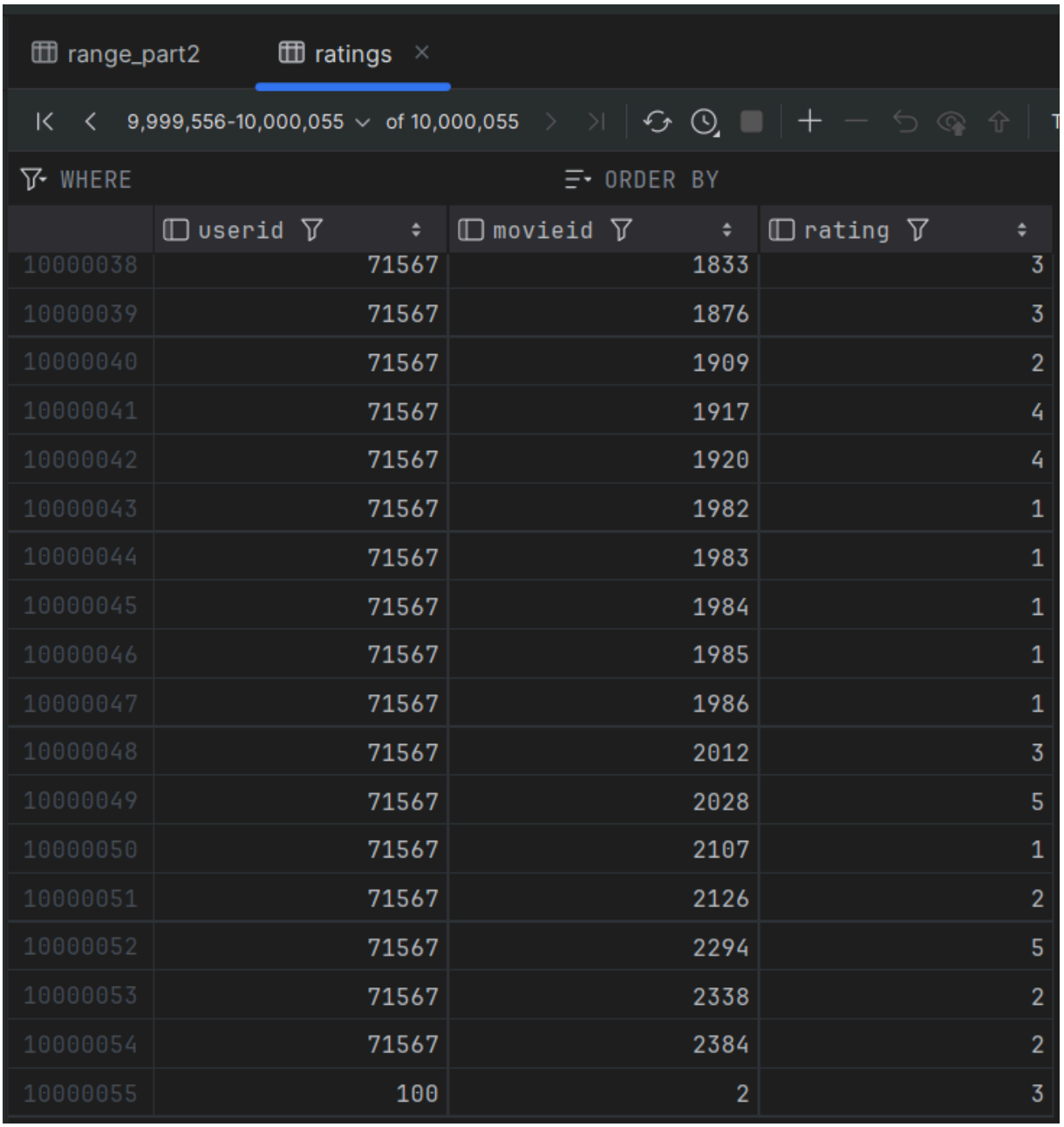
	userid	movieid	rating
1	1	122	5
2	1	185	5
3	1	231	5
4	1	292	5
5	1	316	5
6	1	329	5
7	1	355	5
8	1	356	5
9	1	362	5
10	1	364	5
11	1	370	5
12	1	377	5
13	1	420	5
14	1	466	5
15	1	480	5
16	1	520	5
17	1	539	5

Bảng range_part4 phân mảnh trong dải rangting giá trị 4-5

c. Kết quả sau khi chạy hàm range_insert:

```
Performing Range Insert: UserID=100, MovieID=2, Rating=3.0
Inserted into main table 'ratings'.
Data will be inserted into partition 'range_part2' for Rating 3.0.
Successfully inserted into partition 'range_part2'.
rangeinsert function pass!
```

Thông báo khi xử lý hàm rangeinsert()



	userid	movieid	rating
10000038	71567	1833	3
10000039	71567	1876	3
10000040	71567	1909	2
10000041	71567	1917	4
10000042	71567	1920	4
10000043	71567	1982	1
10000044	71567	1983	1
10000045	71567	1984	1
10000046	71567	1985	1
10000047	71567	1986	1
10000048	71567	2012	3
10000049	71567	2028	5
10000050	71567	2107	1
10000051	71567	2126	2
10000052	71567	2294	5
10000053	71567	2338	2
10000054	71567	2384	2
10000055	100	2	3

Bảng phân mảnh gốc được chèn dữ liệu

range_part2

2,726,356-2,726,855 of 2,726,855

WHERE ORDER BY

	userid	movieid	rating
2726838	71566	305	3
2726839	71566	316	3
2726840	71566	317	3
2726841	71566	329	3
2726842	71566	349	3
2726843	71566	434	3
2726844	71567	32	3
2726845	71567	256	3
2726846	71567	1396	3
2726847	71567	1580	3
2726848	71567	1584	3
2726849	71567	1690	3
2726850	71567	1748	3
2726851	71567	1769	3
2726852	71567	1833	3
2726853	71567	1876	3
2726854	71567	2012	3
2726855	100	2	3

Bảng phân mảnh được chèn dữ liệu

d. Kết quả sau khi chạy hàm roundrobinpartition():

```
A database named "dds_assgn1" already exists
roundrobinpartition function pass!
```

Thông báo xử lý hàm

Danh sách các bảng sau khi phân mảnh vòng tròn:

rrobin_part1

1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

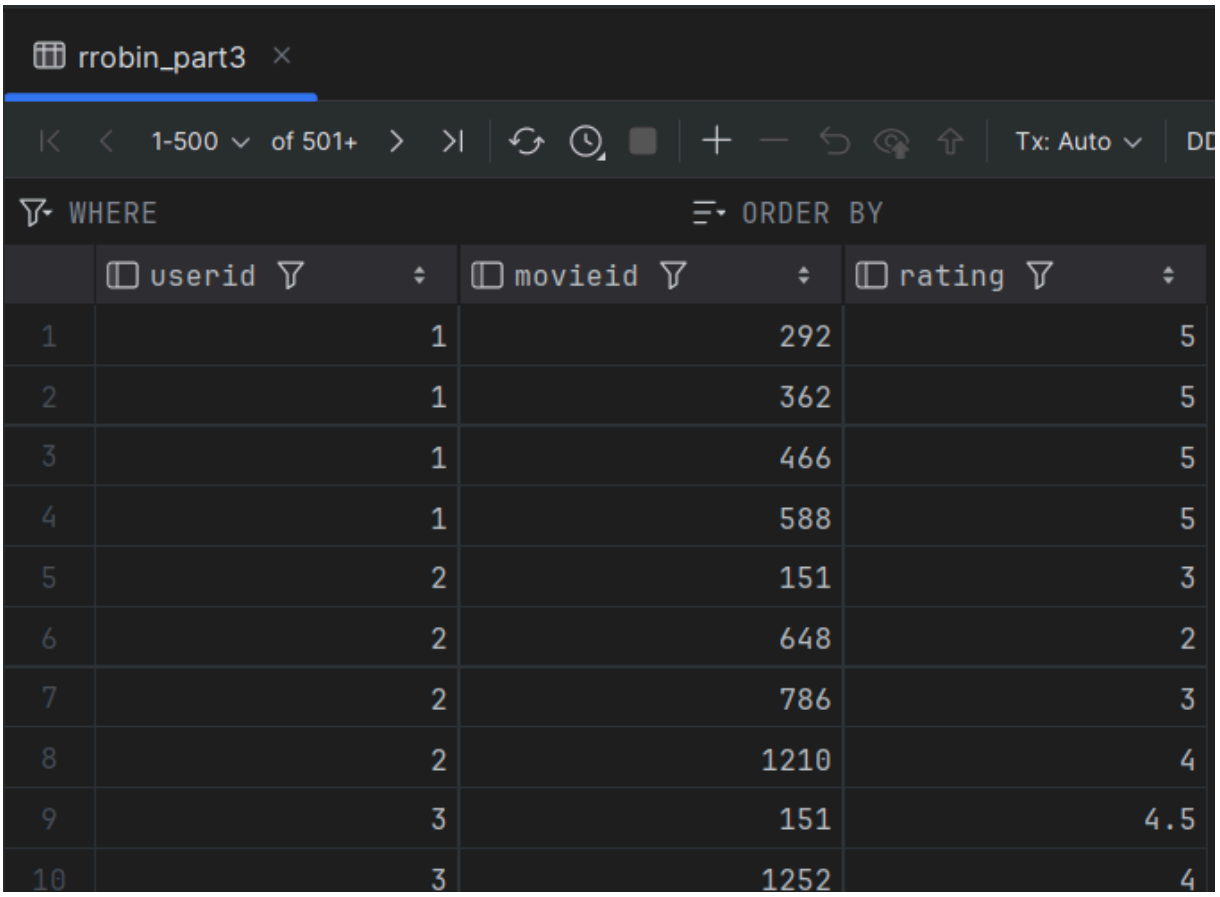
	userid	movieid	rating
1	1	185	5
2	1	355	5
3	1	377	5
4	1	539	5
5	1	616	5
6	2	539	3
7	2	736	3
8	2	1049	3
9	2	1544	3
10	3	1148	4

rrobin_part1 rrobin_part2

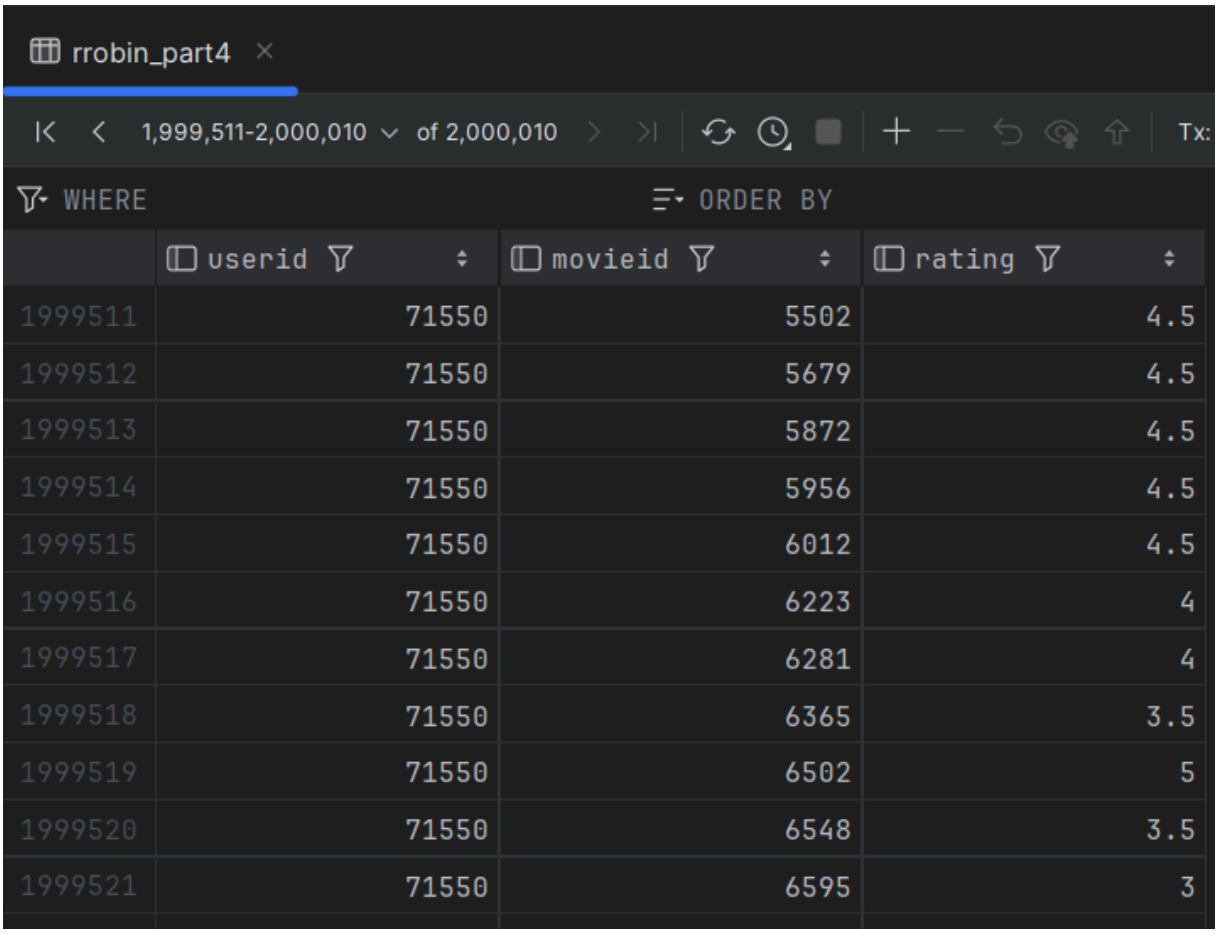
1-500 of 501+ Tx: Auto DD

WHERE ORDER BY

	userid	movieid	rating
1	1	231	5
2	1	356	5
3	1	420	5
4	1	586	5
5	2	110	5
6	2	590	5
7	2	780	3
8	2	1073	3
9	3	110	4.5
10	3	1246	4
11	3	1552	2



	userid	movieid	rating
1	1	292	5
2	1	362	5
3	1	466	5
4	1	588	5
5	2	151	3
6	2	648	2
7	2	786	3
8	2	1210	4
9	3	151	4.5
10	3	1252	4



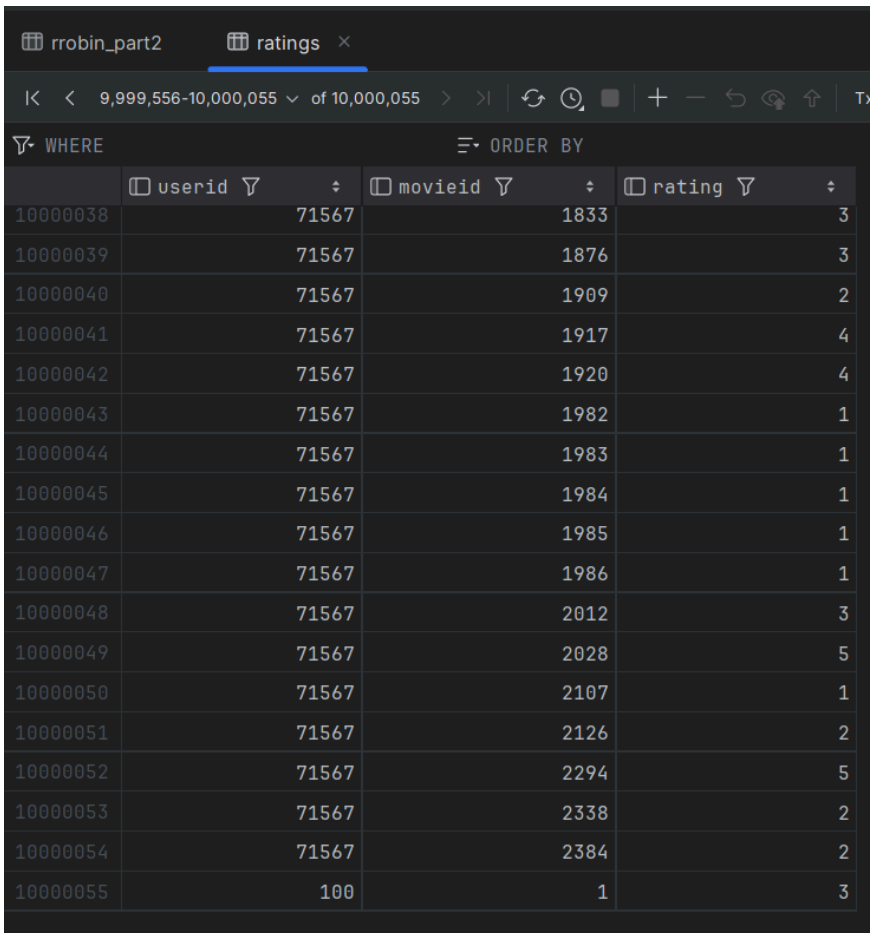
	userid	movieid	rating
1999511	71550	5502	4.5
1999512	71550	5679	4.5
1999513	71550	5872	4.5
1999514	71550	5956	4.5
1999515	71550	6012	4.5
1999516	71550	6223	4
1999517	71550	6281	4
1999518	71550	6365	3.5
1999519	71550	6502	5
1999520	71550	6548	3.5
1999521	71550	6595	3

e. Kết quả sau khi chạy hàm roundrobininsert():

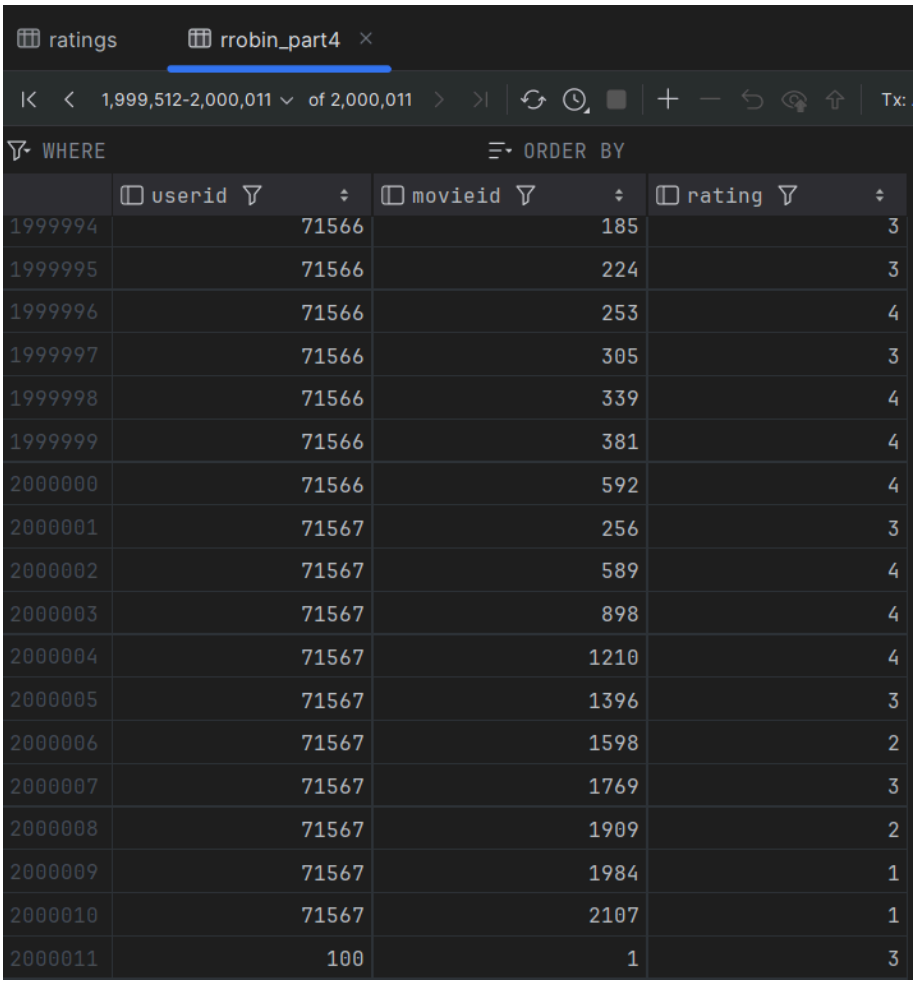
```
A database named "dds_assgn1" already exists
roundrobininsert function pass!
```

Thông báo xử lý hàm

Danh sách các bảng dữ liệu sau khi xử lý hàm



	userid	movieid	rating
10000038	71567	1833	3
10000039	71567	1876	3
10000040	71567	1909	2
10000041	71567	1917	4
10000042	71567	1920	4
10000043	71567	1982	1
10000044	71567	1983	1
10000045	71567	1984	1
10000046	71567	1985	1
10000047	71567	1986	1
10000048	71567	2012	3
10000049	71567	2028	5
10000050	71567	2107	1
10000051	71567	2126	2
10000052	71567	2294	5
10000053	71567	2338	2
10000054	71567	2384	2
10000055	100	1	3



	userid	movieid	rating
1999994	71566	185	3
1999995	71566	224	3
1999996	71566	253	4
1999997	71566	305	3
1999998	71566	339	4
1999999	71566	381	4
2000000	71566	592	4
2000001	71567	256	3
2000002	71567	589	4
2000003	71567	898	4
2000004	71567	1210	4
2000005	71567	1396	3
2000006	71567	1598	2
2000007	71567	1769	3
2000008	71567	1909	2
2000009	71567	1984	1
2000010	71567	2107	1
2000011	100	1	3

PHÂN CHIA CÔNG VIỆC

Nhiệm vụ	Họ tên
Xử lý CSDL, xây dựng các hàm liên quan	Hà Mạnh Dũng
Xây dựng hàm LoadLoadRatings()	Hà Mạnh Dũng
Xây dựng hàm Range_Partition()	Phan Thị Hồng Huế
Xây dựng hàm RoundRobin_Partition()	Hà Mạnh Dũng
Xây dựng hàm RoundRobin_Insert()	Phạm Thị Hương Nhài
Xây dựng hàm Range_Insert()	Phan Thị Hồng Huế
Kiểm thử kết quả	Hà Mạnh Dũng Phan Thị Hồng Huế Phạm Thị Hương Nhài
Viết báo cáo	Hà Mạnh Dũng Phan Thị Hồng Huế Phạm Thị Hương Nhài