

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



BÁO 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

Thành viên nhóm:
NGUYỄN QUANG DŨNG B22DCCN133
NGUYỄN TUẤN NAM B22DCCN561
LÊ ĐĂNG NINH B22DCCN572

Hà Nội, 2025

Mục lục

Danh mục hình ảnh	3
Danh mục bảng biểu	3
1 Phân chia công việc	4
2 Phân tích yêu cầu đề bài	5
2.1 Bối cảnh hiện tại	5
2.2 Nhiệm vụ yêu cầu	5
2.3 Mục tiêu bài tập	5
2.4 Công nghệ sử dụng	7
3 Cơ sở lý thuyết	9
3.1 Phân mảnh	9
3.2 Range Partition	11
3.3 Round Robin Partition	12
3.4 So sánh Range Partition và Round Robin Partition	13
4 Phân tích và thiết kế giải pháp	14
4.1 Phân tích dữ liệu đầu vào	14
4.2 Phân tích và thiết kế các hàm chính	15
4.2.1 Hàm loadrating	15
4.2.2 Hàm rangepartition	15
4.2.3 Hàm roundrobinpartition	16
4.2.4 Hàm rangeinsert	16
4.2.5 Hàm roundrobininsert	17
4.3 Thiết kế giải pháp cho từng hàm chính	18
4.3.1 Hàm loadratings	18
4.3.2 Hàm rangepartition	18
4.3.3 Hàm roundrobinpartition	19
4.3.4 Hàm rangeinsert	19
4.3.5 Hàm roundrobininsert	20

5	Cài đặt và kiểm thử chương trình	21
5.1	Cài đặt chương trình	21
5.1.1	Hàm loadratings	21
5.1.2	Hàm rangepartition	22
5.1.3	Hàm roundrobinpartition	23
5.1.4	Hàm rangeinsert	24
5.1.5	Hàm roundrobininsert	25
5.2	Kiểm thử chương trình	26
5.2.1	Kiểm thử trên tập test_data.dat	26
5.2.2	Kiểm thử trên tập ratings.dat	27
6	Kết luận và hướng phát triển	30

Danh mục hình ảnh

1	file ratings.dat	14
2	Hàm loadratings	21
3	Hàm rangepartition	22
4	Hàm roundrobinpartition	23
5	Hàm rangeinsert	24
6	Hàm roundrobininsert	25
7	File test_data.dat	26
8	Kiểm thử trên tập test_data.dat bằng code nhóm thực hiện . .	27
9	File ratings.dat	27
10	Kiểm thử trên tập test_data.dat bằng code nhóm thực hiện . .	28
11	Kiểm thử trên tập test_data.dat bằng code mẫu của thầy đã thêm tính thời gian	28

Danh mục bảng biểu

1	Bảng phân chia công việc giữa các thành viên	4
2	So sánh Range Partition và Round Robin Partition	13
3	So sánh hiệu năng giữa code của nhóm và code mẫu của thầy .	29

1 Phân chia công việc

Thành viên	Công việc thực hiện
Nguyễn Quang Dũng	<ul style="list-style-type: none">- Thiết kế và cài đặt hàm- Viết tài liệu hướng dẫn sử dụng chương trình- Feedback báo cáo
Nguyễn Tuấn Nam	<ul style="list-style-type: none">- Kiểm thử chương trình- Feedback báo cáo
Lê Đăng Ninh	<ul style="list-style-type: none">- Kiểm thử chương trình- Tổng hợp và viết báo cáo

Bảng 1: Bảng phân chia công việc giữa các thành viên

2 Phân tích yêu cầu đề bài

2.1 Bối cảnh hiện tại

Trong thời đại dữ liệu phát triển mạnh mẽ hiện nay, việc xử lý và quản lý khối lượng dữ liệu lớn đang đặt ra nhiều thách thức cho các hệ thống cơ sở dữ liệu.

Khi dữ liệu gia tăng với tốc độ chóng mặt, các phương pháp truyền thống lưu trữ toàn bộ dữ liệu trong một bảng đơn lẻ dễ dẫn đến tình trạng suy giảm hiệu suất, hạn chế khả năng mở rộng và ảnh hưởng đến tính sẵn sàng của hệ thống.

Phân mảnh dữ liệu (Data Partitioning) là một giải pháp quan trọng trong thiết kế cơ sở dữ liệu phân tán, cho phép chia nhỏ một bảng dữ liệu lớn thành nhiều phân đoạn nhỏ hơn, gọi là các phân mảnh (partitions hoặc fragments).

-> Mỗi phân mảnh này có thể được lưu trữ và xử lý riêng biệt, giúp nâng cao hiệu quả truy vấn, rút ngắn thời gian phản hồi và hỗ trợ hệ thống mở rộng linh hoạt hơn.

2.2 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.

2.3 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
 - Cơ sở dữ liệu: PostgreSQL

- Ngôn ngữ lập trình: Python
- Áp dụng hai phương pháp phân mảnh:
 - **Range Partitioning:** Là phương pháp phân mảnh dữ liệu bằng cách chia bảng thành các phân mảnh dựa trên các khoảng giá trị của một thuộc tính nào đó (ví dụ: giá trị của Rating, UserID, hoặc MovieID). Mỗi phân mảnh chứa các bản ghi có giá trị thuộc tính nằm trong một khoảng xác định trước. Phương pháp này phù hợp khi dữ liệu có phân bố theo thứ tự rõ ràng và các truy vấn thường tập trung vào một khoảng cụ thể.
 - **Round-Robin Partitioning:** Là phương pháp phân mảnh dữ liệu bằng cách phân phối các bản ghi lần lượt và đều đặn vào các phân mảnh theo thứ tự vòng tròn. Bản ghi đầu tiên vào phân mảnh 1, bản ghi thứ hai vào phân mảnh 2, và cứ thế tiếp tục cho đến phân mảnh cuối cùng rồi quay lại phân mảnh 1. Phương pháp này đảm bảo các phân mảnh có kích thước gần như bằng nhau, giúp cân bằng tải, nhưng không quan tâm đến giá trị của dữ liệu nên không tối ưu cho các truy vấn theo điều kiện.
- Các thao tác yêu cầu:
 - **Load dữ liệu vào một bảng gốc tên ratings:**
 - * Viết các hàm Python để đọc dữ liệu từ tệp `ratings.dat` theo đúng định dạng (UserID::MovieID::Rating::Timestamp).
 - * Chuyển dữ liệu thành định dạng phù hợp (ví dụ: tách các trường bằng dấu ::) và tải vào một bảng quan hệ ban đầu có tên là `ratings` trong hệ quản trị cơ sở dữ liệu đã chọn (PostgreSQL hoặc MySQL).
 - * Đảm bảo bảng `ratings` có các cột tương ứng: UserID, MovieID, Rating, Timestamp, với kiểu dữ liệu phù hợp.
 - **Phân mảnh bảng này theo:**
 - * **Khoảng (Range):**

- Tạo các bảng phân mảnh riêng biệt (ví dụ: `ratings_range_part1`, `ratings_range_part2`, ...) dựa trên các khoảng giá trị của trường Rating hoặc trường khác.
- Viết hàm để duyệt qua dữ liệu trong bảng `ratings` và chèn từng bản ghi vào phân mảnh phù hợp theo khoảng giá trị đã định nghĩa.
- * **Vòng luân phiên (Round-Robin):**
 - Tạo các bảng phân mảnh (ví dụ: `ratings_rr_part1`, ...) theo số lượng phân mảnh đã định trước.
 - Viết hàm Python để phân phối các bản ghi từ bảng `ratings` vào các phân mảnh theo thứ tự tuần tự, xoay vòng đều đặn.
- **Chèn dữ liệu mới và tự động đưa vào đúng phân mảnh tương ứng:**
 - * Xây dựng các hàm hỗ trợ chèn bản ghi mới vào hệ thống.
 - * Đảm bảo khi một bản ghi mới được thêm vào, hệ thống sẽ tự động xác định và chuyển bản ghi đó vào phân mảnh phù hợp:
 - Nếu phân mảnh theo Range: xác định khoảng giá trị tương ứng với bản ghi.
 - Nếu phân mảnh theo Round-Robin: tiếp tục phân phối theo thứ tự lần lượt dựa trên chỉ số chèn trước đó.
 - * Có thể sử dụng một bảng phụ hoặc biến hệ thống để theo dõi chỉ số phân phối hiện tại trong phương pháp Round-Robin.

2.4 Công nghệ sử dụng

- **Hệ điều hành:** Ubuntu 22.04 / Windows 10
- **Ngôn ngữ lập trình:** Python 3.12.x

Phiên bản Python 3.12.x đảm bảo tính tương thích với các thư viện hiện đại, cải thiện hiệu suất và hỗ trợ tốt hơn về quản lý bộ nhớ, giúp xây dựng các ứng dụng cơ sở dữ liệu phân tán hiệu quả và ổn định.

- **Hệ quản trị SQL:** PostgreSQL

PostgreSQL được sử dụng làm hệ quản trị cơ sở dữ liệu chính nhờ vào tính năng mạnh mẽ, hỗ trợ đầy đủ các chuẩn SQL, khả năng mở rộng và tính bảo mật cao.

PostgreSQL không chỉ hỗ trợ tốt việc lưu trữ và truy vấn dữ liệu phức tạp mà còn cung cấp các công cụ quản trị mạnh như replication (nhân bản), partitioning (phân mảnh), indexing nâng cao, rất phù hợp với các hệ thống cơ sở dữ liệu phân tán.

- **Thư viện Python:** psycopg2

Đây là thư viện phổ biến nhất để kết nối Python với PostgreSQL. `psycopg2` cho phép thực hiện các thao tác như kết nối cơ sở dữ liệu, truy vấn, thêm, sửa, xóa dữ liệu một cách dễ dàng và hiệu quả.

Trong quá trình triển khai, `psycopg2` được sử dụng để viết các đoạn mã thực hiện phân mảnh, đồng bộ hóa dữ liệu và kiểm tra tính toàn vẹn.

- **IDE:** VSCode / PyCharm

VSCode nổi bật với giao diện nhẹ, khả năng mở rộng mạnh mẽ thông qua các plugin, hỗ trợ đa nền tảng và cộng đồng sử dụng rộng rãi.

PyCharm là một IDE chuyên sâu dành cho Python, hỗ trợ tốt cho việc kiểm tra mã nguồn, gỡ lỗi (debug), và tích hợp công cụ quản lý cơ sở dữ liệu.

3 Cơ sở lý thuyết

3.1 Phân mảnh

a. Định nghĩa

Phân mảnh là quá trình chia một cơ sở dữ liệu thành các phần nhỏ hơn gọi là mảnh (fragments) để lưu trữ trên các vị trí phân tán khác nhau trong hệ cơ sở dữ liệu phân tán. Mỗi mảnh vẫn có thể được xử lý độc lập và đảm bảo rằng dữ liệu vẫn có thể truy xuất đầy đủ và chính xác khi cần thiết.

b. Phân loại

- **Phân mảnh ngang:** Là dạng phân mảnh trong đó các dòng (bản ghi, tuples) của một bảng được chia thành nhiều tập con khác nhau, mỗi tập con là một mảnh.
 - Mỗi mảnh ngang chứa một tập hợp con của các hàng trong bảng.
 - Các mảnh này được xác định bằng điều kiện phân mảnh (ví dụ: `WHERE region = 'HCM'`).
 - Các mảnh phải không trùng lặp và hợp lại sẽ cho ra bảng ban đầu.
 - **Các kiểu phân mảnh ngang:** Range Partition, Round Robin Partition, Hash Partition, List Partition.
 - **Ví dụ:** Giả sử có bảng `Customers(id, name, region)`:
 - * Fragment 1 (F1): chứa các khách hàng ở miền Bắc
`SELECT * FROM Customers WHERE region = 'North';`
 - * Fragment 2 (F2): chứa các khách hàng ở miền Nam
`SELECT * FROM Customers WHERE region = 'South';`
 - * \Rightarrow Tập hợp $F1 \cup F2 =$ bảng `Customers` ban đầu.
- **Phân mảnh dọc:** Là kỹ thuật chia một bảng thành các mảnh dựa trên các cột (thuộc tính). Mỗi mảnh sẽ chứa một tập con của các cột ban đầu (kèm theo khóa chính để đảm bảo khả năng ghép nối).
 - Mỗi mảnh dọc có tập cột riêng biệt, nhưng phải bao gồm khóa chính để có thể kết hợp lại bảng gốc khi cần.

- Dữ liệu vẫn có thể tái tạo bằng JOIN theo khóa chính.
- **Ví dụ:** Bảng Employee(id, name, address, salary):
 - * Fragment 1: Employee_F1(id, name, address)
 - * Fragment 2: Employee_F2(id, salary)
 - * \Rightarrow Tái tạo lại bảng gốc bằng:


```
SELECT * FROM Employee_F1 JOIN Employee_F2 USING(id);
```
- **Phân mảnh hỗn hợp:** Kết hợp của phân mảnh ngang và dọc. Có thể phân mảnh ngang trước rồi phân mảnh dọc, hoặc ngược lại.
 - **Ví dụ:** Bảng Customer(id, name, region, phone, email)
 - * Bước 1: Phân mảnh ngang theo region
 - F1: Khách hàng miền Bắc
 - F2: Khách hàng miền Nam
 - * Bước 2: Phân mảnh dọc F1 và F2
 - F1_1: (id, name, region)
 - F1_2: (id, phone, email)
 - F2_1: (id, name, region)
 - F2_2: (id, phone, email)
 - * \Rightarrow Khôi phục bảng gốc:
 - JOIN F1_1 với F1_2 \rightarrow F1
 - JOIN F2_1 với F2_2 \rightarrow F2
 - UNION F1 với F2 \rightarrow Bảng Customer

c. Tác dụng

- Tăng hiệu suất truy xuất dữ liệu tại các site cục bộ.
 - Khi dữ liệu được phân mảnh và lưu trữ tại các site (địa điểm) khác nhau, mỗi site có thể lưu trữ các mảnh dữ liệu mà nó thường xuyên truy cập.
 - Điều này có nghĩa là các truy vấn do người dùng tại site đó thực hiện có thể được xử lý ngay tại chỗ mà không cần truy xuất dữ liệu từ site khác.

- Giảm lưu lượng truyền dữ liệu qua mạng.
 - Khi áp dụng phân mảnh hiệu quả (đặc biệt là theo cách phân tích các mô hình truy cập), các mảnh dữ liệu được lưu tại nơi chúng thường xuyên được truy vấn.
 - Điều này giúp giảm đáng kể số lượng truy vấn và khối lượng dữ liệu phải truyền qua mạng, tiết kiệm băng thông và giảm chi phí vận hành mạng.
- Cải thiện tính sẵn sàng và khả năng song song của hệ thống.
 - Phân mảnh giúp tăng tính sẵn sàng của hệ thống vì các site có thể hoạt động độc lập ngay cả khi một hoặc vài site khác bị lỗi hoặc gián đoạn.
 - Nhờ đó, hiệu năng tổng thể của hệ thống được cải thiện đáng kể, đặc biệt trong môi trường có nhiều người dùng truy cập đồng thời.

3.2 Range Partition

Định nghĩa:

Là kỹ thuật phân mảnh ngang trong đó các bản ghi được chia thành các mảnh dựa trên giá trị của một thuộc tính nhất định nằm trong các khoảng xác định trước (thường là khóa hoặc cột có thứ tự, ví dụ ngày tháng, ID, số tiền,...).

Đặc điểm:

- Dữ liệu được chia theo điều kiện phạm vi (range).
- Mỗi mảnh chứa các dòng có giá trị trong một khoảng nhất định.
- Các khoảng không được chồng lặp nhau.

Ví dụ:

Bảng `Orders(order_id, customer_id, total_amount)` phân mảnh theo `total_amount`

- Fragment 1 (F1): `total_amount < 100`
- Fragment 2 (F2): `100 ≤ total_amount < 500`

- Fragment 3 (F3): `total_amount` \geq 500

Khi truy vấn:

```
SELECT * FROM Orders WHERE total_amount >= 500;
```

→ Hệ thống chỉ cần truy vấn F3, giúp tiết kiệm thời gian và tài nguyên.

3.3 Round Robin Partition

Định nghĩa:

Là kỹ thuật phân mảnh ngang đơn giản, trong đó các bản ghi được phân phối đều đặn và tuần tự vào các mảnh theo vòng tròn, không phụ thuộc vào giá trị thuộc tính nào.

Đặc điểm:

- Mỗi bản ghi tiếp theo sẽ được gán vào mảnh kế tiếp theo thứ tự vòng tròn.
- Đảm bảo các mảnh có kích thước gần như bằng nhau.
- Không tối ưu cho các truy vấn chọn lọc theo điều kiện.

Ví dụ:

Với bảng `Employee` và 3 mảnh dữ liệu:

- Dòng 1 → Fragment 1
- Dòng 2 → Fragment 2
- Dòng 3 → Fragment 3
- Dòng 4 → Fragment 1
- Dòng 5 → Fragment 2
- ...

Ưu điểm:

- Phân phối dữ liệu đồng đều, giúp cân bằng tải.

- Dễ triển khai, không cần phân tích dữ liệu.

Nhược điểm:

- Không hỗ trợ truy vấn theo điều kiện hiệu quả, vì phải tìm kiếm trên tất cả các mảnh.

3.4 So sánh Range Partition và Round Robin Partition

Tiêu chí	Range Partition	Round Robin Partition
Phân phối dữ liệu	Theo khoảng giá trị, có thể không đều	Đều, tuần tự từng dòng
Dễ bảo trì	Đơn giản	Đơn giản
Khả năng mở rộng	Dễ mở rộng	Dễ mở rộng
Thích hợp với loại dữ liệu	Có tính thứ tự hoặc theo vùng giá trị rõ ràng (thời gian, mã ID,...)	Không có mẫu truy vấn rõ ràng, cần phân phối đều
Hiệu suất INSERT	Nhanh – $O(1)$ nếu có sẵn khóa phân mảnh, phải kiểm tra khoảng phù hợp	Rất nhanh – $O(1)$, chỉ cần ghi lần lượt theo thứ tự vòng
Hiệu suất SELECT với điều kiện	Rất nhanh nếu có điều kiện trên partition key – partition elimination	Chậm – phải scan toàn bộ các partition

Bảng 2: So sánh Range Partition và Round Robin Partition

4 Phân tích và thiết kế giải pháp

4.1 Phân tích dữ liệu đầu vào

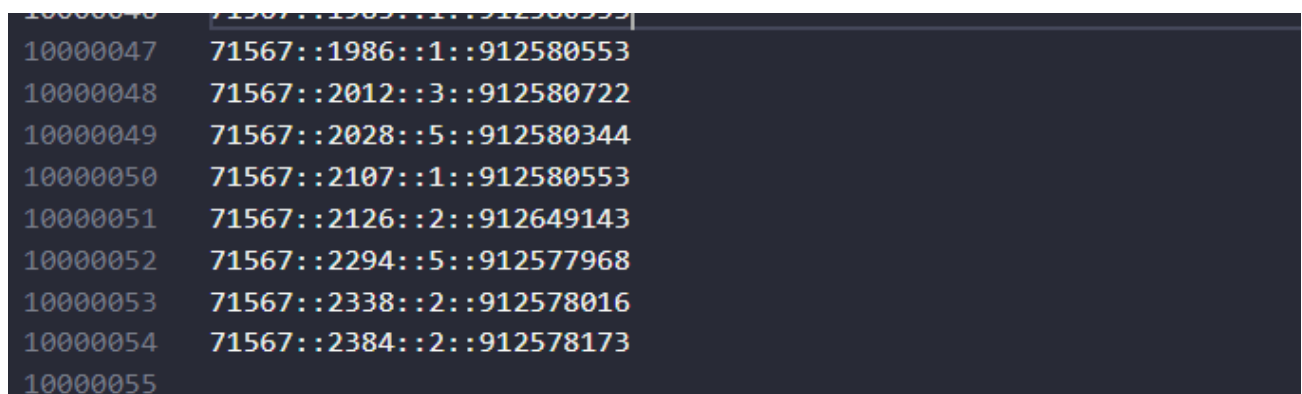
Dữ liệu đầu vào sử dụng trong bài toán là tệp `ratings.dat` từ bộ dữ liệu MovieLens 10M. Mỗi dòng trong tệp có định dạng như sau:

`UserID::MovieID::Rating::Timestamp`

Ý nghĩa các trường:

- **UserID**: ID của người dùng (kiểu số nguyên).
- **MovieID**: ID của bộ phim (kiểu số nguyên).
- **Rating**: Điểm đánh giá của người dùng cho bộ phim, trên thang điểm 5 (có thể có giá trị lẻ như 3.5, 4.0, 5.0).
- **Timestamp**: Dấu thời gian, là số giây kể từ 0 giờ UTC ngày 1 tháng 1 năm 1970 (dạng Unix timestamp).

Dữ liệu cần sử dụng: Trong quá trình xử lý và phân mảnh, chỉ ba trường đầu tiên được sử dụng: `UserID`, `MovieID`, `Rating`. Trường `Timestamp` có thể bỏ qua vì không ảnh hưởng đến quá trình phân mảnh.



```
10000047 71567::1986::1::912580553
10000048 71567::2012::3::912580722
10000049 71567::2028::5::912580344
10000050 71567::2107::1::912580553
10000051 71567::2126::2::912649143
10000052 71567::2294::5::912577968
10000053 71567::2338::2::912578016
10000054 71567::2384::2::912578173
10000055
```

Hình 1: file `ratings.dat`

Mỗi dòng thể hiện một đánh giá của người dùng đối với một bộ phim tại một thời điểm nhất định. Dữ liệu này sẽ được nạp vào bảng `ratings` trong cơ sở dữ liệu để thực hiện các thao tác phân mảnh tiếp theo.

4.2 Phân tích và thiết kế các hàm chính

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 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:

- `ratingtablename`: 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.

4.2.5 Hàm `roundrobininsert`

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

Đầu vào:

- `ratingtablename`: 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.

4.3 Thiết kế giải pháp cho từng hàm chính

4.3.1 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).

4.3.2 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.

4.3.3 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 của số `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.

4.3.4 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.

4.3.5 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 bảng con round-robin 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.

5 Cài đặt và kiểm thử chương trình

5.1 Cài đặt chương trình

5.1.1 Hàm loadratings

```
def loadratings(ratingtablename, 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 = '{ratingtablename}');")
    if cur.fetchone()[0]:
        cur.execute(f"DROP TABLE {ratingtablename};")

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

    # Tạo bảng tạm để tải dữ liệu
    temp_table = ratingtablename + "_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 {ratingtablename} (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

5.1.2 Hàm rangepartition

```
def rangepartition(ratingtablename, 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 {ratingtablename} "
                        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 {ratingtablename} "
                        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: $\text{delta} = 5 / N$
- Với mỗi phân mảnh i , tạo bảng `range_part{i}` và lấy dữ liệu từ bảng gốc:
 - Nếu $i = 0$: lấy $\text{rating} \geq \text{min}$ AND $\leq \text{max}$
 - Nếu $i > 0$: lấy $\text{rating} > \text{min}$ AND $\leq \text{max}$

5.1.3 Hàm roundrobinpartition

```
def roundrobinpartition(ratingtablename, 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 {ratingtablename}) 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

5.1.4 Hàm rangeinsert

```
def rangeinsert(ratingtablename, 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 {ratingtablename} (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

5.1.5 Hàm roundrobininsert

```
def roundrobininsert(ratingtablename, 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 {ratingtablename} (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 {ratingtablename};")
    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

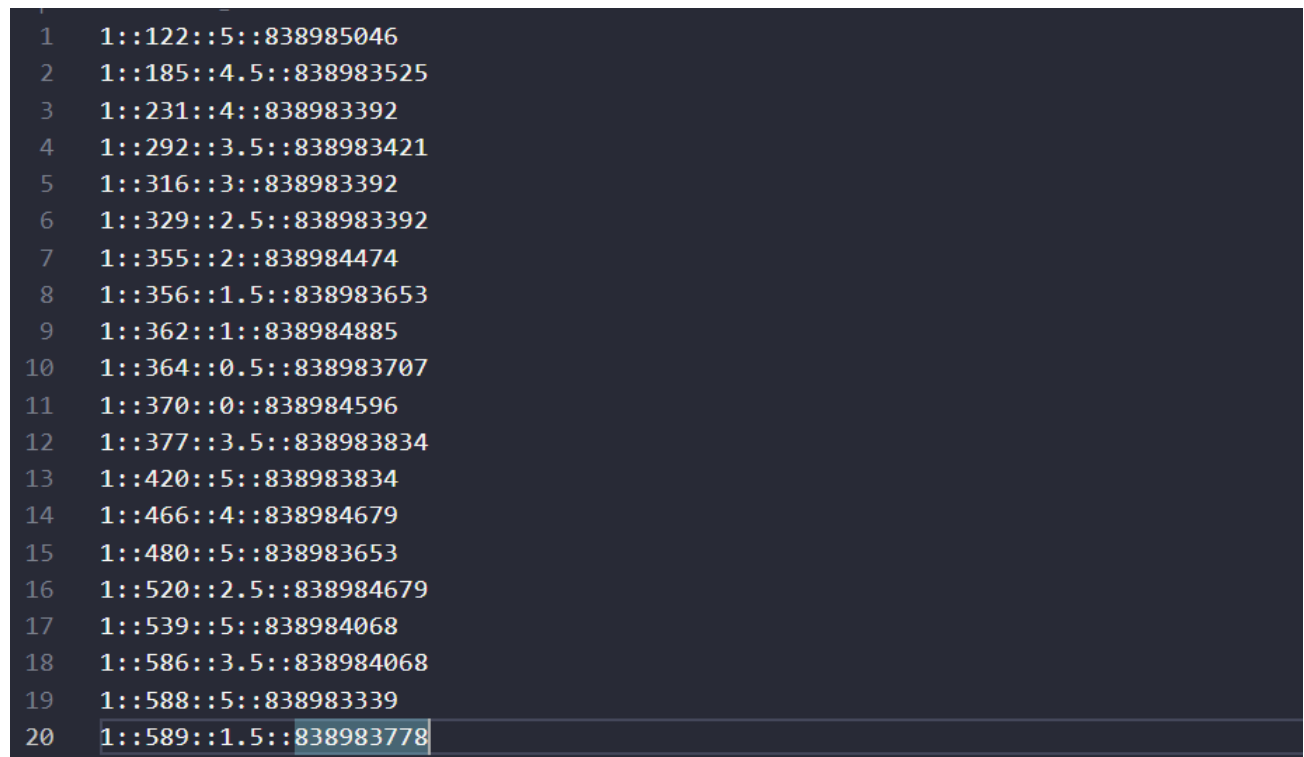
Cụ thể chi tiết xử lý:

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

5.2 Kiểm thử chương trình

5.2.1 Kiểm thử trên tập test_data.dat

Đặc điểm của tập dữ liệu: dữ liệu có định dạng như sau: UserID::MovieID::Rating::Time và có 20 dòng



```
1 1::122::5::838985046
2 1::185::4.5::838983525
3 1::231::4::838983392
4 1::292::3.5::838983421
5 1::316::3::838983392
6 1::329::2.5::838983392
7 1::355::2::838984474
8 1::356::1.5::838983653
9 1::362::1::838984885
10 1::364::0.5::838983707
11 1::370::0::838984596
12 1::377::3.5::838983834
13 1::420::5::838983834
14 1::466::4::838984679
15 1::480::5::838983653
16 1::520::2.5::838984679
17 1::539::5::838984068
18 1::586::3.5::838984068
19 1::588::5::838983339
20 1::589::1.5::838983778
```

Hình 7: File test_data.dat

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:\3Y25\csdlpt> & d:/3Y25/csdlpt/.venv/Scripts/python.exe d:/3Y25/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:\3Y25\csdlpt>
```

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

5.2.2 Kiểm thử trên tập ratings.dat

Đặc điểm của tập dữ liệu: UserID::MovieID::Rating::Timestamp và có 10000054 dòng

```
10000046 71567::1986::1::912580553
10000047 71567::1986::1::912580553
10000048 71567::2012::3::912580722
10000049 71567::2028::5::912580344
10000050 71567::2107::1::912580553
10000051 71567::2126::2::912649143
10000052 71567::2294::5::912577968
10000053 71567::2338::2::912578016
10000054 71567::2384::2::912578173
10000055
```

Hình 9: File ratings.dat

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

```
PS D:\3Y2S\csdlpt> & d:/3Y2S/csdlpt/.venv/Scripts/python.exe d:/3Y2S/csdlpt/abc/bai_tap_lon_CSDL_phan_tan/Assignment1Tester.py
A database named "csdlpt" already exists
[loadratings] Thời gian xử lý: 27.8677 giây
loadratings function pass!
[rangepartition] Thời gian xử lý: 20.4091 giây
rangepartition function pass!
[rangeinsert] Thời gian xử lý: 0.003000 giây
rangeinsert function pass!
[loadratings] Thời gian xử lý: 32.1877 giây
[roundrobinpartition] Thời gian xử lý: 33.5454 giây
roundrobinpartition function pass!
[roundrobininsert] Thời gian xử lý: 0.242677 giây
roundrobininsert function pass!
Press enter to Delete all tables?
PS D:\3Y2S\csdlpt>
```

Hình 11: 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 3: 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.

6 Kết luận và hướng phát triển

Ưu điểm:

- **Đáp ứng đầy đủ yêu cầu đề bài:**
 - Tất cả các chức năng được đề cập trong yêu cầu kỹ thuật đã được triển khai thành công.
 - Hệ thống đảm bảo thực hiện chính xác các kỹ thuật phân mảnh được mô tả (như Round Robin, Range...).
 - Cấu trúc mã nguồn rõ ràng, dễ hiểu và có khả năng mở rộng hoặc chỉnh sửa khi cần thiết.
 - Giao diện dòng lệnh hoặc giao diện web (nếu có) hỗ trợ thao tác dễ dàng, thân thiện với người dùng.
- **Dữ liệu được xử lý chính xác, kiểm thử pass toàn bộ:**
 - Tất cả các bài kiểm thử (test cases) – bao gồm chèn dữ liệu, phân mảnh, truy vấn, thống kê – đều cho kết quả đúng như mong đợi.
 - Không có lỗi logic, không phát sinh ngoại lệ (exception) trong quá trình chạy thử.
 - Các kết quả phân mảnh được kiểm chứng lại bằng truy vấn SQL và so sánh tay, đảm bảo tính toàn vẹn và hợp lý.
- **Sử dụng PostgreSQL hiệu quả qua COPY và ROW_NUMBER():**
 - Lệnh COPY giúp tăng tốc độ nhập dữ liệu từ file CSV vào cơ sở dữ liệu PostgreSQL – nhanh gấp nhiều lần so với cách dùng INSERT từng dòng. Điều này giúp tiết kiệm thời gian khởi tạo dữ liệu ban đầu, đặc biệt với khối lượng lớn.
 - Hàm ROW_NUMBER() được sử dụng thông minh trong quá trình phân mảnh kiểu Round Robin:
 - * Cho phép đánh số thứ tự từng bản ghi một cách chính xác.

- * Kết hợp với modulo (%) để xác định vị trí phân mảnh cho từng dòng – hiệu quả, ngắn gọn, rõ ràng.

Hạn chế:

- **Hiệu suất với dữ liệu lớn:**

- Tập dữ liệu MovieLens chứa 10 triệu đánh giá, rất lớn cho hệ thống.
- Những thuật toán phân mảnh đều phải quét bảng nhiều lần, thời gian hoàn thành xử lý kéo dài.

- **Xử lý chính xác các trường hợp biên:**

- Đảm bảo giá trị 0.0 và 5.0 được phân loại đúng vào các phân mảnh tương ứng.
- Tránh sai số trong quá trình tính toán các khoảng phân mảnh.

- **Quản lý bộ nhớ và tài nguyên:**

- Dữ liệu có thể bị giữ quá lâu trong bộ nhớ dù không dùng đến.
- Có khả năng bộ nhớ bị tràn khi phải xử lý lượng lớn thông tin cùng một thời điểm.

Đề xuất phát triển:

- Thêm bảng metadata để quản lý trạng thái chèn round robin chính xác hơn và ghi lại:

- Partition hiện tại đang chờ chèn tiếp theo.
- Tổng số bản ghi ở mỗi partition (để giám sát cân bằng).
- Lịch sử phân phối (tùy chọn).

- Giao diện web để xem nội dung phân mảnh:

- Xem danh sách các partition hiện có và bảng dữ liệu trong từng partition.
- Trình bày biểu đồ phân phối dữ liệu (dạng thanh, donut...).

- Thực hiện thao tác như: tạo partition mới, gán lại dữ liệu, xem thống kê chi tiết.
- Giao diện truy vấn thử dữ liệu từ các partition.
- Hỗ trợ phân mảnh dọc và kết hợp:
 - **Phân mảnh dọc:** Tách bảng theo cột, giúp mỗi site chỉ lưu những thuộc tính cần thiết, giảm kích thước bản ghi, tối ưu lưu trữ và truyền dữ liệu. Phù hợp khi các ứng dụng địa phương chỉ sử dụng một phần thông tin của bảng.
 - **Phân mảnh kết hợp:** Kết hợp phân mảnh ngang và dọc. Ví dụ: trước tiên phân mảnh theo khoảng (range) rồi tách theo cột. Giải pháp hiệu quả cho hệ thống lớn, phức tạp và nhiều truy vấn đa chiều.

Bài học rút ra:

- Tạo bản thiết kế trước khi thực hiện các bước khác.
- Đảm bảo cơ sở dữ liệu không bị quá tải, tối ưu hóa thường xuyên.
- Kiểm thử thường xuyên và kỹ càng để tránh các lỗi.