HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN 1

BÀI CÁO KẾT THÚC HỌC PHẦN
MÔN CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên hướng dẫn: TS. KIM NGỌC BÁCH

Nhóm học phần: 09

Nhóm bài tập lớn: 05

Sinh viên: NGUYỄN QUANG DŨNG B22DCCN133

NGUYỄN TUẤN NAM B22DCCN561

LÊ ĐĂNG NINH B22DCCN572

Hà Nội, 2025

Mục lục

Danh mục hình ảnh

Danh mục bảng biểu

1 Phân tích yêu cầu đề bài

1.1 Nhiệm vụ yêu cầu

Mô phỏng các phương pháp phân mảnh dữ liệu trên một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở (ví dụ: PostgreSQL hoặc MySQL).
Mỗi nhóm sinh viên phải tạo một tập các hàm Python để tải dữ liệu đầu vào 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 Mục tiêu bài tập

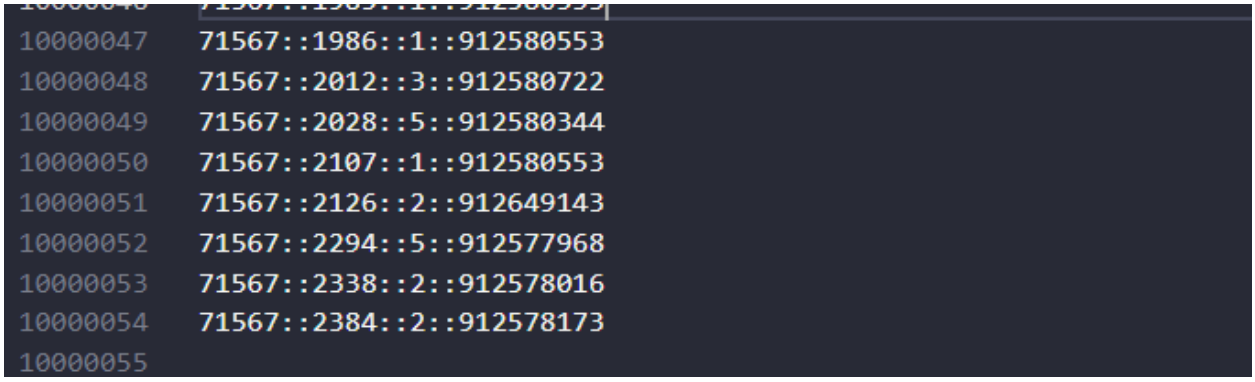
Mô phỏng quá trình phân mảnh ngang dữ liệu trên một bảng quan hệ sử dụng dữ liệu thực tế (Tập tin ratings.dat của MovieLens 10M), sử dụng cơ sở dữ liệu là PostgreSQL hoặc MySQL và ngôn ngữ lập trình là python. Áp dụng hai phương thức phân mảnh là Range Partitioning và Round-Robin Partitioning. Bao gồm các thao tác yêu cầu đó là Load dữ liệu vào một bảng gốc tên ratings, phân mảnh bảng này theo Range và Round-Robin và chèn dữ liệu mới và tự động đưa vào đúng phân mảnh tương ứng.

1.3 Công nghệ sử dụng

- Hệ điều hành: Ubuntu 22.04 / Windows 10
- Ngôn ngữ lập trình: Python 3.12.x
- Hệ quản trị SQL: PostgreSQL
- Thư viện Python: psycopg2
- IDE: VSCode / PyCharm

1.4 Dữ liệu đầu vào

dữ liệu đầu vào là 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



Hình 1: Dữ liệu đầu vào

2 Phân tích và thiết kế giải pháp

Định dạng mỗi dòng: UserID::MovieID::Rating::Timestamp Dữ liệu cần sử dụng: UserID, MovieID, Rating

2.1 Phân tích và thiết kế các hàm chính

2.1.1 Hàm loadrating

Phân tích: Hàm này chịu trách nhiệm đọc dữ liệu từ file (dạng ratings.dat) và nạp vào bảng gốc `ratings` trong cơ sở dữ liệu. Dữ liệu đầu vào thường có định dạng: `UserID:MovieID:Rating:Timestamp`.

Đầu vào:

- `ratingstablename`: Tên bảng sẽ lưu dữ liệu (thường là `ratings`).
- `ratingsfilepath`: Đường dẫn tới file dữ liệu.
- `openconnection`: Kết nối tới cơ sở dữ liệu.

Đầu ra: Không trả về giá trị, nhưng tạo bảng và nạp dữ liệu thành công vào CSDL.

Thiết kế:

- Nếu bảng đã tồn tại thì xóa đi để tránh trùng lặp.
- Tạo bảng mới với các trường: `userid`, `movieid`, `rating`.
- Đọc file, xử lý dữ liệu và nạp vào bảng bằng lệnh `COPY` hoặc `INSERT`.
- Đảm bảo dữ liệu đúng định dạng, loại bỏ các trường không cần thiết.

2.1.2 Hàm rangepartition

Phân tích: Chia bảng `ratings` thành nhiều bảng con theo từng khoảng giá trị của trường `rating` (ví dụ: 0-1, 1-2,...).

Đầu vào:

- `ratingstablename`: Tên bảng gốc.
- `numberofpartitions`: Số lượng phân mảnh (partition).
- `openconnection`: Kết nối tới CSDL.

Đầu ra: Tạo ra các bảng con `range_part0`, `range_part1`, ..., mỗi bảng chứa các dòng thuộc một khoảng rating nhất định.

Thiết kế:

- Tính toán khoảng giá trị cho mỗi partition.
- Tạo các bảng con tương ứng.
- Dùng truy vấn `INSERT INTO ... SELECT ... WHERE ...` để phân phối dữ liệu vào từng bảng con.

2.1.3 Hàm roundrobinpartition

Phân tích: Chia bảng `ratings` thành nhiều bảng con theo phương pháp luân phiên từng dòng (dòng 1 vào bảng 0, dòng 2 vào bảng 1,...).

Đầu vào:

- `ratingstablename`: Tên bảng gốc.
- `numberofpartitions`: Số lượng phân mảnh.
- `openconnection`: Kết nối tới CSDL.

Đầu ra: Tạo ra các bảng con `rrobin_part0`, `rrobin_part1`, ..., mỗi bảng chứa các dòng theo thứ tự luân phiên.

Thiết kế:

- Đánh số thứ tự các dòng trong bảng gốc.
- Chia đều các dòng vào các bảng con dựa trên chỉ số dòng và số partition.
- Đảm bảo mỗi bảng con nhận đúng các dòng theo quy tắc round-robin.

2.1.4 Hàm rangeinsert

Phân tích: Chèn một bản ghi mới vào bảng gốc và vào đúng bảng con range partition phù hợp với giá trị rating.

Đầu vào:

- `ratingstablename`: Tên bảng gốc.
- `userid`, `itemid`, `rating`: Thông tin bản ghi mới.
- `openconnection`: Kết nối tới CSDL.

Đầu ra: Bản ghi mới được thêm vào bảng gốc và đúng bảng con range partition.

Thiết kế:

- Chèn bản ghi vào bảng gốc.
- Xác định partition phù hợp dựa trên giá trị rating.
- Chèn bản ghi vào bảng con tương ứng.

2.1.5 Hàm roundrobininsert

Phân tích: Chèn một bản ghi mới vào bảng gốc và vào bảng con round-robin tiếp theo.

Đầu vào:

- `ratingstablename`: Tên bảng gốc.
- `userid`, `itemid`, `rating`: Thông tin bản ghi mới.
- `openconnection`: Kết nối tới CSDL.

Đầu ra: Bản ghi mới được thêm vào bảng gốc và đúng bảng con round-robin.

Thiết kế:

- Chèn bản ghi vào bảng gốc.
- Xác định bảng con round-robin tiếp theo dựa trên tổng số bản ghi đã có.
- Chèn bản ghi vào bảng con tương ứng.

2.2 Thiết kế giải pháp cho từng hàm chính

2.2.1 Hàm `loadratings`

Thiết kế:

- Đầu vào: tên bảng, đường dẫn file dữ liệu, kết nối CSDL.
- Đầu ra: bảng dữ liệu `ratings` đã được nạp đầy đủ.
- Hàm sử dụng lệnh `COPY` của PostgreSQL để nạp dữ liệu từ file vào bảng tạm, sau đó chuyển dữ liệu sang bảng chính. Việc dùng `COPY` giúp tốc độ nạp dữ liệu nhanh hơn rất nhiều so với việc đọc từng dòng và `INSERT` từng bản ghi, vì `COPY` tối ưu thao tác I/O và giảm số lần round-trip giữa Python và PostgreSQL.
- Sau khi nạp xong, bảng tạm được xóa để giải phóng bộ nhớ.

Phân tích hiệu năng: Việc sử dụng `COPY` và thao tác bulk-insert giúp giảm thời gian xử lý đáng kể, đặc biệt với file dữ liệu lớn (hàng trăm nghìn dòng).

2.2.2 Hàm `rangepartition`

Thiết kế:

- Đầu vào: tên bảng gốc, số lượng phân mảnh, kết nối CSDL.
- Đầu ra: các bảng con `range_part0`, `range_part1`, ...
- Hàm tính toán khoảng giá trị rating cho từng partition, tạo bảng con tương ứng, sau đó dùng một câu lệnh `INSERT INTO ... SELECT ... WHERE ...` để chuyển dữ liệu vào từng bảng con.
- Đối với partition đầu tiên, điều kiện là `rating >= min_range`, các partition sau là `rating > min_range`. Điều này đảm bảo không bị trùng lặp hoặc bỏ sót giá trị biên.

Phân tích hiệu năng: Việc dùng truy vấn SQL dạng `INSERT INTO ... SELECT ... WHERE ...` giúp PostgreSQL xử lý dữ liệu trực tiếp trên server, không cần chuyển dữ liệu về phía client, do đó tốc độ phân mảnh rất nhanh và tận dụng tối đa khả năng tối ưu hóa truy vấn của hệ quản trị CSDL.

2.2.3 Hàm `roundrobinpartition`

Thiết kế:

- Đầu vào: tên bảng gốc, số lượng phân mảnh, kết nối CSDL.
- Đầu ra: các bảng con `rrobin_part0`, `rrobin_part1`, ...
- Hàm sử dụng hàm cửa sổ `ROW_NUMBER()` của SQL để đánh số thứ tự từng dòng, sau đó chia đều các dòng vào các bảng con dựa trên phép chia lấy dư (`rnum % numberofpartitions`).
- Mỗi partition được tạo và nạp dữ liệu chỉ với một truy vấn SQL duy nhất.

Phân tích hiệu năng: Việc tận dụng `ROW_NUMBER()` và thực hiện phân phối dữ liệu hoàn toàn trên phía server giúp giảm thiểu thời gian xử lý, không cần lặp qua từng dòng ở phía Python, do đó tốc độ phân mảnh round-robin rất nhanh và ổn định, kể cả với dữ liệu lớn.

2.2.4 Hàm `rangeinsert`

Thiết kế:

- Đầu vào: tên bảng gốc, thông tin bản ghi mới (userid, itemid, rating), kết nối CSDL.
- Đầu ra: bản ghi mới được thêm vào bảng gốc và đúng bảng con range partition.
- Hàm xác định partition phù hợp dựa trên giá trị rating và số lượng partition hiện có (tính toán bằng công thức giống hàm `rangepartition`), sau đó chèn bản ghi vào cả bảng gốc và bảng con tương ứng.

Phân tích hiệu năng: Việc xác định partition bằng phép toán số học và chỉ thực hiện hai lệnh `INSERT` giúp thao tác chèn rất nhanh, không cần quét lại toàn bộ bảng dữ liệu.

2.2.5 Hàm `roundrobininsert`

Thiết kế:

- Đầu vào: tên bảng gốc, thông tin bản ghi mới (userid, itemid, rating), kết nối CSDL.
- Đầu ra: bản ghi mới được thêm vào bảng gốc và đúng bảng con round-robin.
- Hàm xác định partition tiếp theo dựa trên tổng số bản ghi hiện có trong bảng gốc (dùng `COUNT(*)`), sau đó chèn bản ghi vào bảng con tương ứng theo quy tắc round-robin.

Phân tích hiệu năng: Việc xác định bảng con chỉ cần một phép chia lấy dư, thao tác chèn chỉ thực hiện hai lệnh `INSERT`, do đó tốc độ xử lý rất nhanh và không bị ảnh hưởng bởi kích thước dữ liệu.

3 Cài đặt và kiểm thử chương trình

3.1 Cài đặt chương trình

3.1.1 Hàm `loadratings`

```python
def loadratings(ratingstablename, ratingsfilepath, openconnection):
    start = time.time()
    con = openconnection
    cur = con.cursor()

    # Kiểm tra xem bảng đã tồn tại chưa
    cur.execute(f"SELECT EXISTS (SELECT FROM information_schema.tables WHERE table_name = '{ratingstablename}');")
    if cur.fetchone()[0]:
        cur.execute(f"DROP TABLE {ratingstablename};")

    # Tạo bảng chính với schema đúng
    cur.execute(f"CREATE TABLE {ratingstablename} (userid INTEGER, movieid INTEGER, rating FLOAT);")

    # Tạo bảng tạm để tải dữ liệu
    temp_table = ratingstablename + "_temp"
    cur.execute(f"CREATE TABLE {temp_table} (userid INTEGER, extra1 CHAR, movieid INTEGER, extra2 CHAR, rating FLOAT, extra3 CHAR, timestamp BIGINT);")

    # Tải dữ liệu vào bảng tạm
    try:
        with open(ratingsfilepath, 'r') as f:
            cur.copy_from(f, temp_table, sep=':', columns=('userid', 'extra1', 'movieid', 'extra2', 'rating', 'extra3', 'timestamp'))

        # Chuyển dữ liệu từ bảng tạm sang bảng chính
        cur.execute(f"INSERT INTO {ratingstablename} (userid, movieid, rating) SELECT userid, movieid, rating FROM {temp_table};")

        # Xóa bảng tạm
        cur.execute(f"DROP TABLE {temp_table};")
    except Exception as e:
        print(f"Error loading data: {e}")
        cur.execute(f"DROP TABLE IF EXISTS {temp_table};")
        raise

    cur.close()
    con.commit()
    print(f"[loadratings] Thời gian xử lý: {time.time() - start:.4f} giây")
```

Hình 2: Hàm `loadratings`

Cụ thể chi tiết xử lý:

- Kiểm tra xem bảng đã tồn tại chưa nếu có thì xóa
- Tạo bảng chính ratings với schema đúng
- Tạo bảng tạm ratings_temp để parse dữ liệu theo định dạng đặc biệt với dấu ':'
- Dùng copy_from để nạp dữ liệu nhanh từ dữ liệu sang bảng tạm
- Chuyển dữ liệu từ bảng tạm sang bảng chính trên những trường mà bảng chính cần
- Xóa bảng tạm

3.1.2 Hàm `rangepartition`

```python
def rangepartition(ratingstablename, numberofpartitions, openconnection):
    start = time.time()
    if numberofpartitions < 1:
        raise ValueError("Number of partitions must be at least 1")

    con = openconnection
    cur = con.cursor()
    RANGE_TABLE_PREFIX = 'range_part'
    delta = 5.0 / numberofpartitions

    for i in range(numberofpartitions):
        table_name = f"{RANGE_TABLE_PREFIX}{i}"
        cur.execute(f"DROP TABLE IF EXISTS {table_name};")
        cur.execute(f"CREATE TABLE {table_name} (userid INTEGER, movieid INTEGER, rating FLOAT);")
        min_range = i * delta
        max_range = (i + 1) * delta
        # Sử dụng ">=" cho phân mảnh đầu tiên để bao gồm rating = 0
        if i == 0:
            cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) "
                        f"SELECT userid, movieid, rating FROM {ratingstablename} "
                        f"WHERE rating >= {min_range} AND rating <= {max_range};")
        else:
            cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) "
                        f"SELECT userid, movieid, rating FROM {ratingstablename} "
                        f"WHERE rating > {min_range} AND rating <= {max_range};")

    cur.close()
    con.commit()
    print(f"[rangepartition] Thời gian xử lý: {time.time() - start:.4f} giây")
```

Hình 3: Hàm `rangepartition`

Cụ thể chi tiết xử lý:

- Xác định độ rộng mỗi khoảng: delta = 5 / N
- Với mỗi phân mảnh i, tạo bảng range_part{i} và lấy dữ liệu từ bảng gốc:
  - Nếu i = 0: lấy rating >= min AND <= max
  - Nếu i > 0: lấy rating > min AND <= max

3.1.3 Hàm `roundrobinpartition`

```python
def roundrobinpartition(ratingstablename, numberofpartitions, openconnection):
    start = time.time()
    if numberofpartitions < 1:
        raise ValueError("Number of partitions must be at least 1")

    con = openconnection
    cur = con.cursor()
    RROBIN_TABLE_PREFIX = 'rrobin_part'

    for i in range(numberofpartitions):
        table_name = f"{RROBIN_TABLE_PREFIX}{i}"
        cur.execute(f"CREATE TABLE {table_name} (userid INTEGER, movieid INTEGER, rating FLOAT);")
        cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) "
                    f"SELECT userid, movieid, rating FROM "
                    f"(SELECT userid, movieid, rating, ROW_NUMBER() OVER () - 1 AS rnum FROM {ratingstablename}) t "
                    f"WHERE rnum % {numberofpartitions} = {i};")

    cur.close()
    con.commit()
    print(f"[roundrobinpartition] Thời gian xử lý: {time.time() - start:.4f} giây")
```

Hình 4: Hàm `roundrobinpartition`

Cụ thể chi tiết xử lý:

- Dùng hàm ROW_NUMBER() để gán số thứ tự mỗi dòng
- Chia đều theo số phân mảnh bằng phép chia dư
- Mỗi bảng rrobin_part{i} nhận các dòng row_number

3.1.4 Hàm `rangeinsert`

```python
def rangeinsert(ratingstablename, userid, itemid, rating, openconnection):
    start = time.time()
    if not (0 <= rating <= 5):
        raise ValueError("Rating must be between 0 and 5")

    con = openconnection
    cur = con.cursor()
    RANGE_TABLE_PREFIX = 'range_part'
    numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)
    delta = 5.0 / numberofpartitions

    # Xác định phân mảnh
    index = None
    for i in range(numberofpartitions):
        min_range = i * delta
        max_range = (i + 1) * delta
        if i == 0 and rating >= min_range and rating <= max_range:
            index = i
            break
        elif rating > min_range and rating <= max_range:
            index = i
            break
    if index is None:
        index = numberofpartitions - 1  # Trường hợp rating = 5

    table_name = f"{RANGE_TABLE_PREFIX}{index}"

    # Chèn vào bảng chính
    cur.execute(f"INSERT INTO {ratingstablename} (userid, movieid, rating) "
                f"VALUES ({userid}, {itemid}, {rating});")

    # Chèn vào phân mảnh
    cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) "
                f"VALUES ({userid}, {itemid}, {rating});")

    cur.close()
    con.commit()
    print(f"[rangeinsert] Thời gian xử lý: {time.time() - start:.6f} giây")
```

Hình 5: Hàm `rangeinsert`

Cụ thể chi tiết xử lý:

- Tính index của phân mảnh bằng cách xác định đoạn [min, max] chứa rating
- Chèn bản ghi vào bảng gốc và bảng phân mảnh tương ứng

3.1.5 Hàm `roundrobininsert`

```python
def roundrobininsert(ratingstablename, userid, itemid, rating, openconnection):
    start = time.time()
    if not (0 <= rating <= 5):
        raise ValueError("Rating must be between 0 and 5")

    con = openconnection
    cur = con.cursor()
    RROBIN_TABLE_PREFIX = 'rrobin_part'

    # Chèn vào bảng chính
    cur.execute(f"INSERT INTO {ratingstablename} (userid, movieid, rating) "
                f"VALUES ({userid}, {itemid}, {rating});")

    # Lấy số thứ tự bản ghi (dùng meta-data hoặc biến tạm)
    cur.execute(f"SELECT COUNT(*) FROM {ratingstablename};")
    total_rows = cur.fetchone()[0]
    numberofpartitions = count_partitions(RROBIN_TABLE_PREFIX, openconnection)
    index = (total_rows - 1) % numberofpartitions
    table_name = f"{RROBIN_TABLE_PREFIX}{index}"

    # Chèn vào phân mảnh
    cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) "
                f"VALUES ({userid}, {itemid}, {rating});")

    cur.close()
    con.commit()
    print(f"[roundrobininsert] Thời gian xử lý: {time.time() - start:.6f} giây")
```

Hình 6: Hàm `roundrobininsert`

Cụ thể chi tiết xử lý:

- Lấy tổng số dòng trong bảng gốc -> total_rows
- Tính index = (total_rows – 1)
- Chèn vào bảng gốc và bảng rrobin_part{index}

3.2 Kiểm thử chương trình

3.2.1 Kiểm thử trên tập test_data.dat

Đặc điểm của tập dữ liệu: có 20 dòng
Cấu hình test case:

- Hàm `loadratings`: kiểm tra xem khi nạp vào có đủ 20 dòng không
- Hàm `rangepartition`: với đầu vào số phân mảnh là 5, kiểm tra sau khi thực hiện hàm thì có đủ 5 bảng thỏa mãn có dạng 'range_part%' không
- Hàm `roundrobinpartition`: với đầu vào số phân mảnh là 5, kiểm tra sau khi thực hiện hàm thì có đủ 5 bảng thỏa mãn có dạng 'rrobin_part%' không
- Hàm `rangeinsert`: với đầu vào là (userid, movieid, rating) = (100, 2, 3) thì nó có ở phân mảnh số 3 tức bảng range_part2 không

- Hàm `roundrobininsert`: với đầu vào là (userid, movieid, rating) = (100, 1, 3) thì nó có ở phân mảnh số 1 tức bảng range_part0 không

Khi chạy chương trình ta thu được kết quả tất cả các test case đều pass như hình dưới:

```
PS D:\3Y2S\csdlpt> & d:/3Y2S/csdlpt/.venv/Scripts/python.exe d:/3Y2S/csdlpt/csdlpt/Assignment1Tester.py
A database named "dds_assgn1" already exists
[loadratings] Thời gian xử lý: 0.0140 giây
loadratings function pass!
[rangepartition] Thời gian xử lý: 0.0050 giây
rangepartition function pass!
[rangeinsert] Thời gian xử lý: 0.003015 giây
rangeinsert function pass!
[loadratings] Thời gian xử lý: 0.0097 giây
[roundrobinpartition] Thời gian xử lý: 0.0120 giây
roundrobinpartition function pass!
[roundrobininsert] Thời gian xử lý: 0.002003 giây
roundrobininsert function pass!
Press enter to Delete all tables?
PS D:\3Y2S\csdlpt>
```

Hình 7: Kiểm thử trên tập test_data.dat bằng code nhóm thực hiện

3.2.2 Kiểm thử trên tập ratings.dat

Đặc điểm của tập dữ liệu: có 10000054 dòng
Cấu hình test case:

- Hàm `loadratings`: kiểm tra xem khi nạp vào có đủ 10000054 dòng không
- Hàm `rangepartition`: với đầu vào số phân mảnh là 5, kiểm tra sau khi thực hiện hàm thì có đủ 5 bảng thỏa mãn có dạng 'range_part%' không
- Hàm `roundrobinpartition`: với đầu vào số phân mảnh là 5, kiểm tra sau khi thực hiện hàm thì có đủ 5 bảng thỏa mãn có dạng 'rrobin_part%' không
- Hàm `rangeinsert`: với đầu vào là (userid, movieid, rating) = (100, 2, 3) thì nó có ở phân mảnh số 3 tức bảng range_part2 không
- Hàm `roundrobininsert`: với đầu vào là (userid, movieid, rating) = (100, 1, 3) thì nó có ở phân mảnh số 5 tức bảng range_part4 không

Khi chạy chương trình ta thu được kết quả tất cả các test case đều pass như hình dưới:

```
PS D:\3Y2S\csdlpt> & d:/3Y2S/csdlpt/.venv/Scripts/python.exe d:/3Y2S/csdlpt/csdlpt/Assignment1Tester.py
A database named "dds_assgn1" already exists
[loadratings] Thời gian xử lý: 20.9004 giây
loadratings function pass!
[rangepartition] Thời gian xử lý: 20.9893 giây
rangepartition function pass!
[rangeinsert] Thời gian xử lý: 0.008063 giây
rangeinsert function pass!
[loadratings] Thời gian xử lý: 22.8225 giây
[roundrobinpartition] Thời gian xử lý: 32.2945 giây
roundrobinpartition function pass!
[roundrobininsert] Thời gian xử lý: 0.285790 giây
roundrobininsert function pass!
Press enter to Delete all tables?
```

Hình 8: Kiểm thử trên tập test_data.dat bằng code nhóm thực hiện

Hình 9: Kiểm thử trên tập test_data.dat bằng code mẫu của thầy đã thêm tính thời gian

So sánh hiệu năng giữa code của nhóm và code mẫu của thầy dựa trên thời gian thực hiện(s)

Hàm	Hiệu năng trên code của nhóm (s)	Hiệu năng trên code mẫu của thầy (s)
`loadratings` lần 1	20.9004	27.8677
`rangepartition`	20.9893	20.4091
`rangeinsert`	0.008063	0.003
`loadratings` lần 2	22.8225	32.1877
`roundrobinpartition`	32.2945	33.5454
`roundrobininsert`	0.285790	0.242677

Bảng 1: So sánh hiệu năng giữa code của nhóm và code mẫu của thầy

Từ bảng so sánh hiệu năng giữa code của nhóm và code mẫu của thầy, có thể thấy rằng thời gian thực thi của hai giải pháp là khá tương đương ở hầu hết các hàm. Đặc biệt, ở các hàm xử lý dữ liệu lớn như `loadratings`, `rangepartition` và `roundrobinpartition`, code của nhóm có tốc độ xử lý nhanh hơn hoặc tương đương so với code mẫu. Điều này cho thấy nhóm đã tận dụng tốt các kỹ thuật tối ưu như sử dụng lệnh `COPY` để nạp dữ liệu hàng loạt và các truy vấn SQL tổng hợp để phân mảnh dữ liệu trực tiếp trên server, giảm thiểu thao tác lặp ở phía Python.

Ở các hàm chèn bản ghi đơn lẻ như `rangeinsert` và `roundrobininsert`, thời gian thực thi của code nhóm và code mẫu đều rất nhỏ, sự chênh lệch không đáng kể và chủ yếu phụ thuộc vào tốc độ truy vấn của hệ quản trị cơ sở dữ liệu tại thời điểm kiểm thử.

Nhìn chung, giải pháp của nhóm đảm bảo hiệu năng tốt, tận dụng tối đa sức mạnh của PostgreSQL, đồng thời vẫn đảm bảo tính đúng đắn và dễ mở rộng cho các bài toán lớn hơn trong thực tế.

4 Kết luận và hướng phát triển

Ưu điểm:

- Đáp ứng đầy đủ yêu cầu đề bài
- Dữ liệu được xử lý chính xác, kiểm thử pass toàn bộ
- Tốc độ xử lý truy vấn nhanh
- Sử dụng PostgreSQL hiệu quả qua COPY và ROW_NUMBER()

Hạn chế:

- Không có xử lý rollback nếu chèn lỗi
- Không lưu trạng thái lần chèn cuối cùng trong Round Robin

Đề xuất phát triển

- Thêm bảng metadata để quản lý trạng thái chèn round robin chính xác hơn
- Giao diện web để xem nội dung phân mảnh
- Hỗ trợ phân mảnh dọc và kết hợp

5 Phân chia công việc

Thành viên	**Công việc thực hiện**
Nguyễn Quang Dũng	- Thiết kế và cài đặt hàm - Viết tài liệu hướng dẫn sử dụng chương trình - Feedback báo cáo
Nguyễn Tuấn Nam	- Kiểm thử chương trình - Feedback báo cáo
Lê Đăng Ninh	- Tổng hợp và viết báo cáo

Bảng 2: Bảng phân chia công việc giữa các thành viên