HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN 1

BÁO CÁO BÀI TẬP LỚN
HỌC PHẦN: CƠ SỞ DỮ LIỆU PHÂN TÁN
MÃ HỌC PHẦN: INT14148

ĐỀ TÀI: Mô phỏng các phương pháp phân mảnh dữ liệu

Lớp học phần : N09
Nhóm : 04
Sinh viên thực hiện :

1. Nguyễn Đức Anh : B22DCCN027
2. Phạm Quốc Anh : B22DCCN040
3. Chu Quang Vũ : B22DCCN911

Giảng viên hướng dẫn: TS.Kim Ngọc Bách

HÀ NỘI 2025

PHÂN CÔNG NHIỆM VỤ NHÓM THỰC HIỆN

TT	Công việc / Nhiệm vụ	SV thực hiện	Thời hạn hoàn thành
1	Loadratings + Rangepartition + CheckCode + Test	Nguyễn Đức Anh	9/6/2025
2	RangeInsert + RoundRobinPartition + Báo Cáo + Test	Phạm Quốc Anh	9/6/2025
3	RoundRobinInsert + CheckCode + Test	Chu Quang Vũ	9/6/2025

NHÓM THỰC HIỆN TỰ ĐÁNH GIÁ

TT	SV thực hiện	Thái độ tham gia	Mức hoàn thành CV	Kỹ năng giao tiếp	Kỹ năng hợp tác
1	Nguyễn Đức Anh	5	5	5	5
2	Phạm Quốc Anh	5	5	5	5
3	Chu Quang Vũ	5	5	5	5

Ghi chú:

- Thái độ tham gia: Đánh giá điểm thái độ tham gia công việc chung của nhóm (từ 0: không tham gia, đến 5: chủ động, tích cực).
- Mức hoàn thành CV: Đánh giá điểm mức độ hoàn thành công việc được giao (từ 0: không hoàn thành, đến 5: hoàn thành xuất sắc).
- Kỹ năng giao tiếp: Đánh giá điểm khả năng tương tác, giao tiếp trong nhóm (từ 0: không hoặc giao tiếp rất yếu, đến 5: giao tiếp xuất sắc).
- Kỹ năng hợp tác: Đánh giá điểm khả năng hợp tác, hỗ trợ lẫn nhau, giải quyết mâu thuẫn, xung đột

MỤC LỤC

DANH MỤC CÁC HÌNH VẼ

DANH MỤC CÁC TỪ VIẾT TẮT

Từ viết tắt	**Thuật ngữ tiếng Anh/Giải thích**	**Thuật ngữ tiếng Việt/Giải thích**
DB	Database	Cơ sở dữ liệu
rrobin	Round Robin	Vòng tròng

MỞ ĐẦU

Để hoàn thành báo cáo môn học **Cơ sở dữ liệu phân tán** này, chúng em đã nhận được sự hỗ trợ và hướng dẫn vô cùng quý báu từ các thầy cô tại Khoa Công Nghệ Thông Tin.

Đặc biệt, chúng em xin gửi lời cảm ơn chân thành và sâu sắc nhất đến **Thầy Kim Ngọc Bách** – Giảng viên môn Cơ sở dữ liệu phân tán, người đã trực tiếp hướng dẫn chúng em từ những bước đầu tiên trong việc nghiên cứu tài liệu, đến việc phân tích, xây dựng và hoàn thiện nội dung báo cáo. Những kiến thức chuyên môn vững vàng, kinh nghiệm thực tiễn phong phú và sự tận tâm của thầy đã giúp chúng em vượt qua nhiều khó khăn, tiếp thu được những kiến thức mới mẻ và hoàn thành tốt nhiệm vụ học tập.

Kết quả báo cáo là quá trình học tập và làm việc của nhóm em. Chúng em xin cam đoan toàn bộ nội dung và kết quả sản phẩm đều trung thực và chưa từng xuất hiện trên các diễn đàn Internet nào trước đó hay copy từ nguồn nào khác. Trong trường hợp vi phạm , chúng em xin chịu trách nhiệm với nội dung của bài báo cáo này .

Cuối cùng, chúng em xin kính chúc thầy sức khỏe, hạnh phúc và luôn thành công trong công tác giảng dạy cũng như trong cuộc sống.

CHƯƠNG 1. GIỚI THIỆU

1.1 : Nhiệm vụ yêu cầu

Bài tập lớn này nhằm mô phỏng các phương pháp phân mảnh ngang dữ liệu (horizontal partitioning) bao gồm:

- Phân mảnh theo khoảng (Range Partitioning)
- Phân mảnh vòng tròn (Round Robin Partitioning)

Nhiệm vụ yêu cầu phải tạo một tập các hàm Python để tải dữ liệu đầu vào một bảng quan hệ, phân mảnh bảng này bằng các phương pháp phân mảnh ngang khác nhau và chèn các bộ dữ liệu mới vào đúng phân mảnh.

1.2 : Dữ liệu đầu vào :

Dữ liệu đầu vào là một tập dữ liệu đánh giá phim được thu thập từ trang web MovieLens (http://movielens.org). Dữ liệu thô có trong tệp ratings.dat.

Tệp ratings.dat chứa 10 triệu đánh giá và 100.000 thẻ được áp dụng cho 10.000 bộ phim bởi 72.000 người dùng. Mỗi dòng trong tệp đại diện cho một đánh giá của một người dùng với một bộ phim, và có định dạng như sau: UserID::MovieID::Rating::Timestamp

Các đánh giá được thực hiện trên thang điểm 5 sao, có thể chia nửa sao. Dấu thời gian (Timestamp) là số giây kể từ nửa đêm UTC ngày 1 tháng 1 năm 1970. Ví dụ nội dung tệp:
1::122::5::838985046
1::185::5::838983525
1::231::5::838983392

1.3 : Công cụ sử dụng

+ IDE lập trình : Visual Code
+ Máy Ảo Sử dụng : Ubuntu 24.04.2 và Window 10 (Để test)
+ Cơ sở dữ liệu : PostgreSQL
+ Cấu hình : Python : 3.12.3

1.4 : Quy trình thực hiện

- Tải về máy ảo có môi trường giống với máy chấm điểm (Các thành viên đã sử dụng Ubuntu và Window để test trên 2 hệ điều hành) và Cấu hình máy ảo Python 3.12.3
- Cài đặt cơ sở dữ liệu PostgreSQL
- Tải tệp rating.dat từ trang (http://movielens.org).
- Xây dựng Các hàm (Chi tiết phần 2)
- Kiểm tra code
- Test code trên 2 hệ điều hành ubuntu và window
- Đánh giá kết quả và viết báo cáo

1.5 : Giải quyết vấn đề

- Chi tiết có trong phân tích từng hàm xây dựng
- Tổng quan

- Xây dựng hàm kết nối với cơ sở dữ liệu
- Xây dựng hàm loadRating()
    + Dùng lện COPY : Lệnh copy của PostgreSQL để load dữ liệu bulk trực tiếp từ file
    + Chia dữ liệu thành các bathch nhỏ (150.000 dòng) để tránh việc giữ toàn bộ file lớn trong RAM mà vẫn tận dụng được sức mạnh ghi bulk
    + Dùng StringIO để tạm lưu dữ liệu chuẩn hóa trong RAM, thay vì phải ghi ra file tạm => Tiết kiệm I/O đĩa
- Xây dựng hàm rangepartition
    + Tính bước phân chia step : nếu có n phân vùng thì môi phân vùng tương ứng với khoảng rộng step = 5.0/n
    + Tạo các bảng phân vùng : vd range_part0 , range_part1
    + Chèn dữ liệu vào từng phân vùng : gọi hàm insert_partition
    + Xử lí lỗi và đóng kết nối
- Xây dựng hàm roundrobinpartition
    + Khởi tạo chỉ số vòng lặp round robin
    + Tạo bảng tạm : Sử dụng câu lênh SQL tạo bảng tạm và đánh số theo thứ tự userid . Việc này giúp xác định vị trí của mỗi dòng trong tập dữ liệu để phân chia round robin
    + Tạo các bảng phân vùng con với các tên lần lượt là rrobin_part0 , rrobin_part1 … có cấu trúc giống bảng gốc gồm các cột (userid , movieid , rating)
    + Phân chia dữ liệu vào các bảng con theo RoundRobin
- Xây dựng hàm rangeinsert

➢ Đối với bảng ratings gốc : Ta thực hiện truy vấn SQL để thêm vào các cột tương ứng các giá trị UserID , ItemID , Rating

➢ Đối với bảng Phân mảnh ta làm theo các bước :

+ Làm tương tự với hàm insert_partition , ta xác định được khoảng phân vùng trong mỗi phân mảnh :

delta = MAX_RATING_SCALE / numberofparrtitions

(hay 5/số phân mảnh)

+ Xác định bảng con phù hợp : ***index = int (rating / delta)***
+ Trường hợp ngoại lệ : Nếu rating nằm ngay ranh giới như 1.0 , 2.0 ... và index > 0 thì index trừ đi 1 sẽ ra bảng con được chọn

Thực hiện chèn : Dùng lệnh truy vấn SQL

- Xây dựng hàm roundrobininsert

➢ Đối với bảng ratings gốc : Ta thực hiện truy vấn SQL để thêm vào các cột tương ứng các giá trị UserID , ItemID , Rating

➢ Đối với bảng ratings phân mảnh : ta làm theo các bước :

+ Xác định số phân vùng hiện tại : dùng SELECT COUNT(*)

+ Xác định phân vùng cần ghi dữ liệu : tính số dư của phép chia (current_index) % numberofparrtitions
+ Thực hiện lệnh truy vấn SQL

CHƯƠNG 2. MÔ TẢ CHI TIẾT CÁC HÀM

2.1 : Các hàm phụ trợ

2.1.1 : def getopenconnection()

- **Mục đích** : Thiết lập một kết nốt đến cơ sở dữ liệu PostgreSQL bằng cách sử dụng thư viện psycopg2 trong Python
- **Tác dụng** :
  - Đơn giản hóa việc kết nối : Thay vì phải viết lại các thông số kết nối mỗi khi cần, người dùng chỉ cần gọi hàm này
  - Tái sử dụng mã : Mã kết nối được tập tring trong một hàm, giúp dễ dàng tái sử dụng ở nhiều nơi trong ứng dụng
  - Tăng tính bảo mật/linh hoạt: Cho phép lấy host và port từ biến môi trường, giúp ứng dụng linh hoạt hơn khi triên khai trên các môi trường khác nhau và tránh hardcoding các thông tin nhạy cảm trực tiếp vào code
- Chi tiết :

```
def getopenconnection(dbname='postgres'):
    '''
    Connect to database 'dbname' through unix socket
    '''
    return psycopg2.connect(
        dbname=dbname,
        user=os.getenv("USER"),
        password=os.getenv("PASSWORD"),
        host=os.getenv("HOST"),
        port=os.getenv("PORT")
    )
```

Hình 1 : Hàm getopenconnection()

+ Tham số đầu vào : dbname = ‘postgres’ là tên cơ sở dữ liệu muốn kết nối (mặc định)
+ Psycopg2.connect() : Khởi tạo đối tượng kết nối cơ sở dữ liệu
+ Truyền các tham số kết nối :
  - dbname = dbname (tên cơ sở dữ liệu được sử dụng)
  - user = os.getenv(“USER”) : Lấy username từ biến môi trường USER

- password= os.getenv(“PASSWORD”) : Lấy password từ biến môi trường PASSWORD
- host = os.getenv(“HOST”) : địa chỉ máy chủ host của cơ sở dữ liệu
- port = os.getenv(“PORT”) : lấy port từ biến môi trường PORT

+ Cuối cùng sẽ trả về đối tượng kết nối .

2.1.2 : def creat_db(dbname)

- **Mục đích** : Tạo mới cơ sở dữ liệu (nếu chưa tồn tại trong PostgreSQL)
- **Ý tưởng xây dựng** :
  + Kết nối đến cơ sở dữ liệu mặc định (postgres)
  + Thiết lập chế độ *AUTOCOMMIT* để thực hiện lệnh tạo cơ sở dữ liệu
  + Kiểm tra xem cơ sở dữ liệu dbname đã tồn tại chưa (dùng truy vấn pg_catalog.pg_database)
  + Nếu chưa tồn tại thì tạo mới
  + Nếu tồn tại in thông báo “already exists”
  + Đảm bảo đóng kết nối và xử lí ngoại lệ đúng cách
- **Chi tiết** :

```
def create_db(dbname):
    """
    We create a DB by connecting to the default user and database of Postgres
    The function first checks if an existing database exists for a given name, else creates it.
    :return:None
    """
    conn = None
    cur = None
    try:
        conn = getopenconnection(dbname='postgres')
        conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
        cur = conn.cursor()

        # Check if database exists (proper parameterized query)
        cur.execute("SELECT 1 FROM pg_catalog.pg_database WHERE datname = %s", (dbname,))

        if not cur.fetchone():
            cur.execute(SQL("CREATE DATABASE {}").format(Identifier(dbname)))
            print(f"Database '{dbname}' created successfully")
        else:
            print(f"Database '{dbname}' already exists")

    except psycopg2.Error as e:
        print(e)
        raise
    finally:
        if cur: cur.close()
        if conn: conn.close()
```

Hình 2 : Hàm creat_db()

+ Tên hàm : creat_db(db name) : nhận tham số đầu vào là db (tên database cần kiểm tra và tạo)

+ Khởi tạo các biến : conn (đối tượng kết nối đến cơ sở dữ liệu) , cur (đối tượng để thực thi lệnh truy vấn SQL)
+ Xây dựng khối lệnh try – except – finally để đảm báo đóng đết nối và xử lí
+ Kết nối với cơ sở dữ liệu : dùng hàm getopenconnection (ở trên)

  conn = getopenconnection(dbname='postgres')
+ Bật chế độ AUTOCOMMIT (cho phép lệnh CREAT DATABASE hoạt động ngoài transaction)
+ Tạo cursor để thực thi SQL : ***cur = conn.cursor()***
+ Kiểm tra xem database đã tồn tại chưa : bằng cách tìm trong hệ bảng (system catalog table) của PostgreSQL có tồn tại tên cơ sở dữ liệu là ***dbname*** không :

  cur.execute("SELECT 1 FROM pg_catalog.pg_database WHERE datname = %s", (dbname,))
+ Xử lí kết quả kiểm tra : Chưa có thì tạo mới , đã có thì in thông báo database đã tồn tại :

```
if not cur.fetchone():
    cur.execute(SQL("CREATE DATABASE {}").format(Identifier(dbname)))
    print(f"Database '{dbname}' created successfully")
else:
    print(f"Database '{dbname}' already exists")
```

+ Đóng kết nối và cursor

- **Tóm tắt luồng chạy** :
  - Kết nối vào database mặc định
  - Đặt chế độ autocommit
  - Kiểm tra database dbname có tồn tại không
  - Nếu chưa tồn tại => Tạo mới và in thông báo “created successfully” , Nếu tồn tại thì in thông báo “dbname already exists”
  - Dù thành công hay không => đóng kết nối , xử lí ngoại lệ
- **Tác dụng** :
  + Tự động tạo cơ sở dữ liệu trong PostgreSQL nếu chưa tồn tại
  + Hoạt động như một hàm tiện ích : Kiểm tra đã tồn tại chưa ; chưa tồn tại => tạo mới ; đã tồn tại => Bỏ qua , không tạo nữa

2.1.3 : def preprocess_line(line)

- **Mục đích** : Phân tách dữ liệu đầu vào bằng dấu “::”
- **Ý tưởng xây dựng**:

+ Bỏ khoảng trắng thừa ở đầu và chuối chuỗi
+ Chia dòng thành sanh sách parts , dựa vào dấu phân cách "::"

- **Chi tiết** :

```
def preprocess_line(line):
    parts = line.strip().split("::")
    return f"{parts[0]}\t{parts[1]}\t{parts[2]}\n"
```

Hình 3 : Hàm preprocess_line(line)

+ Tham số đầu vào : **line** : là một chuối String , đại diện cho một dòng dữ liệu.

  VD : 1::122::5::838985046 là chuỗi dữ liệu đại diện cho 1 đánh giá của một người dung với một bộ phim , có định dạng :

  UserID :: MoviceID::Rating::Timestamp

+ Xử lí chuỗi :
  - Bỏ khoảng trắng thừa ở đầu và cuối : ***line.strip()***
  - Chia chuối thành list các phần tử , dựa vào dấu phân cách "::" : ***line.split("::")***
  - Lưu kết quả : ***parts = line.strip().split("::")***
+ Trả về kết quả :
  - Lấy 3 phần tử đầu tiên của danh sách part : parts[0] , part[1] , part[2] tương ứng với 3 cột dữ liệu (UserID , MovieID , Rating)
  - Kí tự tap (\t) để ngăn cách các cột

- **Tóm tắt luồng chạy** :
  + Nhận đầu vào là chuối line (ví dụ : 1::122::5::838985046)
  + Xử lí sơ bộ : Bỏ khoảng trắng thừa (ví dụ : 1::122::5::838985046)
  + Tách chuỗi và lưu vào danh sách : ví dụ part = ['1' , '122' , '5' , '838985046']
  + Lấy 3 phần tử đầu tiên và trả về : ví dụ "1\t122\t5" => Là dữ liệu cần cho 3 cột tương ứng của database (UserID ; MoviceID ; Rating)

- **Tác dụng** :
  + Chuyển đổi dữ liệu bằng :: thành dòng phân tách bằng tab (\t)
  + Chuẩn hóa dữ liệu đầu vào cho tiền xử lí dữ liệu

2.1.4 : def insert_partition(args)

- **Mục đích** : Chèn dữ liệu từ bảng gốc *ratings_table_name* vào các bảng con *partition table* dựa trên khoảng rating cụ thể .
- **Ý tưởng xây dựng** :

- Bảng dữ liệu gốc sẽ chứa các bản ghi với cột userid , movied id , rating
- Bảng con có tên range_part0 , range_part1 , range_part2 tùy thuộc vào giá trị của i
- Mỗi bảng con lưu trữ dữ liệu rating trong khoảng (minRange – maxRange)
- Sử dụng lệnh SQL COPY để copy dữ liệu gốc vào bảng con

- **Chi tiết code** :

```
def insert_partition(args):
    ratings_table_name, i, delta, openconnection = args
    conn = openconnection
    cur = conn.cursor()

    minRange = i * delta
    maxRange = minRange + delta
    table_name = f"range_part{i}"

    if i == 0:
        query = SQL("""
            INSERT INTO {}
            SELECT userid, movieid, rating FROM {}
            WHERE rating >= %s AND rating <= %s
        """).format(
            Identifier(table_name),
            Identifier(ratings_table_name)
        )
    else:
        query = SQL("""
            INSERT INTO {}
            SELECT userid, movieid, rating FROM {}
            WHERE rating > %s AND rating <= %s
        """).format(
            Identifier(table_name),
            Identifier(ratings_table_name)
        )

    cur.execute(query, (minRange, maxRange))
    conn.commit()
    cur.close()
```

Hình 4 : Hàm insert_partition

- **Giải thích code** :

  + Hàm ***insert_partution(args)*** nhận đầu vào là một list gồm bảng gốc ratings_table_name, i (chỉ số partition) , delta (độ rộng khoảng partition => được tính bằng rating/ số phân mảnh) , openconnection (kết nối DB đang mở)

  + Thiết lập conn và khởi tạo các giá trị :

```
conn = openconnection
cur = conn.cursor()

minRange = i * delta
maxRange = minRange + delta
table_name = f"range_part{i}"
```

  Trong đó :

  - MinRange : điểm bắt đầu trong mỗi bảng partition
  - MaxRange : điểm kết thúc trong mỗi bảng partition
  - Table_name = range_part{i} : Tên bảng partition ứng với mỗi giả trị của I

    Ví dụ : Với N bằng 2 , ta tính được delta = 2.5, từ đó ta có 2 bảng

    Range_part0 chứa các giả trị từ [0 ; 2.5]

    Range_part1 chứa các giá trị từ (2.5 ; 5]

  + Xây dựng câu lệnh truy vấn SQL :

```
if i == 0:
    query = SQL("""
        INSERT INTO {}
        SELECT userid, movieid, rating FROM {}
        WHERE rating >= %s AND rating <= %s
    """).format(
        Identifier(table_name),
        Identifier(ratings_table_name)
    )
else:
    query = SQL("""
        INSERT INTO {}
        SELECT userid, movieid, rating FROM {}
        WHERE rating > %s AND rating <= %s
    """).format(
        Identifier(table_name),
        Identifier(ratings_table_name)
    )
```

- Với i= 0 , ta sẽ lấy rating theo [minRange ; maxRange]
- Với i> 0 , ta sẽ lấy rating theo (minRange; maxRange]
- Tạo bảng partition tương ứng với i

+ Thực thi câu lệnh SQL , commit và đóng con trỏ

```
cur.execute(query, (minRange, maxRange))
conn.commit()
cur.close()
```

- **Tóm tắt luồng chạy** :
  - Khởi tạo hàm và nhận các tham số đầu vào
  - Mở kết nối và tạo con trỏ
  - Tính toán khoảng rating của partition i tương ứng với 2 giá trị minRange và maxRange
  - Tạo câu lệnh SQL với 2 trường hợp i= 0 và i != 0
  - Thực thi câu lệnh SQL
  - Commit và đóng con trỏ

- **Ý nghĩa và tác dụng**
  - Phân dữ liệu trong bảng gốc (ratings_table_name) thành các bảng nhỏ hơn (partition) dựa trên các giá trị bảng rating
  - Mỗi partition chứa một khoảng rating nhất định, không bị trùng lặp
  - Là điều kiện tiên quyết để thực hiện hàm ***rangepartition***

2.1.5 : def count_partitions(prefix,openconncection)

- **Mục đích** : Đếm số lượng bảng (tables) trong cơ sở dữ liệu có tên bắt đầu bằng chuỗi tiền tố prefix nào đó
- **Ý tưởng xây dựng** :
  - Sử dụng prefix để xác định nhóm bảng
  - Sử dụng câu lệnh SQL để đếm số bảng
  - Trả về kết quả
- **Chi tiết code** :

```
def count_partitions(prefix, openconnection):
        """
        Count number of tables starting with the given prefix.
        """
        cur = openconnection.cursor()
        cur.execute(
        SQL("SELECT COUNT(*) FROM pg_stat_user_tables WHERE relname LIKE {}").format(
            Literal(prefix + '%')
        )
    )
        count = cur.fetchone()[0]
        cur.close()
        return count
```

Hình 5 : Hàm count_partitions()

- **Giải thích**
  - Khai báo hàm : nhận đầu vào là prefix là chuỗi tiền tố cần đếm ; openconnection : đối tượng kết nối đến PostgreSQL
  - Tạo một cursor để kết nối đến SQL
  - Truy vấn SQL : đếm số bảng có tên giống như prefix

```
    cur.execute(
    SQL("SELECT COUNT(*) FROM pg_stat_user_tables WHERE relname LIKE {}").format(
        Literal(prefix + '%')
    )
```

  - Trả kết quả truy vấn về count
  - Đóng kết nối
- **Luồng thực hiện** :

- Khởi tạo hàm và nhận tham số đầu vào
- Tạo cursor
- Gửi truy vấn đếm bảng
- Lấy kết quả
- Đóng cursor
- Trả kết quả

- **Ý nghĩa và tác dụng** :
  - Tối ưu hóa hiệu suất truy vấn
  - Dễ dàng quản lí và xử lý dữ liệu lớn theo từng phần nhỏ

2.1.6 : def get_rr_index(index)

- **Mục đích** : Đọc một chỉ số số nguyên (index) từ một file (tên là RR_INDEX_FILE) - nếu file không tồn tại hoặc đọc lỗi thì trả về 0.
- **Chi tiết code** :

```
RR_INDEX_FILE = 'rr_index.txt'

def get_rr_index():
    try:
        with open(RR_INDEX_FILE, 'r') as f:
            return int(f.read())
    except:
        return 0
```

Hình 6 : Hàm get_rr_index()

- **Ý nghĩa** :
  Giữ trạng thái của quá trình phân vùng Round-Robin qua nhiều lần thực thi. Điều này giúp hệ thống:
  - Ghi dữ liệu đúng luân phiên qua các bảng con
  - Không bị lặp lại hoặc mất dữ liệu giữa các lần chạy chương trình
  - Dễ mở rộng và khôi phục nếu bị gián đoạn

2.1.7 : def get save_rr_index(index)

- **Mục đích** : Hàm save_rr_index(index) dùng để ghi (lưu) chỉ số index hiện tại vào file RR_INDEX_FILE
- **Chi tiết code** :

```python
def save_rr_index(index):
    with open(RR_INDEX_FILE, 'w') as f:
        f.write(str(index))
```

Hình 7 : Hàm save_rr_index()

- **Ý nghĩa** : Hàm save_rr_index(index) giúp duy trì thứ tự phân vùng liên tục trong hệ thống Round-Robin, đảm bảo rằng:
  - ➢ Không có phân vùng nào bị bỏ sót hoặc ghi lặp
  - ➢ Quá trình ghi dữ liệu được phân phối đều và tuần tự
  - ➢ Hệ thống có thể dừng và tiếp tục mà không mất trạng thái

2.2 : Các hàm chính

2.2.1 : Hàm loadratings()

- **Mục đích** : Nhập dữ liệu ratings từ file vào cơ sở dữ liệu (PostgreSQL)
- **Ý tưởng xây dựng** :
  - + Vấn đề thực tế :
    - o File ratings có thể rất lớn (vài triệu dòng)
    - o Sử dụng INSERT từng dòng sẽ chậm và có thể làm cho database quá tải
  - ⇨ **Giải Pháp :**
    - o Dùng lện COPY : Lệnh copy của PostgreSQL để load dữ liệu bulk trực tiếp từ file
    - o Chia dữ liệu thành các bathch nhỏ (150.000 dòng) để tránh việc giữ toàn bộ file lớn trong RAM mà vẫn tận dụng được sức mạnh ghi bulk
    - o Dùng StringIO để tạm lưu dữ liệu chuẩn hóa trong RAM, thay vì phải ghi ra file tạm => Tiết kiệm I/O đĩa
  - + Lí do tạo bảng nếu chưa có : giúp dễ dàng tái sử dụng hàm ; không cần lo về bảng chưa tồn tại khi load lần đầu
  - + Xử lí an toàn :
    - o Nếu có lỗi => rollback tránh ghi dữ liệu sai
    - o Cuối cùng là đóng cursor để tránh rò rỉ dữ liệu
- **Chi tiết** :

```python
def loadratings(ratings_table_name, ratings_file_path, open_connection):
    conn = open_connection
    cur = None
    try:
        cur = conn.cursor()
        cur.execute(SQL("""
            CREATE TABLE IF NOT EXISTS {} (
                userid INTEGER,
                movieid INTEGER,
                rating FLOAT
            );
        """).format(Identifier(ratings_table_name)))
        conn.commit()

        batch_size = 150_000
        buffer = StringIO()

        with open(ratings_file_path, 'r') as file:
            for i, line in enumerate(file, 1):
                buffer.write(preprocess_line(line))

                if i % batch_size == 0:
                    buffer.seek(0)
                    copy_sql = SQL("""
                        COPY {} (userid, movieid, rating)
                        FROM STDIN WITH (FORMAT TEXT, DELIMITER E'\t')
                    """).format(Identifier(ratings_table_name))
                    cur.copy_expert(copy_sql, buffer)

                    buffer.seek(0)
                    buffer.truncate()
```

```
        # Insert phần còn lại chưa đến batch
        if buffer.tell():
            buffer.seek(0)
            copy_sql = SQL("""
                COPY {} (userid, movieid, rating)
                FROM STDIN WITH (FORMAT TEXT, DELIMITER E'\t')
            """).format(Identifier(ratings_table_name))
            cur.copy_expert(copy_sql, buffer)

    conn.commit()

except (psycopg2.Error, IOError, Exception) as e:
    print("Error:", e)
    if conn:
        conn.rollback()
    raise

finally:
    if cur:
        cur.close()
```

Hình 8 : Hàm load_rating()

- **Giải thích code** :
  - + Đối số : ratings_table_name : Tên bảng trong database để lưu ratings , ratings_file_path : đường dẫn tới file rating ; open_connection : kết nối psycopg2 được mở sẵn
  - + Khởi tạo kết nối và con trỏ : ***conn = open_connection ; cur = None***
  - + Bắt đầu khối try – except – finally
  - + Tạo bảng nếu chưa tồn tại (bảng được tạo gồm 3 cột userid, movieid, rating) và xác nhận commit tạo bảng :

```
cur = conn.cursor()
cur.execute(SQL("""
    CREATE TABLE IF NOT EXISTS {} (
        userid INTEGER,
        movieid INTEGER,
        rating FLOAT
    );
""").format(Identifier(ratings_table_name)))
conn.commit()
```

+ Chuẩn bị batch và buffer : ***batch_size = 150_000 :*** Mỗi lần đẩy 150000 dòng
+ Đọc dữ liệu từ file và chuyển dữ liệu theo hàm preprocess_line (mục 2.1.3)

```
with open(ratings_file_path, 'r') as file:
    for i, line in enumerate(file, 1):
        buffer.write(preprocess_line(line))
```

+ Khi đủ 1 batch là 150000 dòng thì tiến hành đẩy dữ liệu trong batch vào database. Sử dụng lệnh SQL COPY để chèn dữ liệu vào các cột (userid , movieid, rating)

```
if i % batch_size == 0:
    buffer.seek(0)
    copy_sql = SQL("""
        COPY {} (userid, movieid, rating)
        FROM STDIN WITH (FORMAT TEXT, DELIMITER E'\t')
    """).format(Identifier(ratings_table_name))
    cur.copy_expert(copy_sql, buffer)

    buffer.seek(0)
    buffer.truncate()
```

+ Insert phần còn lại chưa đến batch . Kiểm tra xem dữ liệu còn không ***buffer.tell() != 0*** , tức là còn dữ liệu chưa copy vào DB => Tiếp tục sử dụng truy vấn SQL và copy giống phần trên.

```
        # Insert phần còn lại chưa đến batch
        if buffer.tell():
            buffer.seek(0)
            copy_sql = SQL("""
                COPY {} (userid, movieid, rating)
                FROM STDIN WITH (FORMAT TEXT, DELIMITER E'\t')
            """).format(Identifier(ratings_table_name))
            cur.copy_expert(copy_sql, buffer)
```

+ Khối except và finally : Để xử lí ngoại lệ và đóng kết nối

```
except (psycopg2.Error, IOError, Exception) as e:
    print("Error:", e)
    if conn:
        conn.rollback()
    raise

finally:
    if cur:
        cur.close()
```

- **Tóm tắt luồng chạy** :
  - Kết nối và tạo bảng (nếu chưa có)
  - Đọc file theo dòng
  - Theo dõi số dòng đọc được : Sau mỗi batch_size = 150.000 dòng thì thực hiện lệnh COPY
  - Chèn nốt phần còn lại nếu có
  - Xử lí lỗi và đóng cursor sau khi hoàn tất
- **Ý nghĩa và tác dụng** :
  - Tạo bảng ratings nếu chưa có
  - Nạp dữ liệu từ file chứa các dòng userid, movieid , rating
  - Ghi vào database theo từng đợt , tránh quá tải bộ nhớ
  - Xử lí an toàn và nhanh chóng cả với file rất lớn

2.2.2 : Hàm rangepartition()

- **Mục đích** : Chia dữ liệu trong bảng ratingstable thành nhiều phân vùng theo giá trị rating và lưu chúng vào các bảng riêng biệt range_part0 , range_part1 … nhằm tối ưu việc truy vấn hoặc xử lí dữ liệu sau này
- **Ý tưởng xây dựng** :
  - Tính bước phân chia step : nếu có n phân vùng thì môi phân vùng tương ứng với khoảng rộng step = 5.0/n
  - Tạo các bảng phân vùng : vd range_part0 , range_part1

- Chèn dữ liệu vào từng phân vùng : gọi hàm insert_partition (ở mục 2.1.4)
- Xử lí lỗi và đóng kết nối

- **Chi tiết code** :

```
def rangepartition(ratingstablename, numberofpartitions, openconnection):
    con = openconnection
    cur = con.cursor()

    try:
        step = 5.0 / numberofpartitions

        # Tạo bảng range_part{i}
        for i in range(numberofpartitions):
            table_name = f"range_part{i}"
            cur.execute(SQL("DROP TABLE IF EXISTS {}").format(Identifier(table_name)))
            cur.execute(SQL("CREATE TABLE {} (userid INT, movieid INT, rating FLOAT)").format(Identifier(table_name)))

        con.commit()

        # Thực hiện insert tuần tự
        for i in range(numberofpartitions):
            insert_partition((ratingstablename, i, step, con))

    except Exception as e:
        con.rollback()
        print("rangepartition failed:", e)
        raise
    finally:
        cur.close()
```

Hình 9 : Hàm rangerpartition()

- **Giải thích** :
  - Tên hàm def rangepartition nhận tham số đầu vào là : ratingstablename (tên bảng chứa dữ liệu gốc ratings) , numberofpartitions (số lượng phân vùng cần tạo) , openconnection (kết nối tới cơ sở dữ liệu)
  - Gán kết nối conn và tạo một cursor để thực hiện các lênh truy vấn SQL
  - Khôi lệnh try – except – finally
    - Step = 5.0 / numberofpartitons (tương ứng với delta trong hàm insert_partition) : khoảng rating cho mối phân vùng
    - Thực hiện tạo bảng range_part{i}

```
# Tạo bảng range_part{i}
for i in range(numberofpartitions):
    table_name = f"range_part{i}"
    cur.execute(SQL("DROP TABLE IF EXISTS {}").format(Identifier(table_name)))
    cur.execute(SQL("CREATE TABLE {} (userid INT, movieid INT, rating FLOAT)").format(Identifier(ta

con.commit()
```

    - Xóa bảng nếu đã tồn tại
    - Tạo mới bảng với 3 cột userid , movieid , rating . Bảng rỗng sẽ được chèn dữ liệu
    - Chèn dữ liệu vào bảng : gọi hàm insert_partition (2.1.4)

```
# Thực hiện insert tuần tự
for i in range(numberofpartitions):
    insert_partition((ratingstablename, i, step, con))
```

- Xử lí ngoại lệ và đóng con trỏ .

- **Luồng thực hiện** :
  - Mở kết nối và tạo con trỏ
  - Tính khoảng chia theo rating
  - Xóa và tạo mới các bảng partition
  - Lưu lại các thay đổi tạo bảng
  - Gọi hàm insert_partition dữ liệu vào từng bảng theo khoảng rating
  - Nếu lỗi xảy ra , rollback và báo lỗi
  - Đóng con trỏ cursor
- **Ý nghĩa và tác dụng :**
  - Giúp chia bảng dữ liệu rating thành nhiều bảng nhỏ dựa trên khoảng giá trị rating, làm cho việc truy xuất hoặc xử lý dữ liệu theo rating được nhanh hơn, có tổ chức hơn
  - Đây là ký thuật range partitioning rất phổ biến trong quản lý cơ sở dữ liệu lớn

2.2.3 : Hàm roundrobinpartition()

- **Mục đích** :
  - Tự động phân chia dữ liệu từ bảng ratingstablename thành nhiều bảng nhỏ (partition tables) theo thuật toán Round-Robin.
  - Đây là cách phân vùng dữ liệu đồng đều, dùng để:
  + Tránh quá tải một phân vùng
  + Tăng tốc độ truy vấn và xử lý song song
  + Đảm bảo các bảng con nhận dữ liệu luân phiên
- **Ý tưởng xây dựng** :
  - Khởi tạo chỉ số vòng lặp round robin
  - Tạo bảng tạm : Sử dụng câu lênh SQL tạo bảng tạm và đánh số theo thứ tự userid . Việc này giúp xác định vị trí của mỗi dòng trong tập dữ liệu để phân chia round robin
  - Tạo các bảng phân vùng con với các tên lần lượt là rrobin_part0 , rrobin_part1 … có cấu trúc giống bảng gốc gồm các cột (userid , movieid , rating)
  - Phân chia dữ liệu vào các bảng con theo RoundRobin
- **Chi tiết code** :

```python
def roundrobinpartition(ratingstablename, numberofpartitions, openconnection):
    """
    Function to create partitions of main table using round robin approach.
    """
    save_rr_index(0)
    RROBIN_TABLE_PREFIX = 'rrobin_part'
    cur = openconnection.cursor()

    # Create temporary table with row numbers
    temp_tb = SQL("""
        CREATE TEMPORARY TABLE temp AS
        SELECT userid, movieid, rating, ROW_NUMBER() OVER (ORDER BY userid) AS rnum
        FROM {};
    """).format(Identifier(ratingstablename))
    cur.execute(temp_tb)

    for i in range(numberofpartitions):
        table_name = f"{RROBIN_TABLE_PREFIX}{i}"
        # Create partition table
        cur.execute(SQL("""
            CREATE TABLE {} (
                userid INTEGER,
                movieid INTEGER,
                rating FLOAT
            );
        """).format(Identifier(table_name)))

        # Insert into partition table using mod
        query = SQL("""
            INSERT INTO {} (userid, movieid, rating)
            SELECT userid, movieid, rating FROM temp
            WHERE MOD(temp.rnum - 1, %s) = %s;
        """).format(Identifier(table_name))
        cur.execute(query, (numberofpartitions, i))

    openconnection.commit()
    cur.close()
```

Hình 10 : Hàm roundrobinpartition()

- **Giải thích code** :
  - Khởi tạo hàm nhận tham số đầu vào là : ratingstablename (bảng chính chứa dữ liệu gốc) , numberofpartitions (số bảng con muốn tạo ra để phân vùng) , openconnection (kết nối đến cơ sở dữ liệu)
  - Thiết lập chỉ số round robin về 0 , đặt tên cho tiền tố cho các bảng con sẽ tạo

```python
save_rr_index(0)
RROBIN_TABLE_PREFIX = 'rrobin_part'
cur = openconnection.cursor()
```

- Khởi tạo bảng tạm temp chứa toàn bộ dữ liệu gốc , thêm một cột rnum để đánh số thứ tự dòng :

```
temp_tb = SQL("""
    CREATE TEMPORARY TABLE temp AS
    SELECT userid, movieid, rating, ROW_NUMBER() OVER (ORDER BY userid) AS rnum
    FROM {};
""").format(Identifier(ratingstablename))
cur.execute(temp_tb)
```

- Tạo các bảng con tương ứng và có đầy đủ thuộc tính giống bảng gốc : vd : rrobin_part0 , rrobin_part1

```
for i in range(numberofpartitions):
    table_name = f"{RROBIN_TABLE_PREFIX}{i}"
    # Create partition table
    cur.execute(SQL("""
        CREATE TABLE {} (
            userid INTEGER,
            movieid INTEGER,
            rating FLOAT
        );
    """).format(Identifier(table_name)))
```

- Chèn dữ liệu vào bảng con :
  - + Sử dụng câu lệnh truy vấn SQL , thêm các thuộc tính userid , movieid , rating được lấy dữ liệu từ các cột tương ứng của bảng **temp**
  - + số dư = (lấy cột giá trị cột rnum – 1) chia cho (số bảng con muốn tạo ra để phân vùng)
  - + Bảng được thêm dữ liệu vào : rrobin_part{số dư }

```
    query = SQL("""
        INSERT INTO {} (userid, movieid, rating)
        SELECT userid, movieid, rating FROM temp
        WHERE MOD(temp.rnum - 1, %s) = %s;
    """).format(Identifier(table_name))
    cur.execute(query, (numberofpartitions, i))

openconnection.commit()
cur.close()
```

- Commit và đóng con trỏ .

- **Luồng chạy thực hiện chương trình** :
  - Khởi tạo hàm roundrobinpartition với các tham số đầu vào
  - Reset vị trí chỉ số round robin (gọi hàm save_rr_index(0))
  - Tạo con trỏ và bảng tạm
  - Tạo các bảng phân vùng
  - Chèn dữ liệu vào bảng
  - Hoàn thành và đóng con trỏ
- **Ý nghĩa và tác dụng** :

  Hàm roundrobinpartition() là một hàm dùng để phân chia dữ liệu từ một bảng lớn (bảng đánh giá phim) thành n bảng nhỏ hơn (phân vùng) theo phương pháp Round Robin (luân phiên).
  - Dữ liệu từ bảng gốc được chia đều đặn, theo thứ tự dòng (theo userid) vào các bảng phân vùng rrobin_part0, rrobin_part1, ..., rrobin_partN
  - Phân vùng này không dựa trên giá trị cụ thể (như rating), mà chỉ dựa trên thứ tự của dòng

2.2.4 : Hàm rangeinsert()

- **Mục đích** : Chèn một bản ghi (dòng dữ liệu gồm userid, itemid, rating) vào:
  + Bảng gốc (main table) - để lưu toàn bộ lịch sử đánh giá.
  + Bảng phân vùng theo khoảng (range partition table) - để phục vụ truy vấn hiệu quả.
- **Ý tưởng xây dựng** :
  - Đối với bảng ratings gốc : Ta thực hiện truy vấn SQL để thêm vào các cột tương ứng các giá trị UserID , ItemID , Rating
  - Đối với bảng Phân mảnh ta làm theo các bước :
    + Làm tương tự với hàm insert_partition , ta xác định được khoảng phân vùng trong mỗi phân mảnh :

      delta = MAX_RATING_SCALE / numberofparrtitions

      (hay 5/số phân mảnh)
    + Xác định bảng con phù hợp : ***index = int (rating / delta)***

+ Trường hợp ngoại lệ : Nếu rating nằm ngay ranh giới như 1.0 , 2.0 ... và index > 0 thì index trừ đi 1 sẽ ra bảng con được chọn
+ Thực hiện chèn : Dùng lệnh truy vấn SQL

- **Chi tiết code** :

```
def rangeinsert(ratingstablename, userid, itemid, rating, openconnection):
    RANGE_TABLE_PREFIX = 'range_part'
    MAX_RATING_SCALE = 5.0

    cur = openconnection.cursor()
    numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)

    if numberofpartitions <= 0:
        raise ValueError("No range partitions found.")

    delta = MAX_RATING_SCALE / numberofpartitions
    index = min(numberofpartitions - 1, max(0, int(rating / delta)))

    if rating % delta == 0 and index > 0:
        index -= 1
    partition_table = f"{RANGE_TABLE_PREFIX}{index}"

    insert_sql = SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);")
    cur.execute(insert_sql.format(Identifier(ratingstablename)), (userid, itemid, rating))
    cur.execute(insert_sql.format(Identifier(partition_table)), (userid, itemid, rating))

    openconnection.commit()
    cur.close()
```

Hình 11 : Hàm rangeinsert()

- **Giải thích code** :

+ Khởi tạo hàm rangeinsert nhận tham số đầu vào là bảng gốc ratingstablename , userid , itemid, rating (các giá trị cần chèn) , openconnection (kết nối cơ sở dữ liệu) .
+ Định nghĩa các hằng số : RANGE_TABLE_PREFIX (tên bảng phân vùng) ; MAX_RATING_SCALE : Đánh giá tối đa của rating
+ Mở cursor và đếm số phân vùng : gọi hàm count_partitions (2.1.5) và kiểm tra trường hợp ngoại lệ

```
cur = openconnection.cursor()
numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)

if numberofpartitions <= 0:
    raise ValueError("No range partitions found.")
```

+ Tính khoảng giá trị trong mỗi phân vùng và bảng con tương ứng (đối với trường hợp chèn vào bảng phân mảnh) , sau đó xác định bảng con phù hợp với giá trị rating cần chèn

```
delta = MAX_RATING_SCALE / numberofpartitions
index = min(numberofpartitions - 1, max(0, int(rating / delta)

if rating % delta == 0 and index > 0:
    index -= 1
partition_table = f"{RANGE_TABLE_PREFIX}{index}"
```

+ Thực hiện câu lệnh SQL để chèn giá trị vào bảng ratings gốc và ratings phân mảnh . Sau đó commit và đóng con trỏ .

```
insert_sql = SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);")
cur.execute(insert_sql.format(Identifier(ratingstablename)), (userid, itemid, rating)
cur.execute(insert_sql.format(Identifier(partition_table)), (userid, itemid, rating))

openconnection.commit()
cur.close()
```

- **Luồng thực hiện** :
  + Khởi tạo hàm và các tham số đầu vào
  + Định nghĩa các hằng số mặc định
  + Đếm số phân vùng và kiểm tra trường hợp ngoại lệ
  + Tính khoảng rating mỗi phân vùng
  + Điều chỉnh rating nằm đúng biên
  + Xác định bảng phù hợp
  + Tạo câu lện SQL INSERT và thực hiện chèn vào bảng chính và bảng phân mảnh
  + Commit và đóng cursor
- **Ý nghĩa và tác dụng** :
  + Chèn được dữ liệu vào bảng chính và vào đúng bảng rating phân mảnh
  + Hỗ trợ phân mảnh dữ liệu để tăng hiệu năng
  + Giúp truy vấn nhanh hơn nếu chỉ tìm theo khoảng rating

2.2.5 : Hàm roundrobininsert()

- **Mục đích** :
  - ➢ Chèn một bản ghi rating (gồm userid, itemid, rating) vào:
    + Bảng chính ratingstablename

+ Một trong các bảng phân vùng rrobin_partN, theo thứ tự vòng tròn (round robin).

➢ Đảm bảo dữ liệu được phân phối đều giữa các phân vùng để tăng hiệu năng truy vấn và quản lý dữ liệu

- **Ý tưởng xây dựng** :
  - ➢ Đối với bảng ratings gốc : Ta thực hiện truy vấn SQL để thêm vào các cột tương ứng các giá trị UserID , ItemID , Rating
  - ➢ Đối với bảng ratings phân mảnh : ta làm theo các bước :
    - + Xác định số phân vùng hiện tại : dùng SELECT COUNT(*)
    - + Xác định phân vùng cần ghi dữ liệu : tính số dư của phép chia (current_index) % numberofparrtitions
    - + Thực hiện lệnh truy vấn SQL
- **Chi tiết code** :

```
def roundrobininsert(ratingstablename, userid, itemid, rating, openconnection):
    con = openconnection
    cur = con.cursor()
    try:
        cur.execute("SELECT COUNT(*) FROM information_schema.tables WHERE table_name LIKE 'rrobin_part%';")
        numberofpartitions = cur.fetchone()[0]

        current_index = get_rr_index()
        target_partition = current_index % numberofpartitions

        insert_sql = SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);")
        cur.execute(insert_sql.format(Identifier(ratingstablename)), (userid, itemid, rating))

        cur.execute(SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s)")
                    .format(Identifier("rrobin_part" + str(target_partition))),
                    (userid, itemid, rating))

        save_rr_index(current_index + 1)

        con.commit()
    except Exception as e:
        con.rollback()
        print("roundrobininsert failed:", e)
        raise
    finally:
        cur.close()
```

Hình 12 : Hàm roundrobininsert()

+ **Giải thích** :

  + Khởi tạo hàm nhận tham số đầu vào là : ratingstablename (bảng rating gốc) ; userid, itemid, rating (dữ liệu cần chèn) , openconnection (mở kết nối cơ sở dữ liệu)

  + Đếm số phân vùng hiện tại và xác định vụ trí phân vùng thỏa mãn để chèn

```
cur.execute("SELECT COUNT(*) FROM information_schema.tables WHERE table_name LIKE 'rrobin_part
numberofpartitions = cur.fetchone()[0]

current_index = get_rr_index()
target_partition = current_index % numberofpartitions
```

+ Thực hiện chèn dữ liệu vào bảng ratings gốc và bảng ratings phân mảnh bằng câu lệnh INSERT INTO …

```
insert_sql = SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);")
cur.execute(insert_sql.format(Identifier(ratingstablename)), (userid, itemid, rati

cur.execute(SQL("INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s)")
            .format(Identifier("rrobin_part" + str(target_partition))),
            (userid, itemid, rating))
```

+ Gọi hàm save_rr_index để lưu lại file
+ Xử lí ngoại lệ và đóng con trỏ

- **Luồng sự kiện** :
  - Khởi tạo hàm và nhận các đối số đầu vào
  - Xác định các phân vùng
  - Lấy lượt chèn và xác định vùng chèn hợp lí
  - Tính toán phân vùng cần ghi dữ liệu
  - Chèn dữ liệu vào bảng chính và chèn dữ liệu vào phân vùng tương ứng
  - Cập nhật chỉ số round robin
  - Xứ lí lỗi và đóng con trỏ
- **Ý nghĩa là tác dụng**
  - Chèn dữ liệu vào nhiều bảng phân vùng theo thứ tự "xoay vòng" (round-robin)
  - Duy trì dữ liệu đồng thời ở bảng chính và bảng phân vùng
  - Tự động phân phối dữ liệu đều giữa các bảng, không cần người dùng quyết định mỗi lần chèn vào đâu.

CHƯƠNG 3. KẾT QUẢ VÀ ĐÁNH GIÁ

3.1 : Hướng dẫn cài đặt

3.1.1 : Tạo môi trường ảo :

```
python3 -m venv .venv
source ./.venv/bin/activate
```

3.1.2 : Cài đặt thư viện

```
pip3 install -r requirements.txt
```

3.1.3 : Tải file ratings.data

+ Tải tệp rating.dat từ trang (http://movielens.org).
+ Nếu gặp gặp lỗi PostgreSQL : https://stackoverflow.com/questions/18664074/getting-error-peer-authentication-failed-for-user-postgres-when-trying-to-ge

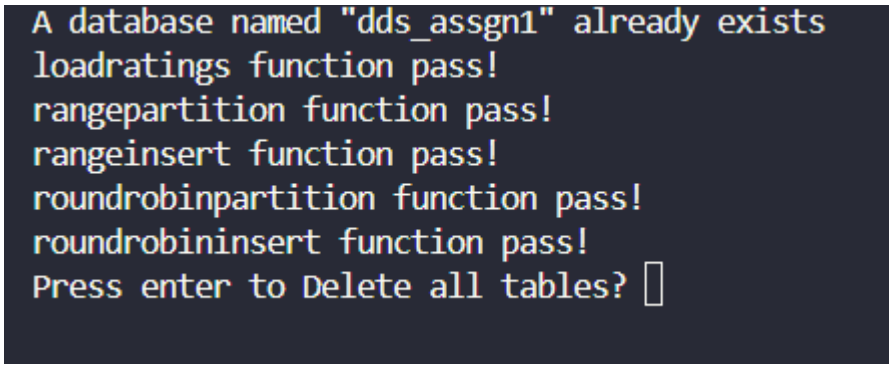
3.1.4 : Cấu hình .env

```
HOST=localhost
PORT=5432
DATABASE_NAME=your_database_name
USER=your_username
PASSWORD=your_password
```

DATABASE_NAME: là tên của database lưu trữ bảng ratings và các phân mảnh của nó. Ví dụ: dds_assgn1

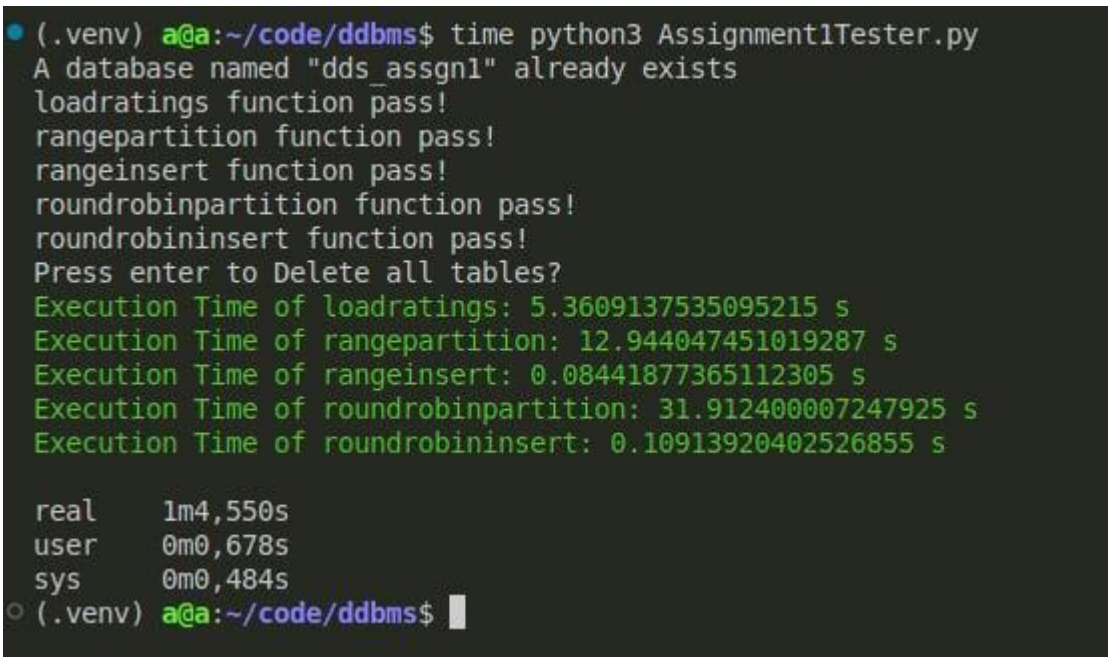
3.2 : Kết quả tổng quan

- Kết quả Test :



Hình 13 : Kết quả tổng quan chạy chương trình

- Thời gian Test :



Hình 14 : Kết quả thời gian test chương trình

3.3 : Kết quả chi tiết

3.3.1 :Kết quả hàm loadratings

- **Thời gian chạy tốt nhất : 5.36s**
- Bảng ratings

```
dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
```

3.3.2 : Kết quả hàm ragepartition

- **Thời gian chạy tốt nhất : 12.94s**
- Các bảng tạo ra :

```
public | range_part0 | table | postgres
public | range_part1 | table | postgres
public | range_part2 | table | postgres
public | range_part3 | table | postgres
public | range_part4 | table | postgres
```

- Bảng range_part0

```
dds_assgn1=# select * from range_part0;
 userid | movieid | rating
--------+---------+--------
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
    100 |       2 |      0
(4 rows)
```

- Bảng range_part1

```
dds_assgn1=# select * from range_part1;
 userid | movieid | rating
--------+---------+--------
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     589 |    1.5
(3 rows)
```

- Bảng range_part2

```
dds_assgn1=# select * from range_part2;
 userid | movieid | rating
--------+---------+--------
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     520 |    2.5
(3 rows)
```

- Bảng range_part3

```
dds_assgn1=# select * from range_part3;
 userid | movieid | rating
--------+---------+--------
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     377 |    3.5
      1 |     466 |      4
      1 |     586 |    3.5
(5 rows)
```

- Bảng range_part4

```
dds_assgn1=# select * from range_part4;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     420 |      5
      1 |     480 |      5
      1 |     539 |      5
      1 |     588 |      5
(6 rows)
```

3.3.3 : Kết quả hàm rangeinsert

- Thời gian chạy tốt nhất : 0.084s
- Bảng ratings (bảng gốc) khi chèn thêm dữ liệu (100;2;0)

```
dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
    100 |       2 |      0
```

- Bảng ratings phân mảnh khi chèn thêm dữ liệu (100;2;0)

```
dds_assgn1=# select * from range_part0;
 userid | movieid | rating
--------+---------+--------
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
    100 |       2 |      0
(4 rows)
```

3.3.4 : Kết quả hàm roundrobinpartition

- **Thời gian chạy tốt nhất : 31.94s**
- Các Bảng tạo ra :

```
 public | rrobin_part0 | table | postgres
 public | rrobin_part1 | table | postgres
 public | rrobin_part2 | table | postgres
 public | rrobin_part3 | table | postgres
 public | rrobin_part4 | table | postgres
```

- Rrobin_part0

```
dds_assgn1=# select * from rrobin_part0;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     329 |    2.5
      1 |     370 |      0
      1 |     520 |    2.5
    100 |       1 |      3
(5 rows)
```

- Rrobin_part1

```
dds_assgn1=# select * from rrobin_part1;
 userid | movieid | rating
--------+---------+--------
      1 |     185 |    4.5
      1 |     355 |      2
      1 |     377 |    3.5
      1 |     539 |      5
(4 rows)
```

- Rrobin_part2

```
dds_assgn1=# select * from rrobin_part2;
 userid | movieid | rating
--------+---------+--------
      1 |     231 |      4
      1 |     356 |    1.5
      1 |     420 |      5
      1 |     586 |    3.5
(4 rows)
```

- Rrobin_part3

```
dds_assgn1=# select * from rrobin_part3;
 userid | movieid | rating
--------+---------+--------
      1 |     292 |    3.5
      1 |     362 |      1
      1 |     466 |      4
      1 |     588 |      5
(4 rows)
```

- Rrobin_part4

```
dds_assgn1=# select * from rrobin_part4;
 userid | movieid | rating
--------+---------+--------
      1 |     316 |      3
      1 |     364 |    0.5
      1 |     480 |      5
      1 |     589 |    1.5
(4 rows)
```

3.3.5 : Kết quả hàm roundrobininsert

- **Thời gian chạy tốt nhất : 0.11s**
- Bảng ratings khi thêm dữ liệu (100 ;1;3)

```
dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
    100 |       1 |      3
```

- Bảng phân mảnh khi thêm dữ liệu (100;1;3)

```
dds_assgn1=# select * from rrobin_part0;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     329 |    2.5
      1 |     370 |      0
      1 |     520 |    2.5
    100 |       1 |      3
(5 rows)
```

3.4 : Đánh giá kết quả

- Các hàm Loadratings , rangepartition , rangeinsert , roundrobinpartition, roundrobininssert đều pass qua Test
- Các hàm trên được Test trên 3 máy , gồm hai hệ điều hành Ubuntu và Window đều cho kết quả pass như nhau
- Quá trình Test không gặp trục trặc về lỗi kĩ thuật hay lập trình gì
- Kết quả đã đáp ứng được yêu cầu nhiệm vụ đề ra

TÀI LIỆU THAM KHẢO

[1] Giáo trình cơ sở dữ liệu phân tán – Học viện công nghệ Bưu Chính Viễn Thông