

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



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

Nhóm lớp: 9
Nhóm bài tập: 21

Nguyễn Xuân Việt	MSSV: B22DCCN898
Lê Tiến Đạt	MSSV: B22DCCN188
Lê Văn Minh	MSSV: B22DCCN534

Giảng viên hướng dẫn : Kim Ngọc Bách

Hà Nội – 2025

Mục Lục

1. Mở đầu.....	3
2. Trình bày và giải thích cách giải quyết vấn đề	4
2.1. loadratings(ratingtablename, ratingsfilepath, openconnection)	4
2.2. rangepartition(ratingtablename, numberofpartitions, openconnection)...	5
2.3. roundrobinpartition(ratingtablename, numberofpartitions, openconnection).....	8
2.4. rangeinsert(ratingtablename, userid, itemid, rating, openconnection)...	10
2.5. roundrobininsert(ratingtablename, userid, itemid, rating, openconnection).....	12
3. Phân chia công việc	14

1. Mở đầu

Bài tập lớn này tập trung vào việc mô phỏng các phương pháp phân mảnh dữ liệu ngang trên một hệ quản trị cơ sở dữ liệu quan hệ (PostgreSQL). Mục tiêu chính là xây dựng một bộ các hàm bằng Python để thực hiện các công việc sau:

- **Tải dữ liệu:** Đọc dữ liệu từ tệp ratings.dat (từ bộ dữ liệu MovieLens 10M) và nạp vào một bảng quan hệ trong PostgreSQL.
- **Phân mảnh theo khoảng (Range Partitioning):** Chia bảng dữ liệu gốc thành N phân mảnh dựa trên các khoảng giá trị của thuộc tính Rating.
- **Phân mảnh kiểu vòng tròn (Round-Robin Partitioning):** Chia bảng dữ liệu gốc thành N phân mảnh theo phương pháp chia đều tuần tự từng dòng dữ liệu.
- **Chèn dữ liệu:** Cập nhật các phân mảnh tương ứng khi có một bản ghi mới được chèn vào bảng gốc.

Giải pháp được xây dựng bằng ngôn ngữ Python, sử dụng thư viện psycopg2 để tương tác với cơ sở dữ liệu PostgreSQL.

2. Trình bày và giải thích cách giải quyết vấn đề

Dưới đây là phần giải thích về kiến trúc và logic của từng hàm.

2.1. loadratings(ratingtablename, ratingsfilepath, openconnection)

Hàm này là bước khởi đầu, có nhiệm vụ tải toàn bộ dữ liệu từ tệp ratings.dat vào cơ sở dữ liệu.

- **Mục đích:** Tạo bảng ratings có 3 cột UserID(int), MovieID(int), Rating(float) và tải dữ liệu từ file được truyền vào vào bảng ratings vừa tạo.
- **Triển khai thuật toán loadratings:**

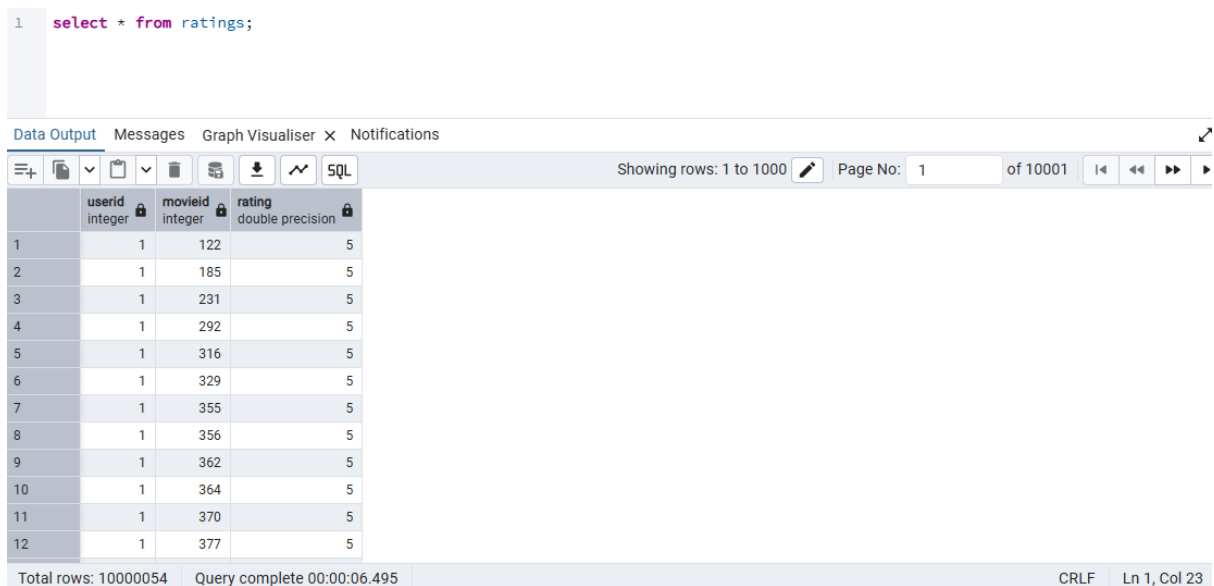
```
def loadratings(ratingtablename, ratingsfilepath, openconnection):  
    """  
    Function to load data in @ratingsfilepath file to a table called @ratingtablename.  
    """  
    create_db(DATABASE_NAME)  
    con = openconnection  
    cur = con.cursor()  
  
    cur.execute("DROP TABLE IF EXISTS " + ratingtablename + ";")  
  
    cur.execute("create table " + ratingtablename + "(userid integer, extra1 char, movieid  
integer, extra2 char, rating float, extra3 char, timestamp bigint);")  
  
    cur.copy_from(open(ratingsfilepath), ratingtablename, sep=':')  
    cur.execute("alter table " + ratingtablename + " drop column extra1, drop column extra2,  
drop column extra3, drop column timestamp;")  
  
    cur.close()  
    con.commit()
```

- **Phân tích chi tiết:**

1. **cur.execute("DROP TABLE IF EXISTS ..."):** Lệnh này đảm bảo môi trường làm việc luôn "sạch sẽ" trước mỗi lần chạy. Tránh lỗi bảng đã tồn tại.
2. **cur.execute("create table ... (... , extra1 char, ...)"):** Cấu trúc bảng tạm thời được tạo ra để khớp chính xác với định dạng của tệp ratings.dat. Dữ liệu trong tệp có dạng UserID::MovieID::Rating::Timestamp. Ký tự '::' được xem là dấu phân cách. Bằng cách tạo các cột extra để "hấp thụ" các ký tự khoảng trống giữa hai dấu phân cách, nhờ đó ta có thể sử dụng hàm copy_from một cách trực tiếp mà không cần tiền xử lý tệp.

3. **cur.copy_from(open(ratingsfilepath), ratingtablename, sep=',')**: Đây là điểm mấu chốt về hiệu năng của hàm. Thay vì đọc từng dòng của tệp 10 triệu bản ghi và thực hiện 10 triệu câu lệnh INSERT (việc này sẽ rất chậm do chi phí giao tiếp mạng và xử lý giao dịch cho mỗi lệnh), **copy_from** là một lệnh chuyên dụng của PostgreSQL. Nó đọc dữ liệu dưới dạng một luồng (stream) và chèn hàng loạt (bulk insert) vào bảng. Quá trình này giảm thiểu đáng kể chi phí và nhanh hơn nhiều lần.
 4. **cur.execute("alter table ... drop column ...")**: Sau khi dữ liệu đã nằm an toàn trong cơ sở dữ liệu, chúng ta mới tiến hành "dọn dẹp" bảng bằng cách loại bỏ các cột tạm thời không cần thiết. Thao tác này chỉ thực hiện một lần trên toàn bộ bảng, hiệu quả hơn nhiều so với việc xử lý chuỗi trên từng dòng dữ liệu trong Python.
 5. **con.commit()**: Xác nhận toàn bộ giao dịch. Tất cả các lệnh từ DROP, CREATE, COPY, đến ALTER sẽ được thực thi một cách nguyên tử.
- **Kết quả**: thực hiện truy vấn bảng rating, thu được kết quả sau:

```
1 select * from ratings;
```



	userid integer	movieid integer	rating double precision
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

Total rows: 10000054 Query complete 00:00:06.495 CRLF Ln 1, Col 23

Hình 2.1. Bảng Ratings

Sau khi thực hiện hàm **loadrating()**, dữ liệu từ file **ratings.dat** được tải đầy đủ vào bảng **ratings** với 3 cột **UserID**, **MovieID** và **Rating**.

2.2. **rangepartition(ratingtablename, numberofpartitions, openconnection)**

Hàm này triển khai phương pháp phân mảnh dựa trên các khoảng giá trị của cột **rating**.

- **Mục đích:** Tạo N phân mảnh ngang của bảng Ratings dựa trên những khoảng giá trị đồng đều của thuộc tính Rating.
- **Triển khai thuật toán rangepartition:**

```
def rangepartition(ratingtablename, numberofpartitions, openconnection):
    """
    Function to create partitions of main table based on range of ratings.
    """
    con = openconnection
    cur = con.cursor()
    delta = 5 / numberofpartitions
    RANGE_TABLE_PREFIX = 'range_part'
    for i in range(0, numberofpartitions):
        minRange = i * delta
        maxRange = minRange + delta
        table_name = RANGE_TABLE_PREFIX + str(i)
        cur.execute("create table " + table_name + " (userid integer, movieid integer, rating float);")

        if i == 0:
            cur.execute("insert into " + table_name + " (userid, movieid, rating) select
userid, movieid, rating from " + ratingtablename + " where rating >= " + str(minRange) + "
and rating <= " + str(maxRange) + ";")

        else:
            cur.execute("insert into " + table_name + " (userid, movieid, rating) select
userid, movieid, rating from " + ratingtablename + " where rating > " + str(minRange) + " and
rating <= " + str(maxRange) + ";")

    cur.close()
    con.commit()
```

- **Phân tích chi tiết:**
 1. **delta = 5.0 / numberofpartitions:** Tính toán "độ rộng" của mỗi khoảng. Ví dụ, với 5 phân mảnh, mỗi phân mảnh sẽ chứa các bản ghi có rating trong một khoảng rộng $5.0 / 5 = 1.0$.
 2. **Vòng lặp for i in range(0, numberofpartitions):** Cấu trúc này lặp qua từng phân mảnh cần tạo.
 3. **Tính toán minRange và maxRange:** Trong mỗi vòng lặp, cận dưới và cận trên của khoảng được xác định.
 - $\text{minRange} = i * \text{delta}$
 - $\text{maxRange} = \text{minRange} + \text{delta}$

4. Tạo bảng và sao chép dữ liệu:

- `create table range_part{i} (...)`: Tạo bảng cho phân mảnh hiện tại.
- `insert into range_part{i} ... select ... from {ratingtablename} where ...`: Sao chép dữ liệu từ bảng gốc.

5. Xử lý các giá trị biên (Crucial Point): Logic trong mệnh đề WHERE là cực kỳ quan trọng để đảm bảo tính **đầy đủ** (completeness) và **không giao nhau** (disjointness) của các phân mảnh.

- Với phân mảnh đầu tiên ($i == 0$): `WHERE rating >= {minRange} AND rating <= {maxRange}`. Chúng ta bao gồm cả cận dưới (0.0).
- Với các phân mảnh còn lại (else): `WHERE rating > {minRange} AND rating <= {maxRange}`. Chúng ta sử dụng `>` thay vì `>=` để loại trừ giá trị `minRange` vì nó đã được bao gồm trong phân mảnh ngay trước đó (`maxRange` của phân mảnh $i - 1$). Điều này ngăn chặn một bản ghi có `rating` nằm ngay tại biên bị chép vào cả hai phân mảnh.
- **Ví dụ (N=2)**: $\text{delta} = 2.5$.
 - Partition 0: `WHERE rating >= 0.0 AND rating <= 2.5`.
 - Partition 1: `WHERE rating > 2.5 AND rating <= 5.0`. Một bản ghi có `rating=2.5` sẽ chỉ thuộc về Partition 0.

- **Kết quả**: thực hiện kiểm tra trên các phân mảnh.



```
1 select '0' as part, count(*) as totalrow, count(*) filter (where rating < 0 or rating > 1) as invalid_rows from range_part0
2 union
3 select '1' as part, count(*) as totalrow, count(*) filter (where rating <= 1 or rating > 2) as invalid_rows from range_part1
4 union
5 select '2' as part, count(*) as totalrow, count(*) filter (where rating <= 2 or rating > 3) as invalid_rows from range_part2
6 union
7 select '3' as part, count(*) as totalrow, count(*) filter (where rating <= 3 or rating > 4) as invalid_rows from range_part3
8 union
9 select '4' as part, count(*) as totalrow, count(*) filter (where rating <= 4 or rating > 5) as invalid_rows from range_part4
10
```

	part	totalrow	invalid_rows
	text	bigint	bigint
1	0	479168	0
2	1	908584	0
3	2	2726854	0
4	3	3755614	0
5	4	2129834	0

Hình 2.2. Kiểm tra trên các phân mảnh

Tất cả các bản ghi trong bảng Ratings được lưu vào 5 phân vùng với các khoảng giá trị của thuộc tính `rating` và không bản ghi nào bị lưu sai phân vùng.

2.3. roundrobinpartition(ratingtablename, numberofpartitions, openconnection)

- **Mục đích:** Phân mảnh dữ liệu dựa theo phương pháp roundrobin.
- **Triển khai thuật toán roundrobinpartition():**

```
def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
    """
    Partition the table using round robin
    """
    con = openconnection
    cur = con.cursor()
    RROBIN_TABLE_PREFIX = 'rrobin_part'

    cur.execute("DROP TABLE IF EXISTS roundrobin_metadata;")
    cur.execute("CREATE TABLE roundrobin_metadata (next_insert_partition INT);")
    cur.execute("INSERT INTO roundrobin_metadata VALUES (0);")

    for i in range(numberofpartitions):
        cur.execute(f"DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i};")
        cur.execute(f"CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (userid INTEGER, movieid INTEGER, rating FLOAT);")

    cur.execute(f"""
        WITH numbered AS (
            SELECT userid, movieid, rating, ROW_NUMBER() OVER () as rn FROM {ratingtablename}
        )
        SELECT * INTO TEMP temp_rr FROM numbered;
    """)

    for i in range(numberofpartitions):
        cur.execute(f"""
            INSERT INTO {RROBIN_TABLE_PREFIX}{i}(userid, movieid, rating)
            SELECT userid, movieid, rating FROM temp_rr WHERE MOD(rn - 1,
{numberofpartitions}) = {i};
        """)

    cur.execute("DROP TABLE temp_rr;")
    cur.close()
    con.commit()
```


- **Phân tích chi tiết:**

1. **Sử dụng ROW_NUMBER():** Một cách tiếp cận đơn giản là đọc từng dòng từ bảng gốc trong Python, và dùng một biến đếm để quyết định chèn vào phân mảnh nào. Tuy nhiên, cách này rất chậm với dữ liệu lớn. Giải pháp tối ưu là đẩy toàn bộ logic xử lý xuống cho CSDL.
2. **WITH numbered AS (SELECT ..., ROW_NUMBER() OVER () as rn FROM ...):** Lệnh này tạo ra một Common Table Expression (CTE) tên là **numbered**. Hàm cửa sổ ROW_NUMBER() OVER () sẽ gán một số thứ tự duy nhất (rn) cho mỗi dòng trong bảng Ratings.
3. **SELECT * INTO TEMP temp_rr FROM numbered;** Kết quả của CTE được lưu vào một bảng tạm thời temp_rr. Việc này giúp tăng hiệu suất cho các bước sau vì CSDL không cần phải tính toán lại ROW_NUMBER nhiều lần.
4. **Phép toán Modulo (%):** Vòng lặp for i in range(numberofpartitions) giờ đây thực hiện các lệnh INSERT hàng loạt.
 - `INSERT INTO rrobin_part{i} ... SELECT ... FROM temp_rr WHERE MOD(rn - 1, {numberofpartitions}) = {i};`
 - Logic này là trái tim của Round Robin. $rn - 1$ để bắt đầu từ 0. Phép toán MOD (modulo) sẽ trả về số dư khi chia số thứ tự dòng cho số phân mảnh.
 - **Ví dụ (N=3):**
 - Dòng có $rn=1$: $MOD(0, 3) = 0$ -> vào Partition 0.
 - Dòng có $rn=2$: $MOD(1, 3) = 1$ -> vào Partition 1.
 - Dòng có $rn=3$: $MOD(2, 3) = 2$ -> vào Partition 2.
 - Dòng có $rn=4$: $MOD(3, 3) = 0$ -> quay lại Partition 0.
 - Quá trình này phân phối các dòng một cách tuần tự và đồng đều mà không cần sự can thiệp của logic phía client.
5. **Quản lý Metadata:** Hàm này cũng tạo ra bảng roundrobin_metadata và chèn giá trị 0. Bảng này đóng vai trò như một "bộ đếm toàn cục", ghi nhớ phân mảnh nào sẽ nhận bản ghi INSERT tiếp theo, điều này rất cần thiết cho hàm roundrobininsert.

- **Kết quả sau khi thực hiện phân mảnh:**

Query Query History

```

1 select '0' as part, count(*) as totalrow from rrobin_part0
2 union
3 select '1' as part, count(*) as totalrow from rrobin_part1
4 union
5 select '2' as part, count(*) as totalrow from rrobin_part2
6 union
7 select '3' as part, count(*) as totalrow from rrobin_part3
8 union
9 select '4' as part, count(*) as totalrow from rrobin_part4;

```

Data Output Messages Notifications

Showing rows: 1 to 5 Page No: 1 of 1

	part text	totalrow bigint
1	0	2000011
2	1	2000011
3	2	2000011
4	3	2000011
5	4	2000010

Hình 2.3. Phân mảnh dựa trên roundrobin

Dữ liệu trong bảng Ratings được phân bố đồng đều cho các phân mảnh.

2.4. rangeinsert(ratingtablename, userid, itemid, rating, openconnection)

- **Mục đích:** Thực hiện thêm 1 bản ghi mới vào bảng Ratings và vào đúng phân mảnh dựa trên giá trị của rating.
- **Triển khai thuật toán rangeinsert():**

```

def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
    """
    Function to insert a new row into the main table and specific partition based on range
    rating.
    """
    con = openconnection
    cur = con.cursor()
    RANGE_TABLE_PREFIX = 'range_part'
    numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)
    delta = 5 / numberofpartitions
    index = int(rating / delta)
    if rating % delta == 0 and index != 0:
        index = index - 1
    table_name = RANGE_TABLE_PREFIX + str(index)
    cur.execute("INSERT INTO " + ratingtablename + "(userid, movieid, rating) values (" +
    str(userid) + "," + str(itemid) + "," + str(rating) + ");")
    cur.execute("insert into " + table_name + "(userid, movieid, rating) values (" +
    str(userid) + "," + str(itemid) + "," + str(rating) + ");")
    cur.close()
    con.commit()

```

- **Phân tích chi tiết:**

1. **INSERT INTO {ratingtablename} ...:** Bước đầu tiên là chèn vào bảng gốc để đảm bảo tính toàn vẹn của dữ liệu tổng thể.
2. **numberofpartitions = count_partitions(...):** Hàm cần biết có bao nhiêu phân mảnh đang tồn tại để tính toán delta một cách chính xác. Thay vì truyền N như một tham số, hàm tự động đếm số lượng bảng có tiền tố range_part để trở nên linh hoạt hơn.
3. **delta = 5.0 / numberOfpartitions:** Tính lại độ rộng khoảng, giống như trong hàm rangepartition.
4. **index = int(rating / delta):** Đây là phép toán cốt lõi để xác định phân mảnh đích. Bằng cách chia rating cho delta và lấy phần nguyên, chúng ta có thể tìm ra chỉ số của khoảng mà rating thuộc về.
 - **Ví dụ (N=5, delta=1.0):**
 - Nếu rating = 3.5, $\text{int}(3.5 / 1.0) = \text{int}(3.5) = 3$. Bản ghi thuộc về range_part3.
 - Nếu rating = 4.9, $\text{int}(4.9 / 1.0) = \text{int}(4.9) = 4$. Bản ghi thuộc về range_part4.
5. **Xử lý biên if rating % delta == 0 and index != 0: index = index - 1:** Xét trường hợp N=5, delta=1.0. Nếu rating=3.0, công thức trên sẽ cho index=3. Tuy nhiên, theo quy ước (min, max], khoảng của range_part2 là (2.0, 3.0] và range_part3 là (3.0, 4.0]. Do đó, rating=3.0 phải thuộc về range_part2. Điều kiện này phát hiện ra trường hợp rating là biên trên của một khoảng (rating % delta == 0) và lùi index về 1 để đặt bản ghi vào đúng phân mảnh.
6. **INSERT INTO range_part{index} ...:** Sau khi đã xác định đúng index, bản ghi được chèn vào bảng phân mảnh tương ứng.

- **Thêm các bản ghi vào các phân mảnh bằng hàm rangeinsert():**

```
MyAssignment.rangeinsert(RATINGS_TABLE, 100, 2, 0, conn)
MyAssignment.rangeinsert(RATINGS_TABLE, 100, 3, 1, conn)
MyAssignment.rangeinsert(RATINGS_TABLE, 100, 4, 3, conn)
MyAssignment.rangeinsert(RATINGS_TABLE, 100, 5, 4, conn)
MyAssignment.rangeinsert(RATINGS_TABLE, 100, 6, 5, conn)
```

- **Kết quả:** kiểm tra dữ liệu trên các phân mảnh sau khi thêm các bản ghi mới

Query		Query History
1	select '0' as part, count(*) as totalrow, count(*) filter (where rating < 0 or rating > 1) as invalid_rows from range_part0	
2	union	
3	select '1' as part, count(*) as totalrow, count(*) filter (where rating <= 1 or rating > 2) as invalid_rows from range_part1	
4	union	
5	select '2' as part, count(*) as totalrow, count(*) filter (where rating <= 2 or rating > 3) as invalid_rows from range_part2	
6	union	
7	select '3' as part, count(*) as totalrow, count(*) filter (where rating <= 3 or rating > 4) as invalid_rows from range_part3	
8	union	
9	select '4' as part, count(*) as totalrow, count(*) filter (where rating <= 4 or rating > 5) as invalid_rows from range_part4	
10		

Data Output		Messages	Graph Visualiser	Notifications
Showing rows: 1 to 5 Page No: 1 of 1				
	part text	totalrow bigint	invalid_rows bigint	
1	0	479170	0	
2	1	908584	0	
3	2	2726855	0	
4	3	3755615	0	
5	4	2129835	0	

Hình 2.4. Dữ liệu trên các phân mảnh

Sau khi thêm, phân mảnh 0 có thêm 2 bản ghi có ratings là 0 và 1, phân mảnh 2, 3, 4 được thêm 1 bản ghi mới với ratings là 3, 4, 5. Không có bản ghi nào bị ghi vào sai phân mảnh.

2.5. roundrobininsert(ratingtablename, userid, itemid, rating, openconnection)

Hàm này chèn một bản ghi mới theo cơ chế Round Robin, sử dụng trạng thái được lưu trong bảng metadata.

- **Mục đích:** Duy trì sự phân bố đều dữ liệu khi có các bản ghi mới được thêm vào.
- **Triển khai thuật toán roundrobininsert():**

```
def roundrobininsert(ratingtablename, userid, itemid, rating, openconnection):
    """
    Insert a new row into the main table and specific round robin partition based on metadata.
    """
    con = openconnection
    cur = con.cursor()
    RROBIN_TABLE_PREFIX = 'rrobin_part'

    cur.execute("INSERT INTO " + ratingtablename + "(userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
    cur.execute("SELECT next_insert_partition FROM roundrobin_metadata;")
    current_index = cur.fetchone()[0]

    numberofpartitions = count_partitions(RROBIN_TABLE_PREFIX, openconnection)

    table_name = RROBIN_TABLE_PREFIX + str(current_index)
    cur.execute("INSERT INTO " + table_name + "(userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
```

```
next_index = (current_index + 1) % numberofpartitions
cur.execute("UPDATE roundrobin_metadata SET next_insert_partition = %s;", (next_index,))

cur.close()
con.commit()
```

- **Phân tích chi tiết:** Toàn bộ logic của hàm này phải được coi là một giao dịch nguyên tử (atomic transaction).

1. **INSERT INTO {ratingstablename} ...:** Chèn vào bảng gốc Ratings.
2. **SELECT next_insert_partition FROM roundrobin_metadata:** Đọc "bộ đếm" toàn cục từ bảng metadata. Đây là bước **ĐỌC** trạng thái hiện tại.
3. **current_index = cur.fetchone()[0]:** Lấy giá trị của bộ đếm.
4. **INSERT INTO rrobin_part{current_index} ...:** Sử dụng giá trị vừa đọc được để chèn bản ghi vào đúng phân mảnh.
5. **next_index = (current_index + 1) % numberofpartitions:** Tính toán giá trị **MỚI** cho bộ đếm.
6. **UPDATE roundrobin_metadata SET next_insert_partition = ...:** Cập nhật lại bộ đếm với giá trị mới. Đây là bước **GHI** lại trạng thái.
7. **con.commit():** Lệnh này cực kỳ quan trọng. Nó đảm bảo rằng tất cả các thao tác trên (chèn vào bảng chính, chèn vào phân mảnh, cập nhật metadata) hoặc là thành công tất cả, hoặc thất bại tất cả. Điều này ngăn ngừa các trường hợp lỗi như: đã chèn bản ghi nhưng chưa kịp cập nhật bộ đếm, dẫn đến lần chèn tiếp theo sẽ bị sai. Việc sử dụng bảng metadata trong CSDL đảm bảo tính nhất quán ngay

- **Thêm các bản ghi vào các phân mảnh bằng roundrobininsert():**

```
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 2, 1, conn)
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 3, 2, conn)
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 4, 3, conn)
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 5, 4, conn)
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 6, 5, conn)
MyAssignment.roundrobininsert(RATINGS_TABLE, 100, 7, 5, conn)
```

- **Kết quả khi thực hiện thêm bản ghi:**

Query Query History		
1	select '0' as part, count(*) as totalrow from rrobin_part0	
2	union	
3	select '1' as part, count(*) as totalrow from rrobin_part1	
4	union	
5	select '2' as part, count(*) as totalrow from rrobin_part2	
6	union	
7	select '3' as part, count(*) as totalrow from rrobin_part3	
8	union	
9	select '4' as part, count(*) as totalrow from rrobin_part4;	

Data Output Messages Notifications		
Showing rows: 1 to 5	Page No: 1	of 1
part	totalrow	
text	bigint	
1	0	2000013
2	1	2000012
3	2	2000012
4	3	2000012
5	4	2000011

Hình 2.5. Thực hiện chèn dữ liệu vào phân mảnh

Việc chèn được bắt đầu từ phân mảnh đầu tiên(phân mảnh 0), qua 6 lần chèn thì phân mảnh 0 có thêm 2 bản ghi, các phân mảnh còn lại có thêm 1 bản ghi.

3. Phân chia công việc

	Nhiệm vụ
Nguyễn Xuân Việt	Kiểm thử, viết báo cáo
Lê Tiến Đạt	Triển khai thuật toán
Lê Văn Minh	Triển khai thuật toán, kiểm thử