

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



BÁO CÁO BÀI TẬP LỚN
CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên: Kim Ngọc Bách

Nhóm: 10

Nhóm BTL: 14

Nguyễn Duy Hoàng - B22DCCN334

Phạm Như Hoàng - B22DCCN344

Lê Khánh Cường - B22DCCN093

Hà Nội - 2025

Phân chia công việc

Họ và tên	Mã SV	Nội dung
Nguyễn Duy Hoàng	B22DCCN334	Tổng hợp bài làm; Load_Ratings()
Phạm Như Hoàng	B22DCCN344	Range_Partition(); Range_Insert()
Lê Khánh Cường	B22DCCN093	RoundRobin_Partition(); RoundRobin_Insert()

MỤC LỤC

GIỚI THIỆU CHUNG	1
1. Giới thiệu chung.....	1
2. Cài đặt môi trường.....	1
I. Load_Ratings()	2
1. Yêu cầu.....	2
2. Bài làm	2
3. Kết quả.....	5
II. Range_Partition().....	5
1. Yêu cầu.....	5
2. Bài làm	6
3. Kết quả và kiểm thử.....	10
III. RoundRobin_Partition().....	16
1. Yêu cầu.....	16
2. Bài làm	16
3. Kết quả.....	20
IV. RoundRobin_Insert()	26
1. Yêu cầu.....	26
2. Bài làm	26
3. Kết quả.....	30
V. Range_Insert().....	37
1. Yêu cầu.....	37
2. Bài làm	37
3. Kết quả và kiểm thử.....	41
VI. Tổng kết và kết luận.....	45
1. Đánh giá thuật toán	45
2. Hạn chế và hướng phát triển	46
3. Kết luận.....	46

GIỚI THIỆU CHUNG

1. Giới thiệu chung

Với sự bùng nổ của dữ liệu đánh giá phim từ các nền tảng như MovieLens, việc thiết kế hệ thống lưu trữ và truy vấn hiệu quả 10 triệu bản ghi đòi hỏi giải pháp tối ưu cả về tốc độ và khả năng mở rộng. Phân mảnh dữ liệu (data fragmentation) là một kỹ thuật được sử dụng rộng rãi để tối ưu hiệu suất của hệ thống cơ sở dữ liệu, đặc biệt trong các ứng dụng đòi hỏi xử lý lượng dữ liệu khổng lồ như hệ thống đánh giá phim trực tuyến. Bài tập lớn này tập trung vào việc mô phỏng các phương pháp phân mảnh ngang (horizontal fragmentation) trên hệ quản trị cơ sở dữ liệu PostgreSQL, sử dụng tập dữ liệu từ MovieLens với hơn 10 triệu đánh giá phim.

Mục tiêu của bài báo cáo là trình bày quá trình triển khai các hàm Python để tải dữ liệu, phân mảnh bảng Ratings theo hai phương pháp: **phân mảnh theo khoảng giá trị (Range Partitioning)** và **phân mảnh luân phiên (Round Robin Partitioning)**, đồng thời đảm bảo việc chèn dữ liệu mới vào đúng phân mảnh tương ứng. Báo cáo cũng sẽ phân tích cách tiếp cận, giải thích thuật toán, và đánh giá kết quả thực hiện, từ đó làm rõ ưu điểm và hạn chế của từng phương pháp trong bối cảnh cụ thể.

2. Cài đặt môi trường

Sau khi cài đặt python và PostgreSQL theo yêu cầu của bài toán, việc tiếp theo bạn phải làm là cài đặt các thư viện cần thiết để có thể kết nối đến database.

Chúng ta sẽ cài đặt thư viện **psycopg2** để làm việc với PostgreSQL trong Python. Bạn có thể cài đặt nó bằng lệnh sau:

```
pip install psycopg2-binary
```

I. Load_Ratings()

1. Yêu cầu

- Cài đặt hàm Python Load_Ratings() nhận vào một đường dẫn tuyệt đối đến tệp ratings.dat. Load_Ratings() sẽ tải nội dung tệp vào một bảng trong PostgreSQL có tên Ratings với schema sau:

- o UserID (int)
- o MovieID (int)
- o Rating (float)

2. Bài làm

+ Đề tải nội dung tệp ratings.dat vào PostgreSQL:

- Tạo hàm **getopenconnection()** để kết nối vào user;

```
import psycopg2

DATABASE_NAME = 'dds_assgn1'
ratingtablename = 'ratings'
INPUT_FILE_PATH = 'ratings.dat'

# Kết nối đến cơ sở dữ liệu PostgreSQL
def getopenconnection(user='postgres',password='1234',dbname='dds_assgn1'):
    return psycopg2.connect("dbname='"+dbname+"' user='"+user+"' host='localhost' password='"+password+"'")
```

+ Tạo hàm **loadratings('ratings', ratings.dat, openconnection):**

Khai báo một con trỏ cơ sở dữ liệu current, dùng để làm việc với CSDL bằng cách đẩy các câu lệnh truy vấn SQL lên CSDL.

- Sử dụng **openconnection** (đối tượng kết nối PostgreSQL đã được mở sẵn)

```
def loadratings(ratingtablename,ratingsfilepath,openconnection):
    current=openconnection.cursor()
```

Tạo bảng ratings

- Sử dụng lệnh **execute()** để thực thi các câu lệnh SQL:

- Xóa bảng 'ratings' nếu CSDL đã tồn tại bảng cùng tên:

```
# Xóa bảng ratings nếu đã tồn tại
droptable="drop table if exists "+ratingtablename + ";"
current.execute(droptable)
```

- Tạo bảng 'ratings' với schemas:

```
# Tạo bảng ratings
createtable="create table if not exists "+ratingtablename+" (userid INT, movieid INT, rating float);"
current.execute(createtable)
```

Đọc dữ liệu từ file và chèn vào bảng ratings

- Do 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**

=> Ta cần tách các dữ liệu của mỗi dòng, rồi lần lượt chèn dữ liệu vào từng cột tương ứng trong CSDL.

- File ratings.dat có dữ liệu rất lớn, nên nếu sử dụng lệnh **INSERT** cho từng dòng vào CSDL thì thời gian thực thi sẽ vô cùng lớn.

=> Ta sẽ sử dụng lệnh **COPY** trong PostgreSQL với file .csv để lưu toàn bộ dữ liệu của file ratings.csv (tốc độ rất nhanh), sau khi chuyển đổi dữ liệu của file ratings.dat sang file ratings.csv.

- Để tránh phải tạo file ratings.csv, ta sử dụng bộ đệm StringIO để chuyển đổi dữ liệu của file ratings.dat sang bộ nhớ, rồi COPY trực tiếp trong bộ nhớ.

- Ở đầu file, import thư viện io: `import io`

- Tạo bộ nhớ tạm để lưu dữ liệu từ file: `buffer = io.StringIO()`

- Để tránh việc chuyển đổi quá nhiều dòng không cần thiết (thể hiện qua

ACTUAL_ROWS_IN_INPUT_FILE của Assignment1Tester.py), ta đặt 2

biến để kiểm tra số dòng được ghi trong bộ nhớ:

```
max_rows = 20 # =ACTUAL_ROWS_IN_INPUT_FILE
current_rows = 0
```

- Đọc dữ liệu từ file và ghi vào bộ nhớ:

```
with open(ratingsfilepath, 'r', encoding='utf-8') as infile:
    for line in infile:
        if current_rows >= max_rows: # Dừng khi đã đọc đủ số dòng
            break
        parts = line.strip().split('::')
        if len(parts) == 4:
            buffer.write(f"{parts[0]},{parts[1]},{parts[2]}\n")
            current_rows += 1
```

- Quay lại đầu bộ nhớ để **COPY** đọc: `buffer.seek(0)`

- Dùng lệnh **COPY** để tải dữ liệu vào bảng:

```
current.copy_expert(f"""
    copy {ratingtablename} (userid, movieid, rating)
    from stdin with (format csv)
""", buffer)
```

- (`copy_expert()` là hàm chuyên dùng để thực thi câu lệnh COPY thủ công, cho phép linh hoạt định nghĩa câu lệnh COPY.)

Lưu lại các thay đổi trên CSDL:

```
# Lưu các thay đổi vào cơ sở dữ liệu
openconnection.commit()
current.close()
```










- Test hàm:

```
if __name__ == "__main__":
    # Khởi tạo kết nối đến cơ sở dữ liệu
    con = getopenconnection()
    # Tạo cơ sở dữ liệu
    loadratings(ratingtablename, INPUT_FILE_PATH, con)
    # Đóng kết nối
    con.close()
```

3. Kết quả

+ Chạy hàm **loadratings()** (ACTUAL_ROWS_IN_INPUT_FILE = 10000054)

thành công, CSDL sẽ cho bảng 'ratings' với dữ liệu của các cột tương ứng:

Query		Query History						
1 select * from public.ratings								
Data Output		Messages		Notifications				
								
	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					
13	1	420	5					
14	1	466	5					
15	1	480	5					
16	1	520	5					
17	1	539	5					
18	1	586	5					
19	1	588	5					
20	1	589	5					
21	1	594	5					
22	1	616	5					
23	2	110	5					
24	2	151	3					
25	2	260	5					
26	2	376	3					
27	2	539	3					
Total rows: 10000054				Query complete 00:00:02.614				

II. Range_Partition()

1. Yêu cầu

- Cài đặt hàm Python Range_Partition() nhận vào: (1) bảng Ratings trong PostgreSQL (hoặc MySQL) và (2) một số nguyên N là số phân mảnh cần tạo. Range_Partition() sẽ tạo N phân mảnh ngang của bảng Ratings và lưu vào PostgreSQL.

- Thuật toán sẽ phân chia dựa trên N khoảng giá trị đồng đều của thuộc tính Rating.

2. Bài làm

Bước 1: Phân tích cơ sở lý luận và ý tưởng của bài toán

- Với yêu cầu bài toán, hướng giải quyết và cơ sở logic sẽ là ta sẽ lấy giá trị max của **rating** trừ cho giá trị min của **rating** để tìm ra khoảng giá trị để phân chia, mà min **rating** là 0 nên ta chỉ cần lấy max của rating chia cho số phân mảnh được truyền vào là sẽ ra từng khoảng giá trị của từng phân mảnh. Sau đó ta sẽ tính toán giá trị bắt đầu và kết thúc của từng phân mảnh.

VD: **range_size** = $(5 - 0) / n$ (với n là số phân mảnh, còn **range_size** là từng khoảng giá trị đồng đều của từng phân mảnh)

- Vị trí bắt đầu và kết thúc có thể được tính bằng cách lấy số thứ tự của phân mảnh nhân với khoảng giá trị đồng đều giữa các phân mảnh là ra biên dưới, còn biên trên ta chỉ cần lấy thứ tự của phân mảnh tăng lên 1 đơn vị. Với phân mảnh đầu tiên ta lấy cả biên dưới và biên trên, còn với các phân mảnh còn lại thì chỉ lấy các giá trị lớn hơn biên dưới và bằng biên trên

VD: Trường hợp N = 2 phân mảnh đầu tiên chứa các giá trị [0, 2.5], phân mảnh thứ 2 chứa các chứa các giá trị (2.5, 5]

- Tiếp theo ta sẽ sử dụng các câu lệnh truy vấn để khởi tạo các bảng phân mảnh, truy xuất dữ liệu từ bảng **ratings** và truyền dữ liệu thỏa mãn điều kiện vào các phân mảnh.
- Để thực hiện các câu lệnh truy vấn ta sẽ sử dụng SQL Composition, đây là một kỹ thuật xây dựng câu lệnh SQL động (dynamic SQL) bằng cách sử dụng các lớp trong module `psycopg2.sql`, ví dụ như **Identifier**, **SQL**, **Literal** . Điều này giúp **tránh lỗi cú pháp và ngăn ngừa SQL Injection**.

Bước 2: Thực hiện thuật toán

- Đầu tiên là kiểm tra xem đầu vào tham số n phân mảnh có hợp lệ hay không, và nếu $n \leq 0$ thì sẽ dừng lại ngay

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
38
39     print(f"Bắt đầu rangepartition bảng '{ratingtablename}' thành {numberofpartitions} phân mảnh.")
40
41     # Kiểm tra dữ liệu đầu vào
42     if not isinstance(numberofpartitions, int) or numberofpartitions <= 0:
43         print("Số lượng phân mảnh phải là một số nguyên dương.")
44         return
45

```

- Tiếp theo ta sẽ phải khai báo một con trỏ cơ sở dữ liệu **current**.
- Con trỏ này sẽ giúp ta làm việc với cơ sở dữ liệu bằng cách đẩy các câu lệnh truy vấn SQL của ta lên cơ sở dữ liệu.

```

45
46     current = openconnection.cursor()

```

- Trong đó **openconnection** là một đối tượng kết nối tới cơ sở dữ liệu, nó sẽ giữ **kết nối mở** giữa chương trình Python và hệ quản trị cơ sở dữ liệu (DBMS), ví dụ ở đây PostgreSQL.
- Ta sẽ khai báo schema cho các bảng phân mảnh **partition_table_schema_sql**, **select_columns_sql** để ứng dụng truy vấn phía sau

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
48     try:
49         # Schema của bảng phân mảnh
50         partition_table_schema_sql = "UserID INT, MovieID INT, Rating FLOAT"
51         select_columns_sql = "UserID, MovieID, Rating"
52

```

- Vì đề bài yêu cầu phải chia ra các phân mảnh thành các khoảng đồng đều của thuộc tính **rating**, vì vậy ta có thể tính toán các khoảng đồng đều **range_size** bằng cách lấy max của giá trị **rating** (Do min của **rating** là 0 nên ta chỉ cần lấy max của **rating**) chia cho số phân mảnh cần phân:

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
48     try:
49         # Schema của bảng phân mảnh
50         partition_table_schema_sql = "UserID INT, MovieID INT, Rating FLOAT"
51         select_columns_sql = "UserID, MovieID, Rating"
52
53         # Tính toán kích thước của mỗi phân vùng
54         range_size = 5.0 / numberofpartitions
55

```

- Trong đó **range_size** là khoảng đồng đều giữa các phân mảnh, **numberofpartitions** là số phân mảnh.
- Tiếp theo ta sẽ chạy một vòng lặp chạy đúng n lần với n là số phân mảnh cần tạo, với mỗi lần lặp thì ta sẽ tạo ra 1 phân mảnh và chèn các giá trị thích hợp vào phân mảnh đó

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
56     # Tạo các bảng phân mảnh
57     for i in range(numberofpartitions):

```

- Để thực hiện truy vấn đầu tiên ta sẽ tạo tên của các bảng cách lấy tiền tố được quy định trong đề bài là **range_part** sau đó ghép với số thứ tự thứ i của vòng lặp. Khi đã tạo xong tên ta sẽ tạo **Identifier** cho tên bảng gốc, cũng như tên bảng vừa tạo. Trong đó **Identifier** là một lớp (class) trong module **psycpg2.sql**, dùng để biểu diễn tên đối tượng trong cơ sở dữ liệu như tên bảng, tên cột.

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
56     # Tạo các bảng phân mảnh
57     for i in range(numberofpartitions):
58         # Tạo tên bảng phân mảnh
59         RANGE_TABLE_PREFIX = 'range_part'
60         partition_table_name = f"{RANGE_TABLE_PREFIX}{i}"
61         partition_table_identifier = psycpg2.sql.Identifier(partition_table_name)
62         ratings_table_identifier = psycpg2.sql.Identifier(ratingtablename)
63

```

- Có thể trong cơ sở dữ liệu còn nhiều phân mảnh hay bảng trùng tên, để tránh ảnh hưởng thì tốt nhất ta nên xóa nó đi, bằng cách sử dụng câu lệnh SQL DROP và gửi lên cơ sở dữ liệu thông qua lệnh `execute()` kết hợp với `Identifier` đã khai báo từ trước.

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
38
39     # Xóa bảng nếu đã tồn tại
64     current.execute(psycopg2.sql.SQL("DROP TABLE IF EXISTS {};").format(
65         partition_table_identifier
66     ))
67
68

```

- Sau khi dọn dẹp xong các bảng trùng tên, ở mỗi vòng lặp sẽ tính toán khoảng giá trị bắt đầu **start_value** và kết thúc **end_value** của mỗi phân mảnh. Giá trị bắt đầu sẽ bằng số thứ tự của phân mảnh đó nhân với kích thước của từng phân mảnh, còn giá trị kết thúc sẽ tương tự nhưng tăng số thứ tự lên một đơn vị.

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
38
39     # Tính toán giá trị bắt đầu và kết thúc cho phân mảnh
69     start_value = i * range_size
70     end_value = (i + 1) * range_size
71
72

```

- Khi đã có giá trị bắt đầu và kết thúc, ta cần tạo bảng phân mảnh mới trên database bằng câu lệnh SQL CREATE. Ta sẽ sử dụng lệnh execute() để đẩy các câu lệnh SQL và sử dụng Identifier tên mảnh phía trên làm tên phân mảnh, còn schema sẽ quy sang câu lệnh SQL bằng **psycopg2.sql.SQL()**

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
38
39     # Tạo bảng phân mảnh
73     current.execute(psycopg2.sql.SQL("CREATE TABLE {} ({});").format(
74         partition_table_identifier,
75         psycopg2.sql.SQL(partition_table_schema_sql)
76     ))
77
78

```

- Sau khi tạo bảng phân mảnh xong thì sẽ thêm dữ liệu phù hợp vào bảng. Vì đã biết giá trị đầu tiên và kết thúc của mỗi phân mảnh, ta thực hiện các câu lệnh truy vấn phù hợp để lấy dữ liệu từ bảng chính sang các phân mảnh.
- Với điều kiện là nếu là bảng đầu tiên thì sẽ lấy cả giá trị bắt đầu đến giá trị kết thúc, còn ở những bảng sau chỉ lấy các giá trị lớn hơn giá trị bắt đầu đến giá trị kết thúc đã được tính ở bước trên, và cũng giống như ví dụ trong đề bài

```

37 def rangepartition(ratingtablename, numberofpartitions, openconnection):
79     # Chèn dữ liệu vào bảng phân vùng
80     if i == 0:
81         # Bao gồm cả giá trị = start_value
82         current.execute(psycopg2.sql.SQL("""
83             INSERT INTO {} ({}
84             SELECT {} FROM {}
85             WHERE rating >= %s AND rating <= %s;
86         """).format(
87             partition_table_identifier,
88             psycopg2.sql.SQL(select_columns_sql),
89             psycopg2.sql.SQL(select_columns_sql),
90             ratings_table_identifier
91         ), (start_value, end_value))
92     else:
93         current.execute(psycopg2.sql.SQL("""
94             INSERT INTO {} ({}
95             SELECT {} FROM {}
96             WHERE rating > %s AND rating <= %s;
97         """).format(
98             partition_table_identifier,
99             psycopg2.sql.SQL(select_columns_sql),
100             psycopg2.sql.SQL(select_columns_sql),
101             ratings_table_identifier
102         ), (start_value, end_value))
103

```

- Ở trên ta sử dụng câu lệnh INSERT kết hợp với cách thực thi truy vấn theo kiểu **SQL Composition** và **Parameterized Queries**. Trong đó **SQL Composition** sẽ giúp truy vấn được mạch lạc rõ ràng cũng như chống tấn công **SQL Injection** thông qua phương thức `.format()`, còn **Parameterized Queries** sẽ sử dụng các placeholder `%s` và `psycopg2` sẽ tự động escape giá trị ở đây là các giá trị `start_value` và `end_value`.
- Cuối cùng là sẽ xác nhận và lưu lại các thay đổi trên cơ sở dữ liệu

```

104     openconnection.commit()
105     print("Phân mảnh thành công!")

```

3. Kết quả và kiểm thử

Để kiểm thử file và xem kết quả khi thử chạy hàm, ta sẽ thử gọi hàm và truyền vào các tham số cần thiết và xem kết quả trên cơ sở dữ liệu.

Giải sử phân thành 3 mảnh

```
211 if __name__ == "__main__":
212     try:
213         conn = getopenconnection()
214         loadratings(ratingstablename, INPUT_FILE_PATH, conn)
215         rangepartition(ratingstablename, 3, conn)
216         rangeinsert(ratingstablename, 1, 100, 2.5, conn)
217     except Exception as e:
218         print(f"Đã xảy ra lỗi: {e}")
219     finally:
220         if conn:
221             conn.close()
```









Trước khi gọi hàm thì trong cơ sở dữ liệu hiện có 1 bảng chính là **ratings** chứa 10 triệu bản ghi ban đầu

✓  Tables (1)
| >  ratings

Query Query History

```
1 SELECT COUNT(*) FROM public.ratings
2
```

Data Output Messages Notifications

									SQL
	count								
	bigint								
1	10000054								

Sau khi chạy hàm thì cơ sở dữ liệu sẽ có thêm 3 phân mảnh được tạo ra

▼	Tables (4)
>	range_part0
>	range_part1
>	range_part2
>	ratings

Với dữ liệu trong phân mảnh lần lượt theo từng khoảng giá trị rating $[0, 1.67]$, $(1.67, 3.34]$, $(3.34, 5]$ là

Query

Query History

1

2

SELECT * FROM public.range_part0

Data Output

Messages

Notifications

	userid integer	movieid integer	rating double precision
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	1895	1.5
10	7	1917	1
11	7	2478	1
12	7	5094	1

Total rows: 597446

Query complete 00:00:00.449

Query

Query History

1

2

SELECT * FROM public.range_part1

Data Output

Messages

Notifications

userid

integer

movieid

integer

rating

double precision

1

2

151

3

2

2

376

3

3

2

539

3

4

2

648

2

5

2

719

3

6

2

733

3

7

2

736

3

8

2

780

3

9

2

786

3

10

2

802

2

11

2

858

2

12

2

1049

3

Total rows: 3517160

Query complete 00:00:01.383

Query

Query History

1

SELECT * FROM public.range_part2

2

Data Output

Messages

Notifications

</

Để kiểm tra kỹ hơn ta thực hiện 1 số truy vấn để xác định có bộ dữ liệu nào nằm ở nhằm phân mảnh không

Truy vấn với phân mảnh **range_part0**

Query Query History

```
1 SELECT COUNT(*) FROM public.range_part0
2 WHERE rating > 1.67
3
```

Data Output Messages Notifications

	count bigint
1	0

Truy vấn với phân mảnh **range_part1**

Query Query History

```
1 SELECT COUNT(*) FROM public.range_part1
2 WHERE rating <= 1.67 AND rating > 3.34
3
```

Data Output Messages Notifications

	count bigint
1	0

Truy vấn với phân mảnh **range_part2**

Query Query History

```
1 SELECT COUNT(*) FROM public.range_part2
2 WHERE rating <= 3.34
```

Data Output Messages Notifications

	count bigint
1	0

Cả 3 phân mảnh đều không có bất kỳ một bộ dữ liệu nào nằm nhầm vị trí, hoàn toàn thỏa mãn yêu cầu đề bài.

Với $n = 10$, thì chương trình mất khoảng 16s. Tốc độ truy vấn tùy vào máy tính thực hiện cũng như là có thể thay đổi theo nhiều lúc khác nhau do nhiều yếu tố cấu thành.

III. RoundRobin_Partition()

1. Yêu cầu

- Cài đặt hàm Python RoundRobin_Partition() nhận vào: (1) bảng Ratings trong PostgreSQL (hoặc MySQL) và (2) một số nguyên N là số phân mảnh cần tạo.
- Hàm sẽ tạo N phân mảnh ngang của bảng Ratings và lưu chúng trong PostgreSQL (hoặc MySQL), sử dụng phương pháp phân mảnh kiểu vòng tròn (round robin) (đã được giải thích trong lớp).

2. Bài làm

a. Mục đích

Hàm `roundrobinpartition` được thiết kế để thực hiện **phân mảnh ngang (horizontal partitioning)** cho một bảng dữ liệu đã cho (`ratingstablename`) trong cơ sở dữ liệu PostgreSQL. Phương pháp phân mảnh được sử dụng là **round-robin (vòng tròn)**, nghĩa là các hàng từ bảng gốc sẽ được phân phối lần lượt và tuần tự vào `numberofpartitions` (N) bảng con (phân mảnh) mới được tạo ra.

b. Ý tưởng thực hiện

Ý tưởng của hàm `RoundRobin_Partition()` là phân chia dữ liệu từ bảng `Ratings` thành nhiều bảng nhỏ hơn theo cách luân phiên từng dòng, nhằm đảm bảo dữ liệu được phân phối đồng đều giữa các phân mảnh. Phương pháp round-robin được sử dụng trong trường hợp này nghĩa là các dòng dữ liệu sẽ được phân phối lần lượt vào từng phân mảnh theo thứ tự: dòng đầu vào phân mảnh 0, dòng tiếp theo vào phân mảnh 1, và cứ như vậy quay vòng đến phân mảnh N-1 rồi lặp lại. Để làm được điều này, trước tiên truy xuất thông tin cấu trúc bảng gốc để tạo ra các bảng phân mảnh có schema giống hệt. Sau đó, mỗi dòng dữ liệu từ bảng gốc được đánh số thứ tự bằng hàm `ROW_NUMBER()`, rồi sử dụng phép chia dư để xác định dòng đó sẽ thuộc về phân mảnh nào.

c. Cách thức hoạt động

Hàm hoạt động theo các bước chính sau:

1. Kiểm tra tham số đầu vào:

- Đảm bảo `numberofpartitions` (số lượng phân mảnh mong muốn) là một số nguyên dương. Nếu không, hàm sẽ báo lỗi và dừng thực thi bằng cách `raise ValueError`.

```
224 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
225     print(f"Bắt đầu roundrobinpartition bảng '{ratingtablename}' thành {numberofpartitions} phân mảnh.")
226     if not isinstance(numberofpartitions, int) or numberofpartitions <= 0:
227         print("Số lượng phân mảnh phải là một số nguyên dương.")
228         raise ValueError("Số lượng phân mảnh phải là một số nguyên dương.")
```

2. Khởi tạo kết nối và con trỏ:

- Sử dụng `openconnection` (đối tượng kết nối PostgreSQL đã được mở sẵn) và tạo một đối tượng `cursor` (`cur`) để thực thi các lệnh SQL.

```
224 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
230     con = openconnection
231     cur = con.cursor()
```

3. Lấy Schema của bảng gốc:

- Thực thi câu lệnh SQL để truy vấn `information_schema.columns`. Câu lệnh này lấy ra danh sách tên cột (`column_name`) và kiểu dữ liệu (`data_type`) của tất cả các cột trong bảng `ratingtablename` (bảng gốc). Kết quả được sắp xếp theo thứ tự cột

```
224 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
236     # Lấy schema gốc của bảng gốc
237     cur.execute(
238         psycopg2.sql.SQL(
239             "SELECT column_name, data_type FROM information_schema.columns WHERE table_schema = 'public' AND table_name = %s ORDER BY ordinal_position;"
240         ), (ratingtablename,)
241     )
242     columns_schema = cur.fetchall()
```

Ý nghĩa: Để đảm bảo các bảng phân mảnh mới được tạo ra sẽ có cấu trúc (schema) giống hệt với bảng gốc. Điều này quan trọng cho tính nhất quán dữ liệu.

4. Chuẩn bị định nghĩa cột và danh sách tên cột cho SQL động:

- Từ `columns_schema` thu được, hai chuỗi SQL động được tạo ra bằng `psycpg2.sql`:
 - `column_definitions_sql`: Một chuỗi định nghĩa các cột và kiểu dữ liệu của chúng, dùng cho lệnh `CREATE TABLE` các bảng phân mảnh (ví dụ: `"UserID" INT, "MovieID" INT, "Rating" REAL`).
 - `select_column_names_sql`: Một chuỗi chứa danh sách tên các cột, dùng cho phần `SELECT` và `INSERT INTO` (ví dụ: `"UserID", "MovieID", "Rating"`).

```

224 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
248     column_definitions_sql = psycpg2.sql.SQL(", ").join(
249         [psycpg2.sql.SQL("{} {}").format(psycpg2.sql.Identifier(col_name), psycpg2.sql.SQL(col_type))
250          for col_name, col_type in columns_schema]
251     )
252     actual_column_names = [psycpg2.sql.Identifier(col_name) for col_name, _ in columns_schema]
253     select_column_names_sql = psycpg2.sql.SQL(", ").join(actual_column_names)

```

5. Tạo và điền dữ liệu vào các bảng phân mảnh (Vòng lặp):

Hàm lặp `i` từ 0 đến `numberofpartitions - 1`. Trong mỗi vòng lặp, một bảng phân mảnh được xử lý:

- **Xác định tên bảng phân mảnh:** Tên được tạo bằng cách ghép `RROBIN_TABLE_PREFIX` (đã định nghĩa là `'rrobin_part'`) với chỉ số `i` (ví dụ: `rrobin_part0`, `rrobin_part1`).
- **Xóa bảng phân mảnh cũ (nếu có):** `DROP TABLE IF EXISTS {part_table_name}` để đảm bảo hàm có thể chạy lại nhiều lần mà không gây lỗi.
- **Tạo bảng phân mảnh mới:** `CREATE TABLE {part_table_name} ({column_definitions_sql})` sử dụng schema đã lấy được từ bảng gốc.
- **Chèn dữ liệu vào bảng phân mảnh:**
 - Câu lệnh SQL `INSERT INTO ... SELECT ...` được xây dựng.
 - Phần `SELECT` sử dụng một truy vấn con (subquery): `(SELECT *, ROW_NUMBER() OVER () as rn FROM {ratingtablename}) AS temp_table_with_row_numbers`. Lệnh `ROW_NUMBER() OVER()` gán một số thứ tự duy nhất (`rn`) cho mỗi hàng trong bảng gốc.
 - Điều kiện `WHERE (temp_table_with_row_numbers.rn - 1) %% %s = %s`; là cốt lõi của logic round-robin:

- $(rn - 1)$: Vì `ROW_NUMBER()` bắt đầu từ 1, còn chỉ số phân mảnh i bắt đầu từ 0.
- `%%`: Toán tử modulo trong PostgreSQL (được escape thành `%%` trong chuỗi Python để `psycopg2` hiểu đúng là ký tự `%` của SQL, không phải placeholder).
- `%s` đầu tiên: Placeholder cho `numberofpartitions` (tổng số phân mảnh).
- `%s` thứ hai: Placeholder cho i (chỉ số của phân mảnh hiện tại).
- Ví dụ: Nếu `numberofpartitions = 4` và $i = 0$, điều kiện là $(rn - 1) \% 4 = 0$. Những hàng có rn là 1, 5, 9,... sẽ thỏa mãn và được chèn vào `rrobin_part0`.
- `cur.execute(sql_insert, (numberofpartitions, i))`: Thực thi câu lệnh `INSERT` với các tham số tương ứng.

```

224 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
259     # Tạo N bảng phân mảnh
260     for i in range(numberofpartitions):
261         part_table_name = f"{RROBIN_TABLE_PREFIX}{i}"
262         part_table_identifier = psycopg2.sql.Identifier(part_table_name)
263
264         cur.execute(psycopg2.sql.SQL("DROP TABLE IF EXISTS {};").format(part_table_identifier))
265         cur.execute(psycopg2.sql.SQL("CREATE TABLE {} ({});").format(part_table_identifier, column_definitions_sql))
266
267         sql_insert = psycopg2.sql.SQL("""
268             INSERT INTO {} ({}))
269             SELECT {}
270             FROM (
271                 SELECT *, ROW_NUMBER() OVER () as rn
272                 FROM {}
273             ) AS temp_table_with_row_numbers
274             WHERE (temp_table_with_row_numbers.rn - 1) %% %s = %s;
275         """).format(
276             part_table_identifier,
277             select_column_names_sql,
278             select_column_names_sql,
279             psycopg2.sql.Identifier(ratingtablename)
280         )
281         cur.execute(sql_insert, (numberofpartitions, i)) # Truyền numberofpartitions và i
282
283     con.commit()
284     print(f"Hàm roundrobinpartition cho bảng '{ratingtablename}' thành công.")

```

d. Ý nghĩa các tham số

- `ratingtablename` (kiểu `str`): Tên của bảng gốc chứa dữ liệu cần phân mảnh (ví dụ: `'ratings'`).
- `numberofpartitions` (kiểu `int`): Số lượng bảng phân mảnh con mong muốn tạo ra.

- `openconnection` (kiểu `psycopg2.extensions.connection`): Đối tượng kết nối đến cơ sở dữ liệu PostgreSQL đã được mở từ trước. Hàm này sẽ sử dụng kết nối này để thực hiện các thao tác.

3. Kết quả

Sau khi phân mảnh thành công, **N** bảng mới sẽ được tạo trong cơ sở dữ liệu (ví dụ: `rrobin_part0`, `rrobin_part1`, ..., `rrobin_part(N-1)`).

Mỗi bảng phân mảnh này sẽ có cấu trúc cột giống hệt bảng `ratingstablename` gốc.

Dữ liệu từ bảng `ratingstablename` sẽ được phân phối đều vào **N** bảng phân mảnh này theo kiểu vòng tròn. Tổng số hàng trong tất cả các bảng phân mảnh sẽ bằng tổng số hàng trong bảng gốc.


Giả sử phân thành 4 mảnh:


```

251  if __name__ == '__main__':
252      DB_TO_USE = DATABASE_NAME
253      USER_PG = 'postgres'
254      PASS_PG = '123456'
255
256      create_db(DB_TO_USE, user=USER_PG, password=PASS_PG)
257      conn_main = None
258      try:
259          conn_main = getopenconnection(user=USER_PG, password=PASS_PG, dbname=DB_TO_USE)
260          print(f"Đã kết nối thành công tới database '{DB_TO_USE}'.")
261
262          print("\n--- BƯỚC 1: TEST loadratings ---")
263          loadratings(RATINGS_TABLE, INPUT_FILE_PATH, conn_main)
264
265          print("\n--- BƯỚC 2: TEST roundrobinpartition ---")
266          N_round_robin = 4 # Thử với 4 phân mảnh round robin
267          roundrobinpartition(RATINGS_TABLE, N_round_robin, conn_main)
268          print(f"Đã tạo {N_round_robin} round robin partitions.")

```

Trước khi gọi hàm thì trong cơ sở dữ liệu hiện có 1 bảng chính là **ratings**

▼  Tables (1)

 >  ratings

Dữ liệu trong bảng ban đầu là có tổng cộng 10000054 dòng dữ liệu:

Query

Query History

1

SELECT * FROM public.ratings

Data Output

Messages

Notifications

Sau khi gọi hàm thì cơ sở dữ liệu có thêm 4 phân mảnh:

Tables (5)	
>	ratings
>	rrobin_part0
>	rrobin_part1
>	rrobin_part2
>	rrobin_part3

Do tách thành 4 phân mảnh nên các dữ liệu trong từng phân mảnh sẽ có số thứ tự cách nhau 4 đơn vị trong bảng dữ liệu gốc. Dữ liệu trong mỗi phân mảnh lần lượt là:

Bảng rrobin_part0 có các hàng trong bảng gốc là 1,5,9,13,17,... do $(\text{số thứ tự hàng} - 1) \% 4 = 0$

Query

Query History

1

SELECT * FROM public.rrobin_part0

Data Output

Messages

Notifications

	userid integer	movieid integer	rating double precision
1	1	122	5
2	1	316	5
3	1	362	5
4	1	420	5
5	1	539	5
6	1	594	5
7	2	260	5
8	2	648	2
9	2	780	3
10	2	1049	3
11	2	1391	3
12	3	213	5
13	3	1252	4
14	3	1552	2
15	3	3408	4
16	3	4995	4.5
17	3	5952	3.5
18	3	7153	4
19	3	8783	5
20	4	34	5

Total rows: 2500014

Query complete 00:00:00.900

Bảng rrobin_part1 có các hàng trong bảng gốc là 2,6,10,14,18,... do $(\text{số thứ tự hàng} - 1) \% 4 = 1$

Query

Query History

1

SELECT * FROM public.rrobin_part1

Data Output

Messages

Notifications

≡+

▼

▼

	userid integer	movieid integer	rating double precision
1	1	185	5
2	1	329	5
3	1	364	5
4	1	466	5
5	1	586	5
6	1	616	5
7	2	376	3
8	2	719	3
9	2	786	3
10	2	1073	3
11	2	1544	3
12	3	590	3.5
13	3	1276	3.5
14	3	1564	4.5
15	3	3684	4.5
16	3	5299	3
17	3	6287	3
18	3	7155	3.5
19	3	27821	4.5
20	4	39	3

Total rows: 2500014

Query complete 00:00:01.047

Bảng rrobin_part2 có các hàng trong bảng gốc là 3,7,11,15,19,... do $(\text{số thứ tự hàng} - 1) \% 4 = 2$

Query

Query History

1

SELECT * FROM public.rrobin_part2

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼

🗑

🗄

⬇

📈

	userid integer 🔒	movieid integer 🔒	rating double precision 🔒
1	1	231	5
2	1	355	5
3	1	370	5
4	1	480	5
5	1	588	5
6	2	110	5
7	2	539	3
8	2	733	3
9	2	802	2
10	2	1210	4
11	3	110	4.5
12	3	1148	4
13	3	1288	3
14	3	1597	4.5
15	3	4535	4
16	3	5505	2
17	3	6377	4
18	3	8529	4
19	3	33750	3.5
20	4	110	5

Total rows: 2500013

Query complete 00:00:00.946

Bảng rrobin_part3 có các hàng trong bảng gốc là 4,8,12,16,20,... do (số thứ tự hàng - 1)%4 = 3

Query

Query History

1

SELECT * FROM public.rrobin_part3

Data Output

Messages

Notifications

≡+

▼

▼

	userid integer	movieid integer	rating double precision
1	1	292	5
2	1	356	5
3	1	377	5
4	1	520	5
5	1	589	5
6	2	151	3
7	2	590	5
8	2	736	3
9	2	858	2
10	2	1356	3
11	3	151	4.5
12	3	1246	4
13	3	1408	3.5
14	3	1674	4.5
15	3	4677	4
16	3	5527	4.5
17	3	6539	5
18	3	8533	4.5
19	4	21	3
20	4	150	5

Total rows: 2500013

Query complete 00:00:00.798

IV. RoundRobin_Insert()

1. Yêu cầu

- Cài đặt hàm Python RoundRobin_Insert() nhận vào: (1) bảng Ratings trong PostgreSQL, (2) UserID, (3) ItemID, (4) Rating.
- RoundRobin_Insert() sẽ chèn một bộ mới vào bảng Ratings và vào đúng phân mảnh theo cách round robin.

2. Bài làm

a.Mục đích

Hàm **roundrobininsert** được thiết kế để chèn một bản ghi đánh giá mới (**userid**, **itemid**, **rating_value**) vào hệ thống đã được phân mảnh theo phương pháp round-robin. Bản ghi này phải được chèn vào cả **bảng ratingstablename** chính và vào **đúng bảng phân mảnh con** (**rrobin_partX**) tương ứng.

b. Ý tưởng thực hiện

Ý tưởng của hàm **RoundRobin_Insert()** là đảm bảo rằng mỗi bản ghi mới được chèn vào hệ thống phân mảnh sẽ **đồng thời được lưu** vào bảng gốc **Ratings** và vào đúng **bảng phân mảnh con** theo quy tắc **vòng tròn (round-robin)** đã thiết lập từ trước. Đầu tiên, bản ghi mới sẽ được kiểm tra tính hợp lệ (giá trị rating nằm trong khoảng cho phép) rồi chèn vào bảng **Ratings** chính. Sau đó, hệ thống sẽ lấy tổng số bản ghi hiện có để xác định **thứ tự** của bản ghi vừa thêm. Dựa vào tổng số dòng sau khi chèn và số lượng bảng phân mảnh đã có (được đếm tự động bằng cách dò theo tên), ta sẽ áp dụng công thức **(tổng dòng - 1) % số phân mảnh** để xác định **phân mảnh đích** cần chèn. Nhờ đó, hệ thống duy trì được vòng lặp đều đặn khi chèn, giúp phân phối dữ liệu một cách công bằng giữa các phân mảnh, đồng thời giữ được tính nhất quán với bảng gốc.

c.Cách thức hoạt động

1.Khởi tạo kết nối và con trỏ: Tương tự như **roundrobinpartition**.

```

321 def roundrobininsert(ratingtablename, userid, itemid, rating_value, openconnection):
322     print(f"Bắt đầu roundrobininsert: UserID={userid}, ItemID={itemid}, Rating={rating_value} vào bảng '{ratingtablename}'.")
323     con = openconnection
324     cur = con.cursor()

```

2. Chèn vào bảng chính (**ratingtablename**):

- Kiểm tra `rating_value` nếu không nằm trong khoảng từ 0 đến 5 sao thì thông báo lỗi.
- Một câu lệnh **INSERT INTO {ratingtablename} (UserID, MovieID, Rating) VALUES (%s, %s, %s)** được thực thi để chèn bản ghi mới (gồm `userid`, `itemid`, `rating_value`) vào bảng `ratingtablename` chính. Schema của bảng `ratings` được giả định là 3 cột này theo kết quả của hàm `loadratings`.

```

321 def roundrobininsert(ratingtablename, userid, itemid, rating_value, openconnection):
326     if rating_value < 0.0 or rating_value > 5.0 :
327         raise ValueError(f"Rating không hợp lệ. Phải nằm trong khoảng từ 0 đến 5.")
328
329     # Chèn vào bảng Ratings chính (schema: UserID, MovieID, Rating)
330     sql_insert_main = psycopg2.sql.SQL("INSERT INTO {} (UserID, MovieID, Rating) VALUES (%s, %s, %s);").format(
331         psycopg2.sql.Identifier(ratingtablename)
332     )
333     cur.execute(sql_insert_main, (userid, itemid, rating_value))

```

- **Ý nghĩa:** Đảm bảo bảng chính luôn chứa tất cả dữ liệu, bao gồm cả dữ liệu mới nhất.

3. Xác định phân mảnh đích:

- **Lấy tổng số hàng sau khi chèn:** Thực thi **SELECT COUNT(*) FROM {ratingtablename}** để lấy `total_rows_after_insert`. Đây là tổng số hàng trong bảng chính sau khi bản ghi mới vừa được thêm vào.
- **Đếm số lượng phân mảnh:** Gọi hàm `count_partitions(RROBIN_TABLE_PREFIX, con)` để xác định có bao nhiêu bảng phân mảnh `rrobin_partX` đang tồn tại. Đây là hàm tìm số lượng phân mảnh dựa theo tiền tố của bảng phân mảnh. **RROBIN_TABLE_PREFIX** là `'rrobin_part'`.

```

297 def count_partitions(prefix, openconnection):
298     count = 0
299     temp_cur = None
300     try:
301         temp_cur = openconnection.cursor()
302         query = "SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = 'public' AND table_name LIKE %s;"
303         temp_cur.execute(query, (prefix + '%',))
304         result = temp_cur.fetchone()
305         if result:
306             count = result[0]
307     except psycopg2.Error as e:
308         print(f"Lỗi khi đếm partitions với prefix '{prefix}': {e}")
309     finally:
310         if temp_cur:
311             temp_cur.close()
312     return count
313

```

fetchone() trả về một tuple chứa một dòng kết quả, hoặc None nếu không có dòng nào.

```
result = temp_cur.fetchone()
```

Giá trị COUNT(*) nằm ở phần tử đầu tiên (chỉ số 0) của tuple kết quả.

```
count = result[0]
```

- **Kiểm tra điều kiện:** Nếu `total_rows_after_insert` là 0 (bất thường) hoặc `numberofpartitions` là 0 (chưa có phân mảnh nào được tạo), hàm sẽ báo lỗi, rollback và dừng.
- **Tính chỉ số phân mảnh đích:** `target_partition_index = (total_rows_after_insert - 1) % numberofpartitions.`
 - Bản ghi vừa chèn là bản ghi thứ `total_rows_after_insert`.
 - Sử dụng `(tổng số hàng - 1)` vì hàng được đánh số từ 1 nhưng chỉ số phân mảnh từ 0.
- **Xác định tên bảng phân mảnh đích:** `target_partition_name = f"{RROBIN_TABLE_PREFIX}{target_partition_index}"`.

```

321 def roundrobininsert(ratingtablename, userid, itemid, rating_value, openconnection):
322     # Xác định phân mảnh đích
323     cur.execute(psycopg2.sql.SQL("SELECT COUNT(*) FROM {};").format(psycopg2.sql.Identifier(ratingtablename)))
324     total_rows_after_insert = 0
325     result = cur.fetchone()
326     if result:
327         total_rows_after_insert = result[0]
328
329     if total_rows_after_insert == 0:
330         print(f"LỖI: Không thể lấy total_rows từ '{ratingtablename}' sau khi chèn.")
331         con.rollback()
332         raise Exception("Không thể lấy total_rows.")
333
334     numberofpartitions = count_partitions(RROBIN_TABLE_PREFIX, con)
335
336     if numberofpartitions == 0:
337         print(f"LỖI: Không tìm thấy phân mảnh round robin nào. Hãy tạo phân mảnh trước.")
338         con.rollback()
339         raise Exception(f"Không có phân mảnh round robin (prefix: {RROBIN_TABLE_PREFIX}).")
340
341     target_partition_index = (total_rows_after_insert - 1) % numberofpartitions
342     target_partition_name = f"{RROBIN_TABLE_PREFIX}{target_partition_index}"

```

4.Chèn vào bảng phân mảnh đích:

- Một câu lệnh **INSERT INTO {target_partition_name} (UserID, MovieID, Rating) VALUES (%s, %s, %s)** được thực thi để chèn cùng bản ghi đó vào bảng phân mảnh đã xác định.

```

321 def roundrobininsert(ratingtablename, userid, itemid, rating_value, openconnection):
322     # Chèn vào bảng phân mảnh đích (schema: UserID, MovieID, Rating)
323     sql_insert_partition = psycopg2.sql.SQL("INSERT INTO {} (UserID, MovieID, Rating) VALUES (%s, %s, %s);").format(
324         psycopg2.sql.Identifier(target_partition_name)
325     )
326     cur.execute(sql_insert_partition, (userid, itemid, rating_value))
327     # print(f" Đã chèn vào bảng phân mảnh '{target_partition_name}'.")
328
329     con.commit()
330     print(f"roundrobininsert hoàn thành.")

```

d.Ý nghĩa các tham số

- **ratingtablename** (kiểu **str**): Tên của bảng gốc (ví dụ: 'ratings').
- **userid** (kiểu **int**): UserID của bản ghi mới.
- **itemid** (kiểu **int**): MovieID (ItemID) của bản ghi mới.
- **rating_value** (kiểu **float**): Giá trị Rating của bản ghi mới.
- **openconnection** (kiểu **psycopg2.extensions.connection**): Đối tượng kết nối PostgreSQL đã mở.

3. Kết quả

- Nếu thành công, một bản ghi mới sẽ được thêm vào cuối bảng `ratingstablename`.
- Đồng thời, bản ghi mới đó cũng sẽ được thêm vào một trong các bảng phân mảnh `rrobin_partX` dựa trên logic round robin (vị trí của nó sau khi được thêm vào bảng chính).
- Tính nhất quán dữ liệu được duy trì giữa bảng chính và các phân mảnh.

Giả sử thêm 5 dữ liệu như hình dưới vào các bảng phân mảnh sau khi chạy hàm `roundrobinpartition()` với 4 phân mảnh.

```
251 if __name__ == '__main__':
252     DB_TO_USE = DATABASE_NAME
253     USER_PG = 'postgres'
254     PASS_PG = '123456'
255
256     create_db(DB_TO_USE, user=USER_PG, password=PASS_PG)
257     conn_main = None
258     try:
259         conn_main = getopenconnection(user=USER_PG, password=PASS_PG, dbname=DB_TO_USE)
260         print(f"Đã kết nối thành công tới database '{DB_TO_USE}'.")
261
262         print("\n--- BƯỚC 1: TEST loadratings ---")
263         loadratings(RATINGS_TABLE, INPUT_FILE_PATH, conn_main)
264
265         print("\n--- BƯỚC 2: TEST roundrobinpartition ---")
266         N_round_robin = 4 # Thử với 4 phân mảnh round robin
267         roundrobinpartition(RATINGS_TABLE, N_round_robin, conn_main)
268         print(f"Đã tạo {N_round_robin} round robin partitions.")
269
270         print("\n--- BƯỚC 3: BẮT ĐẦU TEST roundrobininsert ---")
271         print("Chèn dữ liệu mẫu:")
272         roundrobininsert(RATINGS_TABLE, 88801, 2001, 3.0, conn_main)
273         roundrobininsert(RATINGS_TABLE, 88802, 2002, 4.0, conn_main)
274         roundrobininsert(RATINGS_TABLE, 88803, 2003, 5.0, conn_main)
275         roundrobininsert(RATINGS_TABLE, 88804, 2004, 2.5, conn_main)
276         roundrobininsert(RATINGS_TABLE, 88805, 2005, 1.0, conn_main)
```

Dữ liệu bảng `ratings` ban đầu như hình dưới và các bảng `rrobin_part0`, `rrobin_part1`, `rrobin_part2`, `rrobin_part3` như trên đã phân mảnh.

Sau khi thêm 5 dữ liệu vào các bảng phân mảnh thì dữ liệu sau khi phân mảnh:

Dữ liệu bảng ratings lúc này đã tăng thêm 5 dòng là 10000055, 10000056, 10000057, 10000058, 10000059.

Query

Query History

1

SELECT * FROM public.ratings

Data Output

Messages

Notifications

	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
13	1	420	5
14	1	466	5
15	1	480	5
16	1	520	5
17	1	539	5
18	1	586	5
19	1	588	5
20	1	589	5

Total rows: 10000059

Query complete 00:00:02.723

Dữ liệu các bảng phân mảnh:

- Bảng rrobin_part0 sẽ có thêm 1 dòng là 10000057 từ bảng gốc tương ứng với $userid = 88803$ và $movieid = 2003$ do $(10000057-1)\% 4 = 0$ nên sẽ được thêm vào bảng rrobin_part0.

Như hình bên dưới thì lúc này tổng số dòng đã tăng từ 2500014 lên 2500015

Query Query History

1 **SELECT** * **FROM** public.rrobin_part0

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	1	122	5
2	1	316	5
3	1	362	5
4	1	420	5
5	1	539	5
6	1	594	5
7	2	260	5
8	2	648	2
9	2	780	3
10	2	1049	3
11	2	1391	3
12	3	213	5
13	3	1252	4
14	3	1552	2
15	3	3408	4
16	3	4995	4.5
17	3	5952	3.5
18	3	7153	4
19	3	8783	5
20	4	34	5

Total rows: 2500015 Query complete 00:00:00.757

Query Query History

1 **SELECT** * **FROM** public.rrobin_part0

2 **WHERE** userid = 88803 and movieid = 2003

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	88803	2003	5

Như hình bên dưới thì lúc này tổng số dòng đã tăng từ 2500014 lên 2500015

The screenshot shows the PostgreSQL query editor interface. The 'Query' tab is active, displaying the following SQL query:

```
1 SELECT * FROM public.rrobin_part1
2 WHERE userid = 88804 and movieid = 2004
```

The 'Data Output' tab is also visible, showing the result of the query. The result is a single row with the following columns and values:

	userid integer	movieid integer	rating double precision
1	88804	2004	2.5

- Bảng rrobin_part2 sẽ có thêm 2 dòng là 10000055 và 10000059 từ bảng gốc tương ứng với userid = 88801, movieid = 2001 và userid = 88805, movieid = 2005 do $(10000055-1) \% 4 = 2$ và $(10000059-1) \% 4 = 2$ nên sẽ được thêm vào bảng rrobin_part2.

Như hình bên dưới thì lúc này tổng số dòng đã tăng từ 2500013 lên 2500015

Query

Query History

1

SELECT * FROM public.rrobin_part2

2

Data Output

Messages

Notifications

≡+

▼

▼

	userid integer	movieid integer	rating double precision
1	1	231	5
2	1	355	5
3	1	370	5
4	1	480	5
5	1	588	5
6	2	110	5
7	2	539	3
8	2	733	3
9	2	802	2
10	2	1210	4
11	3	110	4.5
12	3	1148	4
13	3	1288	3
14	3	1597	4.5
15	3	4535	4
16	3	5505	2
17	3	6377	4
18	3	8529	4
19	3	33750	3.5

Total rows: 2500015

Query complete 00:00:00.744

Query Query History

```
1 SELECT * FROM public.rrobin_part2
2 WHERE userid = 88801 and movieid = 2001
```

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	88801	2001	3

Query Query History

```
1 SELECT * FROM public.rrobin_part2
2 WHERE userid = 88805 and movieid = 2005
```

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	88805	2005	1

- Bảng rrobin_part3 sẽ có thêm 1 dòng là 10000056 từ bảng gốc tương ứng với userid = 88802 và movieid = 2002 do $(10000056-1)\% 4 = 3$ nên sẽ được thêm vào bảng rrobin_part3.

Như hình bên dưới thì lúc này tổng số dòng đã tăng từ 2500013 lên 2500014

Query Query History

```
1 SELECT * FROM public.rrobin_part3
2
```

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	1	292	5
2	1	356	5
3	1	377	5
4	1	520	5
5	1	589	5
6	2	151	3
7	2	590	5
8	2	736	3
9	2	858	2
10	2	1356	3
11	3	151	4.5
12	3	1246	4
13	3	1408	3.5
14	3	1674	4.5
15	3	4677	4
16	3	5527	4.5
17	3	6539	5
18	3	8533	4.5
19	4	21	3

Total rows: 2500014 Query complete 00:00:00.944

Query Query History

```
1 SELECT * FROM public.rrobin_part3
2 WHERE userid = 88802 and movieid = 2002
```

Data Output Messages Notifications

	userid integer	movieid integer	rating double precision
1	88802	2002	4

V. Range_Insert()

1. Yêu cầu

- Cài đặt hàm Python Range_Insert() nhận vào: (1) bảng Ratings trong PostgreSQL (hoặc MySQL), (2) UserID, (3) ItemID, (4) Rating.
- Range_Insert() sẽ chèn một bộ mới vào bảng Ratings và vào đúng phân mảnh dựa trên giá trị của Rating.

2. Bài làm

Bước 1: Phân tích cơ sở lý luận và ý tưởng của bài toán

- Với yêu cầu của bài toán để viết được hàm thỏa mãn yêu cầu ta cần xác định logic xử lý. Ở đây ta sẽ xử lý bắt đầu từ dữ liệu đầu vào đảm bảo nó chính xác tránh trường hợp sai sót.
- Tiếp theo ta sẽ chèn bộ dữ liệu mới vào bảng chính, tìm số lượng phân mảnh hiện có trên cơ sở dữ liệu, từ số phân mảnh ta làm tương như lúc chia, ta sẽ tìm khoảng giá trị của từng phân mảnh xong xét giá trị của **rating** phù hợp vào phân mảnh thứ bao nhiêu như là cách thuật toán phân mảnh hoạt động.
- Việc tìm số phân mảnh hiện có trên cơ sở dữ liệu sẽ dựa vào dữ liệu có trong đề bài, được biết phần bắt đầu cho mỗi phân mảnh sẽ là **range_part** kết hợp với số thứ tự phân mảnh bắt đầu từ 0, vì vậy ta chỉ cần đếm xem có bao nhiêu thỏa mãn yêu cầu này thì sẽ tìm được hiện tại đang bảng **ratings** gốc đang chia thành bao nhiêu phân mảnh.
- Sau khi xác định được giá trị **rating** thuộc phân mảnh thứ bao nhiêu, ta sẽ xác định tên mảnh phù hợp và sử dụng câu lệnh truy vấn để chèn bộ dữ liệu.

Bước 2: Thực hiện thuật toán

- Mặc định tên bảng gốc đầu vào là chính xác

- Đầu tiên ta sẽ kiểm tra dữ liệu đầu vào xem **rating** có chính xác hay không, để xét các điều kiện tìm phân mảnh thích hợp theo giá trị **rating** nhập vào

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
120
121     print(f"Bắt đầu chèn bộ dữ liệu vào bảng '{ratingtablename}' với UserID={userid}, MovieID={itemid}, Rating={rating}.")
122
123     validaterating = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]
124     # Kiểm tra dữ liệu đầu vào
125     if rating not in validaterating:
126         print("Rating không hợp lệ. Nó phải nằm trong các giá trị từ 0.0 đến 5.0, cách nhau 0.5.")
127         return
128

```

- Tiếp theo ta sẽ phải khai báo một con trỏ cơ sở dữ liệu **current**.
- Con trỏ này sẽ giúp ta làm việc với cơ sở dữ liệu bằng cách đẩy các câu lệnh truy vấn SQL của ta lên cơ sở dữ liệu.

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
129     current = openconnection.cursor()
130

```

- Trong đó **openconnection** là một đối tượng kết nối tới cơ sở dữ liệu, nó sẽ giữ **kết nối mở** giữa chương trình Python và hệ quản trị cơ sở dữ liệu (DBMS), ví dụ ở đây PostgreSQL.
- Ta sẽ khai báo các schema theo đề bài trước để về sau sử dụng

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
130
131     try:
132         # Schema của bảng phân mảnh
133         rating_table_identifier = psycopg2.sql.Identifier(ratingtablename)
134         select_columns_sql = psycopg2.sql.SQL("userid, movieid, rating")
135

```

- Theo đề bài thì đầu tiên ta sẽ chèn 1 bộ mới vào bảng ratings, dựa trên con trỏ đã khai báo, sử dụng lệnh execute() để đẩy truy vấn lên cơ sở dữ liệu, ở đây ta dùng câu lệnh truy vấn INSERT để thêm bộ dữ liệu là các tham số đầu vào ban đầu vào bảng **ratings**.

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
131     try:
132         # Schema của bảng phân mảnh
133         rating_table_identifier = psycopg2.sql.Identifier(ratingtablename)
134         select_columns_sql = psycopg2.sql.SQL("userid, movieid, rating")
135
136         # Dùng lệnh INSERT để thêm dữ liệu vào bảng ratings
137         current.execute(
138             psycopg2.sql.SQL("INSERT INTO {} ({} ) VALUES (%s, %s, %s)").format(
139                 rating_table_identifier,
140                 select_columns_sql
141             ),
142             (userid, itemid, rating)
143         )
144

```

- Trong câu lệnh trên sử dụng lệnh truy vấn INSERT để thêm bộ dữ liệu mới vào bảng chính, kết hợp với Identifier và schema đã khai báo từ đầu, các placeholder %s tương ứng với các giá trị bộ dữ liệu ta truyền vào.
- Sau khi thêm bộ dữ liệu mới vào bảng chính xong, ta sẽ tính toán xem đang có bao nhiêu phân mảnh, dựa vào tên tiền tố của từng bảng là **range_part**, sử dụng câu lệnh truy vấn tìm xem có bao nhiêu bảng có tên bắt đầu bằng **range_part** , sau đó gán số phân mảnh cho biến **numberofpartitions**

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
145     # Tiền tố tên bảng phân vùng
146     RANGE_TABLE_PREFIX = 'range_part'
147
148     # Truy vấn tính số lượng phân mảnh hiện có (sử dụng SQL composition)
149     current.execute(
150         psycopg2.sql.SQL("SELECT count(*) FROM pg_stat_user_tables WHERE relname LIKE {}").format(
151             psycopg2.sql.Literal(f"{RANGE_TABLE_PREFIX}%")
152         )
153     )
154
155     # Lấy số lượng phân mảnh
156     numberofpartitions = current.fetchone()[0]
157

```

- Câu lệnh truy vấn lấy dữ liệu từ bảng **pg_stat_user_tables**, đây là một view hệ thống nằm trong module thống kê (pg_stat_*), dùng để cung cấp thông tin về các bảng do người dùng tạo ra, ở đây ta sử dụng **relname** là tên bảng để tra cứu xem có bao nhiêu bảng bắt đầu bằng tiền tố đã được quy định trong đề bài.

- Nếu số lượng phân mảnh tìm được là 0 thì trả về lỗi ngay

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
120
121     # Kiểm tra số lượng phân mảnh
122     if numberofpartitions <= 0:
123         raise ValueError("Không có phân mảnh nào được tạo. Vui lòng tạo phân mảnh trước khi chèn dữ liệu.")
124

```

- Khi đã có số phân mảnh, việc tiếp theo ta chỉ cần tính toán khoảng kích thước của mỗi phân mảnh là **range_size**, bằng cách lấy max của rating là 5 (do min của **rating** là 0) chia cho số phân mảnh tìm được.

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
120
121     # Tính khoảng cách của mỗi phân mảnh
122     range_size = 5.0 / numberofpartitions
123

```

- Sau khi có khoảng kích thước của từng phân mảnh, ta sẽ tìm xem giá trị **rating** truyền vào sẽ nằm ở phân mảnh thứ bao nhiêu, bằng cách tương tự như khi ta thêm các giá trị vào một phân mảnh. Ta sẽ lặp qua n phân mảnh, tính toán giá trị ban đầu **start** và giá trị kết thúc **end**, xét với các điều kiện là phân mảnh đầu tiên lấy cả giá trị **start**, còn các phân mảnh còn lại sẽ lấy các giá trị lớn hơn **start**, sau đó xét điều kiện xem **rating** sẽ nằm trong khoảng giá trị của phân mảnh nào và gán cho **partition_index**.

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
120
121     # Tìm phân mảnh phù hợp với rating
122     partition_index = None
123     for i in range(numberofpartitions):
124         start = i * range_size
125         end = (i + 1) * range_size
126         if i == 0:
127             if start <= rating <= end:
128                 partition_index = i
129                 break
130         else:
131             if start < rating <= end:
132                 partition_index = i
133                 break
134

```

- Nếu không tìm thấy lập tức dừng và trả về lỗi

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
179     # Nếu không tìm thấy phân mảnh phù hợp, ném ngoại lệ
180     if partition_index is None:
181         raise ValueError("Không tìm thấy phân mảnh phù hợp cho rating.")
182

```

- Và khi đã tìm được phân mảnh tương ứng chứa giá trị **rating**, ta sẽ tạo tên bảng phân mảnh tương ứng **partition_table** bằng cách lấy tiền tố của tên bảng phân mảnh kết hợp với **partition_index**. Từ tên bảng vừa được tạo ra, ta tạo Identifier xong dùng nó cho câu lệnh truy vấn chèn bộ dữ liệu vào bảng đó bằng lệnh **execute()** kết hợp với các placeholder **%s**, tương tự như khi ta chèn bộ dữ liệu này vào bảng **ratings** gốc.

```

119 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
183     # Tạo tên bảng phân mảnh chứa giá trị rating
184     partition_table = f"{RANGE_TABLE_PREFIX}{partition_index}"
185     partition_table_identifier = psycopg2.sql.Identifier(partition_table)
186
187     # Chèn dữ liệu vào bảng phân mảnh tương ứng
188     current.execute(
189         psycopg2.sql.SQL("INSERT INTO {} ({} ) VALUES (%s, %s, %s)").format(
190             partition_table_identifier,
191             select_columns_sql
192         ),
193         (userid, itemid, rating)
194     )
195

```

- Cuối cùng là sẽ xác nhận và lưu lại các thay đổi trên cơ sở dữ liệu

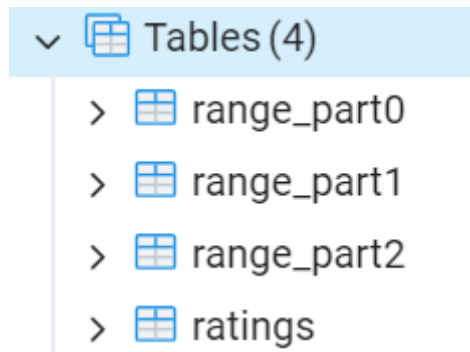
```

196     openconnection.commit()
197     print("Thêm dữ liệu vào bảng phân mảnh thành công!")

```

3. Kết quả và kiểm thử

- Giả sử ngay sau khi chạy hàm **rangepartition()** ở phía trên, chia bảng **ratings** gồm 10 triệu bộ dữ liệu từ file **ratings.dat** thành 3 phân mảnh có tên như sau:



- Ta sẽ thử nghiệm thêm 1 bộ dữ liệu mới vào phân mảnh phù hợp. Ta sẽ gọi hàm **rangeinsert()** và truyền vào nó các tham số bắt buộc như tên bảng **ratings**, **userid**, **itemid**, **rating**

```
211 if __name__ == "__main__":
212     try:
213         conn = getopenconnection()
214         loadratings(ratingstablename, INPUT_FILE_PATH, conn)
215         rangepartition(ratingstablename, 3, conn)
216         rangeinsert(ratingstablename, 1, 100, 2.5, conn)
217     except Exception as e:
218         print(f"Đã xảy ra lỗi: {e}")
219     finally:
220         if conn:
221             conn.close()
```

- Trước khi chạy trong bảng ratings và các phân mảnh không có bộ dữ liệu trên

Query	Query History
1	SELECT COUNT(*) FROM public.ratings
2	WHERE userid = 1 AND movieid = 100

Data Output	Messages	Notifications
<div> <div>≡+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> <div>SQL</div> </div>		
	count	
	bigint	🔒
1	0	

- Ở đây ta truyền thử vào 1 bộ dữ liệu với rating bằng 2.5, theo như logic ta viết thì bộ dữ liệu này sẽ được thêm vào bảng gốc lần phân mảnh phù hợp. Vì ta chia thành 3 phân mảnh với khoảng dữ liệu từng phân mảnh là $[0, 1.67]$, $(1.67, 3.34]$, $(3.34, 5]$, vì vậy bộ dữ liệu kiểm thử sẽ được thêm vào phân mảnh thứ 2 có tên là **range_part1**
- Kết quả sau khi chạy ta muốn bộ dữ liệu sẽ thêm vào bảng gốc và thêm vào phân mảnh phù hợp
- Tra cứu bảng ratings xem đã có bộ dữ liệu ta thêm vào hay không

Query	Query History
1	SELECT * FROM public.ratings
2	WHERE userid = 1 AND movieid = 100

Data Output	Messages	Notifications
<div> <div>≡+</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> <div>SQL</div> </div>		
	userid	movieid
	integer	integer
	🔒	🔒
		rating
		double precision
		🔒
1	1	100
		2.5

- Bộ dữ liệu đã được thêm vào bảng ratings, tiếp theo kiểm tra xem bộ dữ liệu có được thêm vào phân mảnh **range_part1** hay không

```
1 SELECT COUNT(*) FROM public.range_part0
2 WHERE userid = 1 AND movieid = 100
3
```

Data Output Messages Notifications

	count bigint
1	0

Query Query History

```
1 SELECT COUNT(*) FROM public.range_part2
2 WHERE userid = 1 AND movieid = 100
```

Data Output Messages Notifications

	count bigint
1	0

Query

Query History

1

▼

SELECT * FROM public.range_part1

2

WHERE userid = 1 AND movieid = 100

3

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

SQL

	userid integer	movieid integer	rating double precision
1	1	100	2.5

- Như vậy bộ dữ liệu đã thêm vào bảng ratings và đúng phân mảnh thích hợp, không thêm vào bất kỳ phân mảnh nào khác thỏa mãn yêu cầu bài toán.
- Tốc độ truy vấn chủ yếu là 2 lần INSERT do mọi xử lý tìm kiếm logic được xử lý riêng bằng các lệnh python.

VI. Tổng kết và kết luận

1. Đánh giá thuật toán

Qua quá trình thực hiện bài tập lớn, nhóm đã triển khai thành công hai phương pháp phân mảnh ngang (Range Partitioning và Round Robin Partitioning) trên cơ sở dữ liệu PostgreSQL, sử dụng tập dữ liệu đánh giá phim từ MovieLens. Các hàm LoadRatings(), Range_Partition(), RoundRobin_Partition(), cùng với cơ chế chèn dữ liệu động (Range_Insert() và RoundRobin_Insert()) đã được kiểm thử và hoạt động ổn định, đảm bảo dữ liệu được phân phối đúng phân mảnh dựa trên giá trị Rating hoặc nguyên tắc luân phiên.

Kết quả cho thấy:

- Range Partitioning phù hợp cho truy vấn theo khoảng giá trị (ví dụ: tìm phim có rating từ 3.5 đến 5), giúp giảm thời gian truy vấn nhờ loại bỏ dữ liệu không liên quan.
- Round Robin Partitioning cân bằng tải khi chèn dữ liệu, phù hợp cho hệ thống có tần suất ghi cao, nhưng không tối ưu cho truy vấn theo điều kiện cụ thể.

Bảng đánh giá tốc độ trung bình của từng thuật toán ở trong điều kiện thích hợp và $n = 10$ (với n là số phân mảnh)

	Thời gian hoàn thành
LoadRatings()	15s
Range_Partition()	17s
Range_Insert()	1s
RoundRobin_Partition()	26s
RoundRobin_Insert()	1s

2. Hạn chế và hướng phát triển

Bài toán chỉ mới xét với độ đánh giá cố định max là 5 và min là 0 và có thể chia nửa sao, với các bài toán có đánh giá chi tiết hơn như max là 10 sao thì không thể hoạt động.

Có thể mở rộng bằng cách kết hợp phân mảnh dọc (Vertical Partitioning) hoặc thử nghiệm trên hệ quản trị cơ sở dữ liệu phân tán.

3. Kết luận

Bài tập này không chỉ giúp nhóm hiểu rõ nguyên lý phân mảnh dữ liệu mà còn rèn luyện kỹ năng làm việc với PostgreSQL và Python trong các bài toán thực tế. Kết quả đạt được là cơ sở để tiếp tục nghiên cứu các kỹ thuật tối ưu hóa cơ sở dữ liệu quy mô lớn trong tương lai. Tổng kết lại, phân mảnh dữ liệu là một

công cụ mạnh để cân bằng giữa hiệu suất truy vấn và khả năng mở rộng, nhưng việc lựa chọn phương pháp phải dựa trên đặc thù ứng dụng và yêu cầu nghiệp vụ cụ thể.

--- HẾT ---