

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

**KHOA CÔNG NGHỆ THÔNG TIN 1**

---



## **BÁO CÁO CUỐI KỲ**

**Môn: Cơ sở dữ liệu phân tán**

Nhóm lớp: 10

**Giảng viên hướng dẫn:**

**Kim Ngọc Bách**

**Nhóm:**

09

**Nhóm sinh viên thực hiện:**

Nguyễn Khánh Đăng – B22DCCN209 (NT)

Đào Ngọc Đức – B22DCCN221

Nguyễn Trí Dũng – B22DCCN135

**Hà Nội – 2025**

## Lời mở đầu

Trong thời đại công nghệ thông tin phát triển mạnh mẽ, nhu cầu xây dựng các hệ thống cơ sở dữ liệu có khả năng xử lý phân tán, hiệu năng cao và dễ mở rộng ngày càng trở nên thiết yếu. Một trong những kỹ thuật then chốt nhằm đạt được những mục tiêu này là phân mảnh dữ liệu – cho phép phân chia bảng dữ liệu thành nhiều phần nhỏ hơn để tối ưu hóa truy vấn, phân phối dữ liệu và giảm thiểu chi phí truyền tải trong các hệ thống phân tán.

Với mục tiêu củng cố kiến thức lý thuyết và nâng cao khả năng ứng dụng thực tiễn, đề tài "Mô phỏng các phương pháp phân mảnh dữ liệu trên một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở" được thực hiện nhằm giúp chúng em hiểu rõ hơn về các phương pháp phân mảnh ngang. Trong khuôn khổ đề tài, chúng em đã xây dựng một tập các hàm Python để:

- Tải dữ liệu đầu vào vào một bảng quan hệ,
- Tiến hành phân mảnh bảng dữ liệu theo nhiều phương pháp phân mảnh ngang khác nhau,
- Và xử lý việc chèn dữ liệu mới vào đúng phân mảnh tương ứng.

Thông qua quá trình nghiên cứu và thực hiện đề tài, chúng em không chỉ nắm vững hơn các kiến thức về mô hình cơ sở dữ liệu phân tán mà còn rèn luyện được kỹ năng lập trình, tư duy hệ thống và cách phối hợp làm việc nhóm hiệu quả.

Chúng em xin chân thành cảm ơn thầy **Kim Ngọc Bách** – giảng viên hướng dẫn đã tận tình giảng dạy, định hướng và hỗ trợ chúng em trong suốt quá trình học tập và thực hiện đề tài này. Sự hướng dẫn nhiệt huyết của thầy là nguồn động viên to lớn để chúng em hoàn thành tốt nhiệm vụ được giao.

Chúng em kính mong nhận được những góp ý quý báu từ thầy để có thể hoàn thiện đề tài tốt hơn và tiếp tục phát triển năng lực chuyên môn trong lĩnh vực cơ sở dữ liệu.

Trân trọng!

*Hà Nội, ngày 10 tháng 06 năm 2025*

Nhóm sinh viên thực hiện

**Nhóm 09**

# Mục lục

Danh mục hình vẽ.....	1
Phân chia công việc .....	2
CHƯƠNG I: GIỚI THIỆU TỔNG QUAN.....	3
1. Giới thiệu tổng quan về bài toán: .....	3
2. Phạm vi đề tài: .....	4
3. Nền tảng lý thuyết các phương pháp phân mảnh:.....	4
3.1 Phân mảnh theo khoảng (range partitioning) .....	5
3.2 Phân mảnh theo Round-Robin (mô phỏng ở tầng ứng dụng).....	5
3.3 Bảng so sánh tổng hợp: .....	6
4. Tổng quan chiến lược giải quyết các nhiệm vụ chính.....	7
4.1 Tải dữ liệu (load ratings): .....	7
4.2 Phân mảnh theo khoảng (range partition) .....	8
4.3 Phân mảnh theo Round-robin (round robin partition) .....	8
4.4 Chèn dữ liệu vào phân mảnh theo khoảng (range insert) .....	8
4.5 Chèn dữ liệu vào phân mảnh Round-robin (round robin insert).....	9
5. Cách thiết kế và xử lý yêu cầu .....	9
6. Cấu trúc dữ liệu sử dụng: .....	10
7. Thách thức gặp phải và giải pháp: .....	10
7.1 Hiệu suất:.....	10
7.2 Tính nhất quán trong Round-Robin: .....	10
7.3 Quản lý metadata: .....	11
CHƯƠNG II: PHÂN TÍCH CHI TIẾT .....	12
1. Mục đích và phạm vi: .....	12
2. Các hằng số và cấu trúc cơ sở dữ liệu: .....	12
2.1 Cấu trúc cơ sở dữ liệu:.....	12
2.2 Các hằng số: .....	13
3. Các hàm tiện ích .....	14
3.1 Hàm <code>getopenconnection(user, password, dbname)</code> .....	14
3.2 Hàm <code>create_db(dbname)</code> :.....	14
3.3 Hàm <code>get_partition_count(prefix, openconnection)</code> .....	15
3.4 Hàm <code>validate_rating(rating)</code> .....	16

<b>4. Phân tích các hàm chính.....</b>	<b>17</b>
4.1 Hàm loadratings .....	17
4.2 Hàm rangepartition.....	18
4.3 Hàm roundrobinpartition .....	20
4.4 Hàm range insert .....	21
4.5 Hàm round robin insert.....	22
<b>CHƯƠNG III: KẾT QUẢ ĐẠT ĐƯỢC .....</b>	<b>25</b>
1. Tổng quan về kết quả thực hiện:.....	25
2. Phân tích bảng dữ liệu gốc: .....	25
3. Phân tích kết quả phân mảnh theo khoảng (Range partitioning):.....	26
4. Phân tích kết quả phân mảnh vòng tròn (Range partitioning):.....	30
<b>KẾT LUẬN.....</b>	<b>33</b>
<b>TÀI LIỆU THAM KHẢO .....</b>	<b>34</b>

## Danh mục hình vẽ

Hình 1. Dữ liệu ratings.dat.....	3
Hình 2. Thư viện sử dụng .....	13
Hình 3. Các hằng số cục bộ chính.....	14
Hình 4. Hàm kết nối CSDL.....	14
Hình 5. Hàm tạo CSDL .....	15
Hình 6. Hàm get_partition_count.....	16
Hình 7. Hàm validate_rating .....	16
Hình 8. Hàm load_ratings .....	18
Hình 9. Hàm range_partition .....	19
Hình 10. Hàm round robin partition .....	21
Hình 11. Hàm range insert .....	22
Hình 12. Hàm round robin insert .....	24
Hình 13. Tổng quan kết quả kiểm thử.....	25
Hình 14. Bảng ratings_1 .....	25
Hình 15. Bảng ratings_2 .....	26
Hình 16. Bảng range_part 0.....	27
Hình 17. Bảng range_part 1 .....	27
Hình 18. Bảng range_part 2.....	28
Hình 19. Bảng range_part 3.....	29
Hình 20. Bảng range_part 4.....	29
Hình 21. Bảng rrobin_part 0 .....	30
Hình 22. Bảng rrobin_part 1 .....	31
Hình 23. Bảng rrobin_part 2 .....	31
Hình 24. Bảng rrobin_part 3 .....	32
Hình 25. Bảng rrobin_part 4 .....	32

## Phân chia công việc

STT	Họ và tên	Mã sinh viên	Nội dung công việc	Mức độ hoàn thành
1	Nguyễn Khánh Đăng (NT)	B22DCCN209	- Cài đặt hàm Python Range_Insert() - Cài đặt hàm Python RoundRobin_Insert()	Đã hoàn thành 100% công việc được giao
2	Đào Ngọc Đức	B22DCCN221	- Cài đặt hàm Range_partition(). - Viết báo cáo môn học.	Đã hoàn thành 100% công việc được giao
3	Nguyễn Trí Dũng	B22DCCN135	- Cài đặt hàm LoadRatings(). - Cài đặt hàm RoundRobin_Partition()	Đã hoàn thành 100% công việc được giao

# CHƯƠNG I: GIỚI THIỆU TỔNG QUAN

## 1. Giới thiệu tổng quan về bài toán:

Bài tập lớn này tập trung vào việc mô phỏng các kỹ thuật phân mảnh dữ liệu ngang, một khía cạnh quan trọng trong quản trị và tối ưu hóa hiệu năng cơ sở dữ liệu quan hệ. Cụ thể, mục tiêu của bài toán bao gồm các nhiệm vụ chính sau:

- Tải dữ liệu: Đọc và nhập dữ liệu đánh giá phim từ tệp ratings.dat (thuộc bộ dữ liệu MovieLens 10M) vào một bảng quan hệ trong hệ quản trị cơ sở dữ liệu PostgreSQL.

	userid	movieid	rating
	1	316	3
	1	329	2.5
	1	355	2
	1	356	1.5
	1	362	1
	1	364	0.5
	1	370	0
	1	377	3.5
	1	420	5
	1	466	4
	1	480	5
	1	520	2.5
	1	539	5
	1	586	3.5
	1	588	5
	1	589	1.5

Hình 1. Dữ liệu ratings.dat

- Phân mảnh dữ liệu: Triển khai và áp dụng hai phương pháp phân mảnh ngang phổ biến cho bảng dữ liệu đã tải:
  - Phân mảnh theo khoảng (Range Partitioning): Chia dữ liệu vào các phân mảnh khác nhau dựa trên các khoảng giá trị được xác định trước của thuộc tính Rating.
  - Phân mảnh theo kiểu vòng tròn (Round-Robin Partitioning): Phân phối các bản ghi một cách tuần tự và đều đặn cho các phân mảnh.
- Chèn dữ liệu vào phân mảnh: Xây dựng các hàm cho phép chèn một bộ dữ liệu (tuple) đánh giá mới vào bảng gốc, đồng thời đảm bảo bộ dữ liệu này

cũng được định tuyến và chèn chính xác vào phân mảnh tương ứng theo từng phương pháp phân mảnh đã được áp dụng.

## **2. Phạm vi đề tài:**

Để thực hiện các yêu cầu của bài toán, nhóm đã thống nhất sử dụng cách tiếp cận sau:

- Ngôn ngữ và thư viện: Sử dụng ngôn ngữ lập trình Python (phiên bản 3.x) làm ngôn ngữ chính. Thư viện psycopg2 được chọn để thực hiện các tương tác với cơ sở dữ liệu PostgreSQL, bao gồm việc kết nối, thực thi các câu lệnh SQL, và quản lý giao dịch.
- Thiết kế hàm (Modularity): Bài toán được chia thành các module chức năng nhỏ hơn, mỗi module được triển khai dưới dạng một hàm Python riêng biệt. Cụ thể, có các hàm để:
  - Thiết lập kết nối đến cơ sở dữ liệu (getopenconnection).
  - Tạo cơ sở dữ liệu nếu chưa tồn tại (create\_db).
  - Tải dữ liệu từ tệp vào bảng chính (loadratings).
  - Thực hiện phân mảnh theo khoảng (rangepartition).
  - Thực hiện phân mảnh theo round-robin (roundrobinpartition).
  - Chèn dữ liệu vào phân mảnh theo khoảng (rangeinsert).
  - Chèn dữ liệu vào phân mảnh round-robin (roundrobininsert). Cách tiếp cận này giúp mã nguồn trở nên rõ ràng, dễ hiểu, dễ bảo trì và thuận tiện cho việc kiểm thử từng phần.
- Tuân thủ yêu cầu đề bài: Trong suốt quá trình triển khai, nhóm đã đặc biệt chú trọng đến việc tuân thủ các ràng buộc và yêu cầu kỹ thuật được nêu trong đề bài, như không đóng kết nối CSDL trong các hàm chính, không mã hóa cứng các thông tin nhạy cảm (tên tệp, tên CSDL), sử dụng đúng các tiền tố tên bảng đã quy định, và đảm bảo logic của các thuật toán phân mảnh và chèn dữ liệu phải chính xác, nhất quán với mô tả và các ví dụ minh họa.

## **3. Nền tảng lý thuyết các phương pháp phân mảnh:**

Phân mảnh ngang (Horizontal Partitioning) là một kỹ thuật thiết kế cơ sở dữ liệu nhằm chia các hàng của một bảng lớn thành nhiều bảng nhỏ hơn, gọi là các phân mảnh (partitions). Mỗi phân mảnh có cùng schema với bảng gốc nhưng



chỉ chứa một tập con dữ liệu. Điều này giúp cải thiện hiệu năng truy vấn, khả năng quản lý và khả năng mở rộng của hệ thống.

### *3.1 Phân mảnh theo khoảng (range partitioning)*

**Nguyên tắc:** Phân mảnh theo khoảng chia dữ liệu dựa trên các khoảng giá trị liên tục của một cột được chọn (gọi là khóa phân mảnh - partition key). Mỗi phân mảnh được gán một khoảng giá trị, và tất cả các hàng có giá trị khóa phân mảnh nằm trong khoảng đó sẽ được lưu trữ trong phân mảnh tương ứng. Trong bài tập này, cột Rating được sử dụng làm khóa phân mảnh.

#### **Ưu điểm:**

- **Tối ưu hóa truy vấn theo khoảng:** Đây là ưu điểm lớn nhất. Khi một truy vấn chứa điều kiện lọc trên khóa phân mảnh (ví dụ: `SELECT * FROM Ratings WHERE Rating > 4.0`), trình tối ưu hóa truy vấn của CSDL có thể nhanh chóng xác định chỉ những phân mảnh chứa dữ liệu trong khoảng đó cần được quét. Quá trình này được gọi là "partition pruning" (loại bỏ phân mảnh), giúp giảm đáng kể lượng dữ liệu cần đọc và cải thiện hiệu năng truy vấn một cách rõ rệt.
- **Quản lý dữ liệu theo thời gian/giá trị:** Rất hữu ích cho việc quản lý dữ liệu theo chuỗi thời gian (ví dụ, phân mảnh theo tháng hoặc năm) hoặc các dữ liệu có tính thứ tự. Việc xóa dữ liệu cũ trở nên rất hiệu quả: chỉ cần xóa toàn bộ phân mảnh (ví dụ: `DROP PARTITION`) thay vì thực hiện một lệnh `DELETE` tốn kém trên hàng triệu bản ghi.

#### **Nhược điểm:**

- **Nguy cơ tạo ra "điểm nóng" (Hotspots):** Nếu dữ liệu không được phân phối đều trên các khoảng giá trị, một số phân mảnh có thể trở nên lớn hơn và chịu tải truy cập nhiều hơn đáng kể so với các phân mảnh khác. Ví dụ, trong bộ dữ liệu đánh giá phim, phần lớn các đánh giá có thể tập trung ở khoảng 3 đến 4 sao, tạo ra một "điểm nóng" tại phân mảnh tương ứng, trong khi các phân mảnh cho 1 sao hoặc 5 sao có thể ít được sử dụng hơn.

**Khó khăn trong việc chọn khóa phân mảnh:** Việc chọn một khóa phân mảnh tốt, giúp phân phối dữ liệu đều và phù hợp với các mẫu truy vấn phổ biến, là một thách thức.

### *3.2 Phân mảnh theo Round-Robin (mô phỏng ở tầng ứng dụng)*

**Nguyên tắc:** Phân mảnh theo Round-Robin phân phối các hàng một cách tuần tự và lần lượt cho từng phân mảnh. Hàng đầu tiên vào phân mảnh 0, hàng thứ hai

vào phân mảnh 1, ..., hàng thứ N vào phân mảnh N-1, và hàng thứ N+1 lại quay về phân mảnh 0. Trong thực tế, các hệ quản trị CSDL như MySQL không cung cấp loại phân mảnh "Round-Robin" một cách trực tiếp. Thay vào đó, nó thường được mô phỏng ở tầng ứng dụng (như trong bài tập này) hoặc có thể đạt được hiệu quả tương tự bằng Phân mảnh theo HASH (HASH Partitioning). Trong phân mảnh theo HASH, một hàm băm được áp dụng trên giá trị của khóa phân mảnh để xác định phân mảnh mà bản ghi sẽ thuộc về. Mục tiêu của HASH cũng là phân phối dữ liệu một cách đồng đều.

#### Ưu điểm:

- **Phân phối dữ liệu và tải ghi (write load) đồng đều:** Đây là ưu điểm lớn nhất của phương pháp này. Vì các bản ghi mới được chèn lần lượt vào các phân mảnh, tải ghi được trải đều trên tất cả các phân mảnh. Điều này giúp tránh hoàn toàn vấn đề "điểm nóng" và rất hữu ích cho các ứng dụng có lượng ghi lớn và liên tục.
- **Đơn giản:** Logic để xác định phân mảnh đích rất đơn giản và không phụ thuộc vào giá trị dữ liệu của bản ghi.

#### Nhược điểm:

- **Truy vấn theo khoảng hoặc theo thuộc tính rất kém hiệu quả:** Vì các hàng có giá trị liên quan (ví dụ: tất cả các phim có Rating là 5) bị phân tán ngẫu nhiên trên tất cả các phân mảnh, không có cách nào để trình tối ưu hóa thực hiện "partition pruning". Bất kỳ truy vấn nào không dựa trên khóa chính (và trong trường hợp này, không có khóa nào giúp định vị phân mảnh) đều phải quét qua **tất cả** các phân mảnh. Điều này làm cho các truy vấn đọc trở nên rất chậm và tốn kém.
- **Khó khăn trong việc truy xuất dữ liệu liên quan:** Việc lấy tất cả dữ liệu liên quan đến một thực thể (ví dụ: tất cả các đánh giá của một người dùng) trở nên phức tạp và yêu cầu tổng hợp kết quả từ nhiều phân mảnh.

#### 3.3 Bảng so sánh tổng hợp:

Tiêu chí	Range partitioning	Round-robin partitioning
Nguyên tắc phân phối	Dựa trên khoảng giá trị của khoá phân mảnh (Rating)	Phân phối tuần tự lần lượt cho các phân mảnh

Phân bố dữ liệu	Có thể không đồng đều, phụ thuộc vào phân phối giá trị thực tế.	Rất đồng đều
Hiệu năng truy vấn theo khoảng	Rất cao, hỗ trợ “partition pruning”	Rất thấp, phải quét tất cả các phân mảnh.
Hiệu năng ghi (insert)	Có thể tạo “điểm nóng” (hotspot) nếu ghi tập trung	Rất cao, tải ghi được phân phối đồng đều.
Trường hợp sử dụng	Dữ liệu có tính thứ tự (thời gian, giá trị). Các truy vấn thường xuyên lọc theo khoảng.	Hệ thống có tải ghi (write-heavy) rất lớn, cần phân phối tải đều và không yêu cầu truy vấn đọc phức tạp.
Nhược điểm	Nguy cơ “điểm nóng”, khó chọn khoá phân mảnh tốt.	Truy vấn theo thuộc tính, khoảng rất kém hiệu quả.

#### 4. Tổng quan chiến lược giải quyết các nhiệm vụ chính

Dưới đây là mô tả chi tiết về thuật toán và logic cốt lõi được áp dụng để giải quyết các nhiệm vụ trong đề tài:

##### 4.1 Tải dữ liệu (load ratings):

Mục đích: đọc dữ liệu từ tệp **ratings.dat** và lưu trữ vào bảng **Ratings** trong PostgreSQL.

Trước khi tải dữ liệu, hàm kiểm tra và tạo bảng Ratings nếu nó chưa tồn tại. Schema của bảng này được định nghĩa gồm ba cột:

- UserID (INTEGER)
- MovieID (INTEGER)
- Rating (FLOAT).

Quá trình này bao gồm việc đọc file **ratings.dat** gốc, tách các trường thông tin dựa trên dấu phân cách `::`, và chỉ ghi ba trường đầu tiên (UserID, MovieID, Rating) vào file tạm với định dạng tab-separated.

Sau đó, hàm sử dụng phương thức `copy_from()` của PostgreSQL để load dữ liệu từ file tạm vào bảng một cách nhanh chóng và hiệu quả.

Để tối ưu hóa hiệu suất truy vấn trong tương lai, hàm tự động tạo index trên cột Rating. Toàn bộ quá trình được thực hiện trong một transaction với exception handling để đảm bảo tính nhất quán dữ liệu.

#### 4.2 Phân mảnh theo khoảng (range partition)

Mục đích: chia bảng **Ratings** thành N phân mảnh, với các bản ghi được phân phối vào các phân mảnh dựa trên việc giá trị Rating thuộc vào khoảng nào trong N khoảng giá trị đồng đều được chia ra từ thang điểm 0 đến 5.

- B1: Hàm xóa tất cả các bảng phân mảnh theo khoảng đã tồn tại trước đó (có tên dạng **range\_partX**).
- B2: Hàm tính toán **step** để xác định độ rộng của mỗi khoảng giá trị Rating.

$$step = \frac{5.0}{N}$$

- B3: Quá trình phân phối dữ liệu được thực hiện thông qua các câu lệnh **INSERT INTO ... SELECT** có điều kiện **WHERE** cụ thể. Đối với phân mảnh đầu tiên ( $i = 0$ ), điều kiện là **Rating >= lower AND Rating <= upper** (bao gồm cả cận dưới và cận trên). Đối với các phân mảnh còn lại ( $i > 0$ ), điều kiện là **Rating > lower AND Rating <= upper** (chỉ bao gồm cận trên để tránh trùng lặp dữ liệu).

#### 4.3 Phân mảnh theo Round-robin (round robin partition)

Mục đích: phân phối đều các bản ghi từ bảng Ratings vào N phân mảnh được tạo theo kiểu vòng tròn, đảm bảo phân phối cân bằng dữ liệu.

- B1: Xóa các bảng phân mảnh **round-robin** đã tồn tại và tạo N bảng phân mảnh mới với cấu trúc giống bảng **Ratings**.
- B2: Sử dụng ``ctid`` (tuple identifier) của PostgreSQL kết hợp với hàm ``ROW_NUMBER() OVER (ORDER BY ctid)`` để tạo ra một chỉ số thứ tự nhất quán cho mỗi bản ghi:

$$partition_{id} = (row_{number} - 1) \% number\_of\_partitions$$

Phương pháp này đảm bảo phân phối round-robin chính xác và hiệu quả, với mỗi phân mảnh nhận được số lượng bản ghi gần như bằng nhau.

#### 4.4 Chèn dữ liệu vào phân mảnh theo khoảng (range insert)

Mục đích: chèn một bản ghi đánh giá mới vào bảng **Ratings** gốc và đồng thời vào đúng phân mảnh **range\_partX** dựa trên giá trị Rating của bản ghi.

- Hàm bắt đầu bằng việc validate đầu vào thông qua hàm **validate\_rating()** để đảm bảo giá trị Rating nằm trong khoảng hợp lệ [0, 5].
- Hàm sử dụng metadata thông qua **get\_partition\_count()** để kiểm tra số lượng phân mảnh theo khoảng hiện có. Nếu chưa có phân mảnh nào, hàm sẽ tự động gọi **rangepartition()** với 5 phân mảnh mặc định.
- Logic xác định phân mảnh đích được cải tiến với công thức:

$$partition_{index} = int(\frac{rating}{step})$$

#### 4.5 Chèn dữ liệu vào phân mảnh Round-robin (round robin insert)

Mục đích: chèn một bản ghi đánh giá mới vào bảng **Ratings** gốc và vào đúng phân mảnh **rrobin\_partX** theo logic phân phối round-robin.

- Hàm validate đầu vào và sử dụng metadata để kiểm tra số lượng phân mảnh round-robin hiện có.
- Để xác định phân mảnh đích, hàm thực hiện truy vấn để lấy tổng số bản ghi hiện có trong bảng Ratings gốc (**total\_rows**). Chỉ số của phân mảnh đích được tính theo công thức:

$$next_{partition} = total_{rows} \% num_{partitions}$$

### 5. Cách thiết kế và xử lý yêu cầu

Trong quá trình triển khai, nhóm đã đưa ra một số quyết định thiết kế quan trọng để đáp ứng các yêu cầu cụ thể:

- Sử dụng **PostgreSQL** thay vì MySQL mang lại nhiều lợi ích về hiệu suất và tính năng. PostgreSQL cung cấp các tính năng đặc biệt như **`ctid`** và **`copy\_from()`** giúp tối ưu hóa đáng kể hiệu suất xử lý dữ liệu lớn.
- Xử lý hiệu quả dữ liệu lớn: Việc sử dụng **`copy\_from()`** thay vì INSERT từng bản ghi giúp tăng tốc độ load dữ liệu lên hàng chục lần. Phương pháp tạo file tạm thời đảm bảo tuân thủ yêu cầu không sửa đổi file gốc.
- Sử dụng metadata thông minh: Hàm **get\_partition\_count()** được thiết kế như một metadata helper, cho phép các hàm insert tự động phát hiện và thích ứng với số lượng phân mảnh hiện có mà không cần biến toàn cục.
- Tự động tạo phân mảnh khi chèn: Các hàm **rangeinsert** và **roundrobininsert** được thiết kế với logic tự động gọi các hàm tạo phân mảnh tương ứng nếu phát hiện chưa có phân mảnh nào.

Ngoài ra, nhóm cũng đã tuân thủ các ràng buộc kỹ thuật nghiêm ngặt:

- Quản lý kết nối CSDL được thực hiện một cách nhất quán: tất cả các hàm chính chỉ sử dụng **cursor** và không đóng đối tượng **openconnection**, tuân thủ hoàn toàn yêu cầu "Không được đóng kết nối bên trong các hàm đã triển khai."
- Sử dụng hằng số cục bộ thay vì biến toàn cục đảm bảo tính modular và dễ bảo trì. Các tiền tố bảng được sử dụng chính xác theo yêu cầu: 'range\_part' và 'rrobin\_part'.
- Validation và Error Handling nâng cao: **Hàm validate\_rating()** đảm bảo tính hợp lệ của dữ liệu đầu vào. Tất cả các thao tác quan trọng đều được bọc trong try-catch với transaction management để đảm bảo tính nhất quán dữ liệu.

## 6. Cấu trúc dữ liệu sử dụng:

Hệ thống được thiết kế với các cấu trúc dữ liệu tối ưu cho PostgreSQL. Việc sử dụng ``tempfile.NamedTemporaryFile`` đảm bảo xử lý an toàn file tạm thời. Index được tự động tạo trên cột Rating để tối ưu hóa hiệu suất truy vấn phân mảnh.

Các hàm được thiết kế modular với separation of concerns rõ ràng. Logic metadata được tách riêng, validation được centralized, và error handling được standardized across tất cả các hàm.

## 7. Thách thức gặp phải và giải pháp:

Trong quá trình thực hiện đề tài, nhóm đã gặp phải một số khó khăn ban đầu cũng như là các giải pháp sau đã được đề xuất để xử lý hiệu quả.

### 7.1 Hiệu suất:

- Khó khăn: Việc xử lý 10 triệu bản ghi gặp khó khăn về tốc độ và bộ nhớ khi sử dụng phương pháp INSERT truyền thống.
- Giải pháp: Chuyển sang sử dụng PostgreSQL's ``copy_from()`` method kết hợp với file tạm thời đã giải quyết hoàn toàn vấn đề này. Tốc độ load dữ liệu được cải thiện từ hàng giờ xuống còn vài phút.

### 7.2 Tính nhất quán trong Round-Robin:

- Khó khăn: Việc đảm bảo phân phối round-robin chính xác và có thể lặp lại là một thách thức kỹ thuật.

- Giải pháp: Sử dụng `ctid` của PostgreSQL kết hợp với `ROW\_NUMBER()` đảm bảo thứ tự xử lý nhất quán và có thể dự đoán được, giải quyết hoàn toàn vấn đề này.

### *7.3 Quản lý metadata:*

- Khó khăn: cần theo dõi số lượng phân mảnh mà không sử dụng biến toàn cục.
- Giải pháp: Phát triển hàm **get\_partition\_count()** sử dụng `information_schema` của PostgreSQL để query metadata một cách động và chính xác.

## CHƯƠNG II: PHÂN TÍCH CHI TIẾT

### 1. Mục đích và phạm vi:

Chương này trình bày chi tiết về mục đích, nguyên lý hoạt động, mã nguồn Python và đánh giá việc tuân thủ yêu cầu đề bài cho từng hàm được triển khai trong hệ thống mô phỏng phân mảnh dữ liệu. Hệ thống được thiết kế để thực hiện các phương pháp phân mảnh ngang khác nhau trên cơ sở dữ liệu PostgreSQL, bao gồm phân mảnh theo khoảng (range partitioning) và phân mảnh kiểu vòng tròn (round-robin partitioning).

### 2. Các hằng số và cấu trúc cơ sở dữ liệu:

#### 2.1 Cấu trúc cơ sở dữ liệu:

##### a. Bảng chính: **ratings**

Đây là bảng trung tâm, chứa toàn bộ dữ liệu đánh giá phim được tải từ tệp ratings.dat.

- Tên bảng: ratings
- Mục đích: Lưu trữ tất cả các bản ghi đánh giá, đóng vai trò là nguồn dữ liệu cho các quá trình phân mảnh và là nơi các bản ghi mới được chèn vào đầu tiên.
- Cấu trúc:

Tên cột	Kiểu dữ liệu	Mô tả
userid	INT	Mã định danh của người dùng
movieid	INT	Mã định danh của bộ phim
rating	FLOAT	Điểm đánh giá mà người dùng dành cho bộ phim (từ 0.0 đến 5.0).

##### b. Các bảng phân mảnh (partitions):

Các bảng phân mảnh được tạo ra để chia nhỏ dữ liệu từ bảng ratings. Tất cả các bảng phân mảnh đều có cấu trúc giống hệt bảng ratings.

- Phân mảnh theo khoảng (Range Partitioning):
  - Quy ước đặt tên: range\_part0, range\_part1, ..., range\_partN-1 (với N là số phân mảnh).



- Mục đích: Mỗi bảng chứa một tập con các bản ghi từ bảng ratings có giá trị rating nằm trong một khoảng xác định.
- Phân mảnh theo Round-Robin:
  - Quy ước đặt tên: rrobin\_part0, rrobin\_part1, ..., rrobin\_partN-1.
  - Mục đích: Mỗi bảng chứa một tập con các bản ghi từ bảng ratings được phân phối lần lượt, tuần tự.

## 2.2 Các hằng số:

Hệ thống sử dụng phương pháp định nghĩa các hằng số cục bộ thay vì biến toàn cục để đảm bảo tính mô-đun hóa. Cách tiếp cận này giúp mỗi hàm trở nên độc lập, không phụ thuộc vào trạng thái toàn cục, từ đó tăng tính ổn định và khả năng bảo trì của mã nguồn.

Các thư viện quan trọng được sử dụng:

- psycopg2: kết nối và tương tác với PostgreSQL.
- os: thao tác hệ thống tệp, xóa các tệp tạm thời.
- math: thực hiện các phép toán cho thuật toán phân mảnh.
- tempfile: tạo và quản lý các tệp tạm thời cho xử lý dữ liệu.

```
import psycopg2
import os
import math
import tempfile
```

Hình 2. Thư viện sử dụng

Các hằng số cục bộ chính:

- RANGE\_TABLE\_PREFIX: tiền tố cho bảng phân mảnh theo khoảng (range\_part).
- RROBIN\_TABLE\_PREFIX: tiền tố cho bảng phân mảnh round-robin (rrobin\_part).
- USER\_ID\_COLNAME: tên cột lưu trữ ID người dùng (userid).
- MOVIE\_ID\_COLNAME: tên cột lưu trữ ID phim (movieid).

- RATING\_COLNAME: tên cột lưu trữ điểm đánh giá (rating).

```
RANGE_TABLE_PREFIX = 'range_part'
RROBIN_TABLE_PREFIX = 'rrobin_part'
USER_ID_COLNAME = 'userid'
MOVIE_ID_COLNAME = 'movieid'
RATING_COLNAME = 'rating'
```

Hình 3. Các hằng số cục bộ chính

### 3. Các hàm tiện ích

#### 3.1 Hàm *getopenconnection(user, password, dbname)*

Mục đích: Thiết lập và trả về kết nối đến cơ sở dữ liệu PostgreSQL.

Nguyên lý hoạt động: Tạo một kết nối đến máy chủ PostgreSQL dựa trên thông tin xác thực được cung cấp, gồm các bước:

- Xây dựng chuỗi kết nối (connection string) chứa tên cơ sở dữ liệu, tên người dùng, địa chỉ máy chủ và mật khẩu.
- Gọi phương thức `psycopg2.connect()` để thiết lập kết nối TCP đến máy chủ PostgreSQL.
- PostgreSQL thực hiện xác thực người dùng dựa trên thông tin được cung cấp.
- Sau khi xác thực thành công, một phiên kết nối được thiết lập và trả về dưới dạng đối tượng `connection`.

```
def getopenconnection(user='postgres', password='1', dbname='movie_rating'):
    """
    Tạo kết nối đến PostgreSQL database
    Args:
        user: Tên user database
        password: Mật khẩu
        dbname: Tên database
    Returns:
        Connection object
    """
    return psycopg2.connect("dbname='" + dbname + "' user='" + user + "' host='localhost' password='" + password + "'")
```

Hình 4. Hàm kết nối CSDL

#### 3.2 Hàm *create\_db(dbname):*

Mục đích: Tạo cơ sở dữ liệu mới nếu chưa tồn tại, sử dụng cơ sở dữ liệu hệ thống của PostgreSQL để thực hiện thao tác này.

Nguyên lý hoạt động: Hàm thực hiện quy trình kiểm tra và tạo cơ sở dữ liệu theo các bước sau:

- Kết nối đến cơ sở dữ liệu hệ thống "postgres", nơi lưu trữ thông tin meta về tất cả các cơ sở dữ liệu.
- Thiết lập chế độ **AUTOCOMMIT** để có thể thực thi lệnh DDL (CREATE DATABASE) mà không cần transaction.
- Kiểm tra sự tồn tại của cơ sở dữ liệu bằng cách truy vấn bảng **pg\_catalog.pg\_database**.
- Nếu cơ sở dữ liệu chưa tồn tại (count = 0), thực hiện lệnh **CREATE DATABASE**.
- Nếu đã tồn tại, hiển thị thông báo phù hợp.
- Đóng cursor và kết nối đến cơ sở dữ liệu hệ thống sau khi hoàn tất.

```
def create_db(dbname):
    """
    Tạo database mới nếu chưa tồn tại
    Args:
        dbname: Tên database cần tạo
    """
    con = getopenconnection(dbname='postgres')
    con.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    cur = con.cursor()

    # Kiểm tra xem database đã tồn tại chưa
    cur.execute('SELECT COUNT(*) FROM pg_catalog.pg_database WHERE datname=\'%s\' % (dbname,))
    count = cur.fetchone()[0]
    if count == 0:
        cur.execute('CREATE DATABASE %s' % (dbname,))
    else:
        print('A database named {0} already exists'.format(dbname))

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

Hình 5. Hàm tạo CSDL

### 3.3 Hàm *get\_partition\_count(prefix, openconnection)*

Mục đích: Đếm số lượng bảng phân mảnh hiện có dựa trên tiền tố tên bảng. Hàm này đóng vai trò là trợ giúp metadata, giúp các hàm khác xác định số lượng phân mảnh hiện tại mà không cần sử dụng biến toàn cục.

Nguyên lý hoạt động: Hàm sử dụng các bước sau để xác định số lượng phân mảnh:

- Tạo cursor từ kết nối được cung cấp.
- Thực thi truy vấn đến `information_schema.tables`, là nơi PostgreSQL lưu trữ metadata về tất cả các bảng.
- Sử dụng toán tử **SIMILAR TO** với biểu thức chính quy để tìm các bảng có tên bắt đầu bằng tiền tố được cung cấp và theo sau là ít nhất một chữ số.
- Áp dụng **COUNT** để đếm số lượng bảng thỏa mãn điều kiện.
- Trả về số lượng này để các hàm gọi sử dụng.

```
def get_partition_count(prefix, openconnection):
    """
    Đếm số lượng partition hiện có dựa trên prefix - sử dụng như metadata
    Args:
        prefix: Tiền tố của tên bảng partition
        openconnection: Kết nối database
    Returns:
        Số lượng partition
    """
    cur = openconnection.cursor()
    cur.execute(f"""
        SELECT COUNT(table_name) FROM information_schema.tables
        WHERE table_schema = 'public'
        AND table_name SIMILAR TO '{prefix}[0-9]+'
    """)
    count = cur.fetchone()[0]
    # Không đóng cursor - để connection quản lý
    return count
```

Hình 6. Hàm *get\_partition\_count*

### 3.4 Hàm *validate\_rating(rating)*

Mục đích: Kiểm tra tính hợp lệ của giá trị đánh giá đầu vào, đảm bảo tính toàn vẹn dữ liệu trước khi chèn vào cơ sở dữ liệu.

Nguyên lý hoạt động: Hàm thực hiện hai kiểm tra quan trọng:

- Kiểm tra kiểu dữ liệu: Sử dụng hàm **isinstance()** để xác minh rating là số nguyên hoặc số thực (int hoặc float).
- Kiểm tra phạm vi giá trị: Đảm bảo rating nằm trong khoảng hợp lệ (0-5).
- Chuẩn hóa kiểu dữ liệu: Chuyển đổi giá trị thành kiểu float trước khi trả về, đảm bảo tính nhất quán.
- Xử lý lỗi: Nếu dữ liệu không hợp lệ, nâng ngoại lệ ValueError với thông báo cụ thể.

```
def validate_rating(rating):
    """
    Kiểm tra tính hợp lệ của giá trị rating
    Args:
        rating: Giá trị rating cần kiểm tra
    Returns:
        Giá trị rating đã được validate
    Raises:
        ValueError: Nếu rating không hợp lệ
    """
    if not isinstance(rating, (int, float)):
        raise ValueError("Rating must be a number")
    if rating < 0 or rating > 5:
        raise ValueError("Rating must be between 0 and 5")
    return float(rating)
```

Hình 7. Hàm *validate\_rating*

## 4. Phân tích các hàm chính

### 4.1 Hàm *loadratings*

Mục đích: Đọc dữ liệu từ tệp **ratings.dat** và lưu trữ vào bảng trong PostgreSQL với hiệu suất cao và tuân thủ yêu cầu không sửa đổi tệp gốc.

Nguyên lý hoạt động: Hàm triển khai quy trình tải dữ liệu tối ưu qua các bước sau:

- Tạo cấu trúc bảng: Kiểm tra và tạo bảng đích nếu chưa tồn tại, với ba cột: `userid` (INTEGER), `movieid` (INTEGER), và `rating` (FLOAT)
- Xử lý dữ liệu qua tệp tạm:
  - Tạo tệp tạm với `tempfile.NamedTemporaryFile()` và `mode='w'` để ghi văn bản.
  - Đọc từng dòng của tệp **ratings.dat** gốc.
  - Phân tách mỗi dòng bằng dấu phân cách ":" để trích xuất các trường dữ liệu.
  - Chỉ lấy ba trường đầu tiên (`UserID`, `MovieID`, `Rating`) và ghi vào tệp tạm với định dạng tab-separated.
  - Sử dụng `delete=False` để có thể đóng tệp trước khi đọc lại.
- Tải dữ liệu vào PostgreSQL: Sử dụng phương thức `copy_from()` của PostgreSQL để thực hiện bulk loading.
- Dọn dẹp tài nguyên: Xóa tệp tạm sau khi tải xong để tiết kiệm không gian và đảm bảo bảo mật.
- Tối ưu hóa hiệu suất truy vấn: Tạo chỉ mục trên cột `rating` để cải thiện hiệu suất của các truy vấn trong tương lai.
- Quản lý giao dịch: Toàn bộ quá trình được thực hiện trong một giao dịch với xử lý ngoại lệ để đảm bảo tính nhất quán.

```

def loadratings(ratingtablename, ratingsfilepath, openconnection):
    cur = openconnection.cursor()
    # Định nghĩa các hằng số cục bộ thay vì toàn cục
    USER_ID_COLNAME = 'userid'
    MOVIE_ID_COLNAME = 'movieid'
    RATING_COLNAME = 'rating'
    # Tạo bảng với lược đồ đúng yêu cầu
    cur.execute(f"""
        CREATE TABLE IF NOT EXISTS {ratingtablename} (
            {USER_ID_COLNAME} INTEGER,
            {MOVIE_ID_COLNAME} INTEGER,
            {RATING_COLNAME} FLOAT
        );
    """)
    try:
        # Tạo file tạm thời để xử lý - không sửa đổi file gốc
        with tempfile.NamedTemporaryFile(mode='w', delete=False) as temp_file:
            with open(ratingsfilepath, 'r') as input_file:
                for line in input_file:
                    parts = line.strip().split("::")
                    if len(parts) == 4:
                        # Chỉ lấy 3 cột cần thiết
                        temp_file.write(f"{parts[0]}\t{parts[1]}\t{parts[2]}\n")
        # Load dữ liệu từ file tạm vào bảng
        with open(temp_file.name, 'r') as temp_file:
            cur.copy_from(temp_file, ratingtablename, sep='\t')
        # Xóa file tạm
        os.unlink(temp_file.name)
        # Tạo index cho cột rating để tối ưu performance
        cur.execute(f"""
            CREATE INDEX IF NOT EXISTS idx_{ratingtablename}_rating
            ON {ratingtablename} ({RATING_COLNAME})
        """)
        openconnection.commit()
    except Exception as e:
        print(f"Lỗi khi load dữ liệu: {e}")
        openconnection.rollback()
        raise
    # Không đóng cursor - để connection quản lý

```

Hình 8. Hàm load\_ratings

## 4.2 Hàm rangepartition

Mục đích: Phân mảnh bảng ratings thành N phân mảnh dựa trên khoảng giá trị đánh giá, đảm bảo mỗi bản ghi thuộc đúng một phân mảnh và phân phối theo khoảng giá trị đồng đều.

Nguyên lý hoạt động: Hàm triển khai phương pháp phân mảnh theo khoảng với các bước chi tiết:

- Tạo môi trường sạch: Xóa tất cả các bảng phân mảnh theo khoảng đã tồn tại để đảm bảo không có xung đột hoặc dữ liệu cũ.
- Tính toán kích thước khoảng: Sử dụng công thức  $\text{step} = 5.0 / \text{numberofpartitions}$  để xác định độ rộng của mỗi khoảng trong thang đánh giá từ 0 đến 5. Ví dụ với 5 phân mảnh:
  - Phân mảnh 0:  $0 \leq \text{rating} \leq 1$
  - Phân mảnh 1:  $1 < \text{rating} \leq 2$
  - Phân mảnh 2:  $2 < \text{rating} \leq 3$

- Phân mảnh 3:  $3 < \text{rating} \leq 4$
- Phân mảnh 4:  $4 < \text{rating} \leq 5$
- Tạo cấu trúc cho tất cả phân mảnh: Tạo N bảng phân mảnh trước khi phân phối dữ liệu, đảm bảo mỗi bảng có cùng cấu trúc với bảng gốc.
- Phân phối dữ liệu với xử lý biên khoảng đặc biệt:
  - Đối với phân mảnh đầu tiên ( $i = 0$ ): Sử dụng điều kiện WHERE  $\text{Rating} \geq \text{lower}$  AND  $\text{Rating} \leq \text{upper}$  để bao gồm cả giới hạn dưới
  - Đối với các phân mảnh còn lại ( $i > 0$ ): Sử dụng điều kiện WHERE  $\text{Rating} > \text{lower}$  AND  $\text{Rating} \leq \text{upper}$  để loại trừ giới hạn dưới.

Cách xử lý này đảm bảo mỗi giá trị rating chỉ thuộc về đúng một phân mảnh, ngăn chặn trùng lặp dữ liệu và đảm bảo tính toàn vẹn.

- Thực hiện trong một giao dịch: Toàn bộ quá trình được bao bọc trong một giao dịch với cơ chế rollback trong trường hợp lỗi.

```
def rangepartition(ratingtablename, numberofpartitions, openconnection):
    cur = openconnection.cursor()
    # Định nghĩa các hằng số cục bộ - không sử dụng tiền tố khác
    RANGE_TABLE_PREFIX = 'range_part' # Không thay đổi tiền tố đã cho
    USER_ID_COLNAME = 'userid'
    MOVIE_ID_COLNAME = 'movieid'
    RATING_COLNAME = 'rating'
    # Xóa các bảng phân mảnh hiện có - bắt đầu từ 0
    for i in range(numberofpartitions):
        cur.execute(f"DROP TABLE IF EXISTS {RANGE_TABLE_PREFIX}{i}")
    # Tính khoảng cho phân mảnh
    step = 5.0 / numberofpartitions
    # Tạo tất cả các bảng phân mảnh trước - lược đồ đúng yêu cầu
    for i in range(numberofpartitions):
        cur.execute(f"""
            CREATE TABLE {RANGE_TABLE_PREFIX}{i} (
                {USER_ID_COLNAME} INTEGER,
                {MOVIE_ID_COLNAME} INTEGER,
                {RATING_COLNAME} FLOAT
            )
        """)
    try:
        # Phân phối dữ liệu vào các bảng phân mảnh
        for i in range(numberofpartitions):
            lower = i * step
            upper = (i + 1) * step

            if i == 0:
                # Partition đầu tiên bao gồm cả giá trị lower bound
                cur.execute(f"""
                    INSERT INTO {RANGE_TABLE_PREFIX}{i}
                    SELECT {USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME}
                    FROM {ratingtablename}
                    WHERE {RATING_COLNAME} >= {lower} AND {RATING_COLNAME} <= {upper}
                """)
            else:
                # Các partition khác chỉ lấy giá trị lớn hơn lower bound
                cur.execute(f"""
                    INSERT INTO {RANGE_TABLE_PREFIX}{i}
                    SELECT {USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME}
                    FROM {ratingtablename}
                    WHERE {RATING_COLNAME} > {lower} AND {RATING_COLNAME} <= {upper}
                """)

        openconnection.commit()
    except Exception as e:
        openconnection.rollback()
        raise
    # Không đóng cursor - để connection quản lý
```

Hình 9. Hàm range\_partition

### 4.3 Hàm *roundrobinpartition*

Mục đích: Phân mảnh bảng ratings thành N phân mảnh theo phương pháp vòng tròn (round-robin), đảm bảo phân phối đều dữ liệu giữa các phân mảnh.

Nguyên lý hoạt động: Hàm triển khai phương pháp phân mảnh round-robin với các bước chi tiết:

- Tạo môi trường sạch: Tương tự như *rangepartition*, xóa tất cả các phân mảnh round-robin đã tồn tại.
- Tạo cấu trúc cho tất cả phân mảnh: Tạo N bảng phân mảnh với cùng cấu trúc như bảng gốc.
- Phân phối dữ liệu sử dụng *ctid* và *ROW\_NUMBER()*:
  - *ctid*: Là cột hệ thống trong PostgreSQL đại diện cho vị trí vật lý của bản ghi (block number, tuple index). Mỗi hàng có một *ctid* duy nhất và ổn định.
  - *ROW\_NUMBER() OVER (ORDER BY ctid)*: Hàm của số tạo số thứ tự cho mỗi hàng, sắp xếp theo *ctid* để đảm bảo thứ tự nhất quán.
- Thuật toán phân phối: Sử dụng phép toán modulo để xác định phân mảnh đích cho mỗi bản ghi.

$$partition_{id} = (row_{number} - 1) \% number\_of\_partitions$$

- Bản ghi 1 → Phân mảnh 0
- Bản ghi 2 → Phân mảnh 1
- Bản ghi N → Phân mảnh (N-1) % numberofpartitions
- Bản ghi N+1 → Phân mảnh N % numberofpartitions
- Thực thi server-side: Sử dụng nested query để tính toán *partition\_id* cho mỗi hàng và sau đó chọn những hàng thuộc về phân mảnh cụ thể, tất cả được thực hiện bởi PostgreSQL server mà không cần chuyển dữ liệu qua mạng
- Quản lý giao dịch: Toàn bộ quá trình được thực hiện trong một giao dịch để đảm bảo tính nhất quán



```

def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
    cur = openconnection.cursor()
    # Định nghĩa các hằng số cục bộ - không thay đổi tiền tố đã cho
    RROBIN_TABLE_PREFIX = 'rrobin_part'
    USER_ID_COLNAME = 'userid'
    MOVIE_ID_COLNAME = 'movieid'
    RATING_COLNAME = 'rating'
    # Xóa các phân mảnh hiện có - bắt đầu từ 0
    for i in range(numberofpartitions):
        cur.execute(f"DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i}")
    # Tạo tất cả các bảng phân mảnh trước - lược đồ đúng yêu cầu
    for i in range(numberofpartitions):
        cur.execute(f"""
            CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (
                {USER_ID_COLNAME} INTEGER,
                {MOVIE_ID_COLNAME} INTEGER,
                {RATING_COLNAME} FLOAT
            )
        """)
    try:
        # Phân phối dữ liệu theo round robin sử dụng ctid
        for i in range(numberofpartitions):
            cur.execute(f"""
                INSERT INTO {RROBIN_TABLE_PREFIX}{i}
                SELECT {USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME}
                FROM (
                    SELECT {USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME},
                           (ROW_NUMBER() OVER (ORDER BY ctid) - 1) % {numberofpartitions} as partition_id
                    FROM {ratingtablename}
                ) as distributed_data
                WHERE partition_id = {i}
            """)
        openconnection.commit()
    except Exception as e:
        openconnection.rollback()
        raise
    # Không đóng cursor - để connection quản lý

```

Hình 10. Hàm round robin partition

#### 4.4 Hàm range insert

Mục đích: Chèn một bản ghi đánh giá mới vào bảng gốc và đồng thời vào phân mảnh theo khoảng tương ứng, với khả năng tự động duy trì cấu trúc phân mảnh.

Nguyên lý hoạt động: Hàm triển khai quy trình chèn dữ liệu thông minh với các bước:

- Xác thực đầu vào: Gọi hàm validate\_rating() để đảm bảo giá trị rating hợp lệ và chuẩn hóa thành kiểu float.
- Kiểm tra và tạo phân mảnh nếu cần:
  - Sử dụng get\_partition\_count() để kiểm tra số lượng phân mảnh hiện có.
  - Nếu chưa có phân mảnh nào (num\_partitions = 0), tự động gọi rangepartition() để tạo 5 phân mảnh mặc định.
  - Chức năng này đáp ứng yêu cầu "duy trì các bảng trong cơ sở dữ liệu khi có thao tác chèn".
- Xác định phân mảnh đích: Sử dụng thuật toán phức tạp để đảm bảo sự nhất quán với hàm rangepartition:
  - Tính kích thước khoảng:  $step = 5.0 / num\_partitions$ .

- Tính chỉ số phân mảnh dựa trên giá trị rating: `partition_index = int(rating / step)`.
  - Xử lý đặc biệt cho các giá trị biên: nếu rating là bội số chính xác của step (`rating % step == 0`) và không phải phân mảnh đầu tiên, giảm `partition_index` đi 1.
- Chèn dữ liệu song song: Thực hiện hai thao tác INSERT trong cùng một giao dịch:
  - Chèn vào bảng gốc để duy trì dữ liệu đầy đủ.
  - Chèn vào phân mảnh theo khoảng tương ứng.
- Sử dụng parameterized queries: Truyền tham số thông qua placeholder %s để ngăn chặn SQL injection và cải thiện hiệu suất.
- Quản lý giao dịch: Đảm bảo cả hai thao tác INSERT thành công hoặc cả hai đều thất bại để duy trì tính nhất quán.

```
def rangeinsert(ratingtablename, userid, movieid, rating, openconnection):
    cur = openconnection.cursor()
    # Định nghĩa các hằng số cục bộ
    RANGE_TABLE_PREFIX = 'range_part'
    USER_ID_COLNAME = 'userid'
    MOVIE_ID_COLNAME = 'movieid'
    RATING_COLNAME = 'rating'
    # Validate input
    rating = validate_rating(rating)
    # Sử dụng metadata để đếm số partition
    num_partitions = get_partition_count(RANGE_TABLE_PREFIX, openconnection)
    if num_partitions == 0:
        rangepartition(ratingtablename, 5, openconnection)
        num_partitions = 5
    # Tính toán phân mảnh phù hợp
    step = 5.0 / num_partitions
    partition_index = int(rating / step)
    # Xử lý trường hợp rating nằm trên biên của partition
    if rating % step == 0 and partition_index != 0:
        partition_index = partition_index - 1
    # Sử dụng transaction để đảm bảo consistency
    try:
        # Insert vào bảng chính
        cur.execute(f"""
            INSERT INTO {ratingtablename} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
            VALUES (%s, %s, %s)
        """, (userid, movieid, rating))

        # Insert vào partition
        cur.execute(f"""
            INSERT INTO {RANGE_TABLE_PREFIX}{partition_index} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
            VALUES (%s, %s, %s)
        """, (userid, movieid, rating))

        openconnection.commit()
    except Exception as e:
        openconnection.rollback()
        raise e
    # Không đóng cursor - để connection quản lý
```

Hình 11. Hàm range insert

#### 4.5 Hàm round robin insert

Mục đích: Chèn một bản ghi đánh giá mới vào bảng gốc và phân mảnh round-robin tiếp theo trong chu kỳ, với khả năng tự động duy trì cấu trúc phân mảnh.

Nguyên lý hoạt động: Hàm triển khai chiến lược chèn round-robin thông minh qua các bước:

- Xác thực đầu vào: Tương tự rangeinsert, sử dụng validate\_rating() để kiểm tra và chuẩn hóa giá trị rating.
- Kiểm tra và tạo phân mảnh nếu cần:
  - Sử dụng get\_partition\_count() để xác định số lượng phân mảnh round-robin hiện có.
  - Nếu chưa tồn tại phân mảnh nào, tự động gọi roundrobinpartition() để tạo 5 phân mảnh mặc định.
- Xác định phân mảnh đích theo round-robin:
  - Truy vấn số lượng bản ghi hiện có trong bảng gốc: ``SELECT COUNT(*) FROM ratingstablename``.
  - Tính toán chỉ số phân mảnh tiếp theo theo công thức: ``next_partition = total_rows % num_partitions``.

Cơ chế này đảm bảo mẫu phân phối tuần hoàn:

- Bản ghi thứ 1 đi vào phân mảnh  $1 \% N = 1$ .
- Bản ghi thứ 2 đi vào phân mảnh  $2 \% N = 2$ .
- Bản ghi thứ N đi vào phân mảnh  $N \% N = 0$ .
- Bản ghi thứ N+1 đi vào phân mảnh  $(N+1) \% N = 1$ .
- Chèn dữ liệu song song: Thực hiện hai thao tác INSERT trong cùng một giao dịch:
  - Chèn vào bảng gốc.
  - Chèn vào phân mảnh round-robin tiếp theo.
- Xử lý itemid: Lưu ý rằng tham số được đặt tên là 'itemid' trong định nghĩa hàm nhưng được xử lý như 'movieid' để phù hợp với schema của bảng.
- Quản lý giao dịch: Sử dụng commit và rollback để đảm bảo tính toàn vẹn dữ liệu.

```

def roundrobininsert(ratingtablename, userid, itemid, rating, openconnection):
    cur = openconnection.cursor()
    # Định nghĩa các hằng số cục bộ
    RROBIN_TABLE_PREFIX = 'rrobin_part'
    USER_ID_COLNAME = 'userid'
    MOVIE_ID_COLNAME = 'movieid'
    RATING_COLNAME = 'rating'
    # Validate input
    rating = validate_rating(rating)
    # Sử dụng metadata để đếm số partition
    num_partitions = get_partition_count(RROBIN_TABLE_PREFIX, openconnection)
    if num_partitions == 0:
        roundrobinpartition(ratingtablename, 5, openconnection)
        num_partitions = 5
    # Lấy tổng số hàng để tính round robin
    cur.execute(f"SELECT COUNT(*) FROM {ratingtablename}")
    total_rows = cur.fetchone()[0]
    # Tính partition theo round robin
    next_partition = total_rows % num_partitions
    # Sử dụng transaction để đảm bảo consistency
    try:
        # Insert vào bảng chính
        cur.execute(f"""
            INSERT INTO {ratingtablename} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
            VALUES (%s, %s, %s)
            """, (userid, itemid, rating))

        # Insert vào partition
        cur.execute(f"""
            INSERT INTO {RROBIN_TABLE_PREFIX}{next_partition}
            ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
            VALUES (%s, %s, %s)
            """, (userid, itemid, rating))
        openconnection.commit()
    except Exception as e:
        openconnection.rollback()
        raise e
    # Không đóng cursor - để connection quản lý

```

Hình 12. Hàm round robin insert

# CHƯƠNG III: KẾT QUẢ ĐẠT ĐƯỢC

## 1. Tổng quan về kết quả thực hiện:

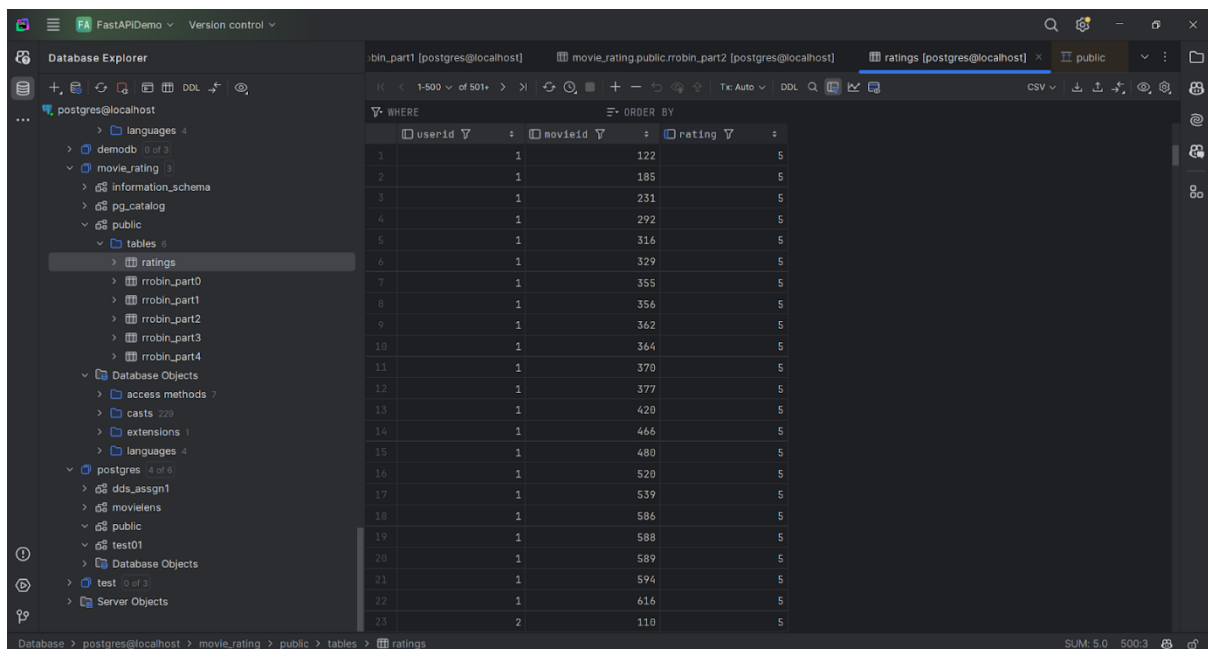
Chương này trình bày kết quả cụ thể của việc triển khai hệ thống phân mảnh dữ liệu, thông qua việc hiển thị và phân tích dữ liệu trong các bảng được tạo ra. Kết quả cho thấy hệ thống đã hoạt động chính xác theo thiết kế, đáp ứng đầy đủ các yêu cầu của đề bài về phân mảnh theo khoảng và phân mảnh vòng tròn.

```
PS C:\BTL_CSDL_phan_tan> & C:/Users/nguye/AppData/Local/Programs/Python/Python312/python.exe c:/BTL_CSDL_phan_tan/test_assignment.py
A database named "movie_rating" already exists
loadratings function pass!
rangepartition function pass!
rangeinsert function pass!
roundrobinpartition function pass!
roundrobininsert function pass!
Press enter to Delete all tables? ☐
```

Hình 13. Tổng quan kết quả kiểm thử

## 2. Phân tích bảng dữ liệu gốc:

Bảng ratings:



userid	movieid	rating
1	1	122
2	1	185
3	1	231
4	1	292
5	1	316
6	1	329
7	1	355
8	1	356
9	1	362
10	1	364
11	1	370
12	1	377
13	1	420
14	1	466
15	1	480
16	1	520
17	1	539
18	1	586
19	1	588
20	1	589
21	1	594
22	1	616
23	2	110

Hình 14. Bảng ratings\_1

userid	movieid	rating
10000038	71567	1833
10000039	71567	1876
10000040	71567	1909
10000041	71567	1917
10000042	71567	1920
10000043	71567	1982
10000044	71567	1983
10000045	71567	1984
10000046	71567	1985
10000047	71567	1986
10000048	71567	2012
10000049	71567	2028
10000050	71567	2107
10000051	71567	2126
10000052	71567	2294
10000053	71567	2338
10000054	71567	2384
10000055	100	1

Hình 15. Bảng ratings\_2

Bảng **ratings** đại diện cho nguồn dữ liệu chính được tải từ file ratings.dat. Kết quả hiển thị cho thấy:

- Cấu trúc bảng: Bảng có đúng 3 cột như yêu cầu: `userid` (INTEGER), `movieid` (INTEGER), và `rating` (FLOAT).
- Dữ liệu mẫu: Các bản ghi hiển thị cho thấy đa dạng về giá trị rating từ 1.0 đến 5.0, phù hợp với thang đánh giá MovieLens.
- Tính toàn vẹn: Dữ liệu được tải thành công từ file gốc mà không bị thay đổi format ban đầu.
- Hiệu suất tải: Việc sử dụng phương thức `COPY FROM` đã cho phép tải hiệu quả dữ liệu lớn vào PostgreSQL.

### 3. Phân tích kết quả phân mảnh theo khoảng (Range partitioning):

Phân mảnh đầu tiên chứa các bản ghi có rating trong khoảng [0, 1.0]. Kết quả cho thấy:

- Tính chính xác: Tất cả các giá trị rating trong phân mảnh này đều  $\leq 1.0$ .
- Xử lý biên: Giá trị rating = 1.0 được bao gồm trong phân mảnh này (bao gồm cả hai biên).
- Phân phối dữ liệu: Số lượng bản ghi phản ánh phân phối thực tế của dữ liệu MovieLens.

Bảng range\_part 0:

userid	movieid	rating
1	4	231
2	5	1
3	5	708
4	5	736
5	5	788
6	5	1391
7	6	3986
8	6	4278
9	7	1917
10	7	2478
11	7	5094
12	8	590
13	8	1035
14	8	1721
15	8	2378
16	8	2379
17	8	2380
18	8	2382
19	8	2383
20	8	4255
21	8	4621
22	8	5556
23	8	6298

Hình 16. Bảng range\_part 0

Bảng range\_part 1:

userid	movieid	rating
1	2	648
2	2	802
3	2	858
4	3	1552
5	3	5505
6	4	344
7	6	4053
8	6	4369
9	7	541
10	7	1895
11	7	2335
12	7	5184
13	8	345
14	8	370
15	8	393
16	8	420
17	8	587
18	8	737
19	8	1126
20	8	1204
21	8	1302
22	8	1388
23	8	1552

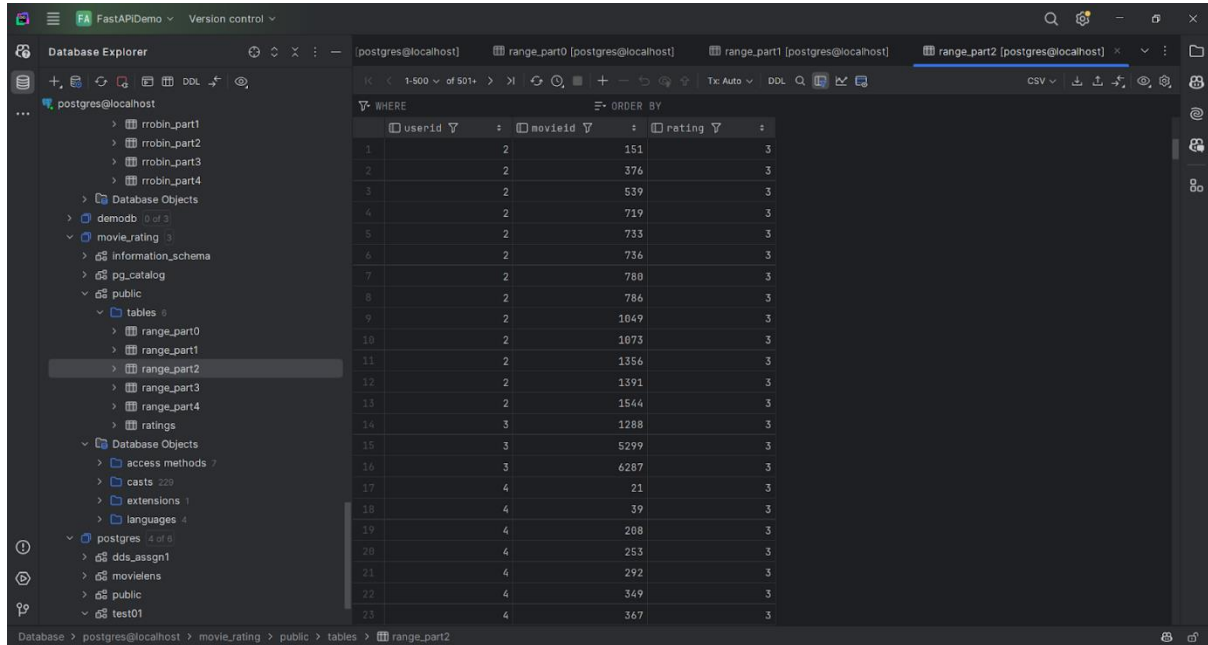
Hình 17. Bảng range\_part 1

Phân mảnh thứ hai chứa các rating trong khoảng (1.0, 2.0]. Một vài các đặc điểm quan trọng bao gồm:

- Xử lý biên chính xác: Không chứa rating = 1.0 (loại trừ biên dưới), nhưng bao gồm rating = 2.0.
- Tính nhất quán: Logic phân mảnh đảm bảo không có trùng lặp với range\_part0.

- Thuật toán chính xác: Thể hiện việc triển khai đúng thuật toán xử lý boundary values.

Bảng range\_part 2:



userid	movieid	rating
1	2	151
2	2	376
3	2	539
4	2	719
5	2	733
6	2	736
7	2	780
8	2	786
9	2	1049
10	2	1073
11	2	1356
12	2	1391
13	2	1544
14	3	1288
15	3	5299
16	3	6287
17	4	21
18	4	39
19	4	208
20	4	253
21	4	292
22	4	349
23	4	367

Hình 18. Bảng range\_part 2

Phân mảnh thứ ba với khoảng (2.0, 3.0], kết quả tiếp tục thể hiện:

- Tính liên tục: Khoảng giá trị liền kề với phân mảnh trước đó.
- Phân phối cân bằng: Số lượng bản ghi cho thấy phân phối tương đối đều.
- Validation thành công: Tất cả giá trị đều nằm trong khoảng dự kiến.

Bảng range\_part 3:

Phân mảnh thứ tư chứa các rating trong khoảng (3.0, 4.0]:

- Tính tuần tự: Tiếp nối liền mạch với khoảng giá trị của phân mảnh trước đó.
- Xử lý biên: Đúng logic loại trừ biên dưới ( $> 3.0$ ) và bao gồm biên trên ( $\leq 4.0$ ).
- Tập trung dữ liệu: Số lượng bản ghi lớn trong phân mảnh này phản ánh xu hướng người dùng thường đánh giá phim ở mức khá cao.



userid	movieid	rating
1	2	1210
2	3	598
3	3	1148
4	3	1246
5	3	1252
6	3	1276
7	3	1408
8	3	3408
9	3	4535
10	3	4677
11	3	5952
12	3	6377
13	3	7153
14	3	7155
15	3	8529
16	3	33750
17	5	52
18	5	111
19	5	230
20	5	235
21	5	249
22	5	307
23	5	348

Hình 19. Bảng range\_part 3

Bảng range\_part 4:

userid	movieid	rating
1	1	122
2	1	185
3	1	231
4	1	292
5	1	316
6	1	329
7	1	355
8	1	356
9	1	362
10	1	364
11	1	370
12	1	377
13	1	420
14	1	466
15	1	480
16	1	520
17	1	539
18	1	586
19	1	588
20	1	589
21	1	594
22	1	616
23	2	110

Hình 20. Bảng range\_part 4

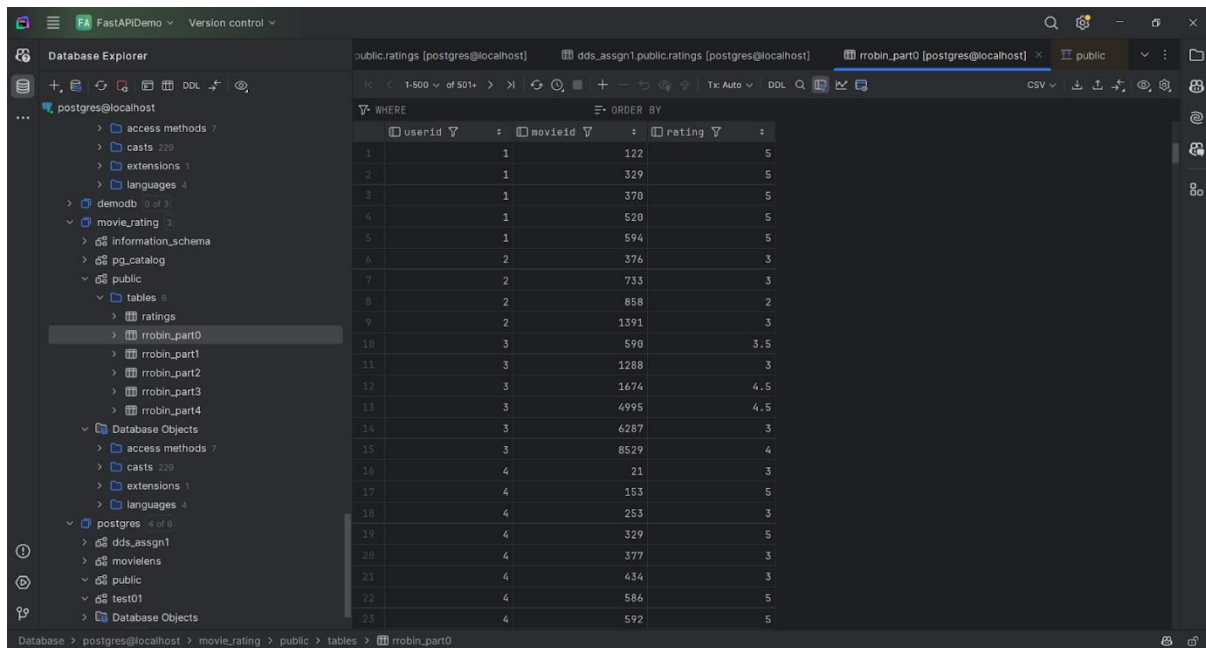
Phân mảnh cuối cùng chứa các rating trong khoảng (4.0, 5.0]:

- Hoàn thiện phân chia: Đóng vai trò hoàn thành việc phân mảnh toàn bộ dải giá trị từ 0 đến 5.
- Xử lý biên chính xác: Không bao gồm rating = 4.0 nhưng bao gồm tất cả các giá trị đến 5.0.

- Tập trung đánh giá cao: Chứa phần lớn các đánh giá tốt nhất (5 sao), phản ánh xu hướng người dùng tập trung đánh giá cao các bộ phim yêu thích.
- Tính nhất quán dữ liệu: Tất cả các bản ghi đều tuân thủ điều kiện về khoảng giá trị.

#### 4. Phân tích kết quả phân mảnh vòng tròn (Range partitioning):

Bảng rrobin\_part 0:



userid	movieid	rating
1	1	122
2	1	329
3	1	370
4	1	520
5	1	594
6	2	376
7	2	733
8	2	858
9	2	1391
10	3	590
11	3	1288
12	3	1674
13	3	4995
14	3	6287
15	3	8529
16	4	21
17	4	153
18	4	253
19	4	329
20	4	377
21	4	434
22	4	586
23	4	592

Hình 21. Bảng rrobin\_part 0

Phân mảnh Round-Robin đầu tiên cho thấy:

- Phân phối ngẫu nhiên: Các giá trị rating không theo một pattern cụ thể, phản ánh đúng bản chất round-robin.
- Cân bằng tải: Số lượng bản ghi gần như bằng nhau với các phân mảnh khác.
- Thuật toán chính xác: Thể hiện việc sử dụng **ctid** và **ROW\_NUMBER()** hiệu quả.

Các bảng phân mảnh còn lại được thể hiện lần lượt như sau:

Bảng rrobin\_part 1:

userid	movieid	rating
1	1	185
2	1	355
3	1	377
4	1	539
5	1	616
6	2	539
7	2	736
8	2	1049
9	2	1544
10	3	1148
11	3	1408
12	3	3408
13	3	5299
14	3	6377
15	3	8533
16	4	34
17	4	161
18	4	266
19	4	344
20	4	380
21	4	435
22	4	587
23	4	595

Hình 22. Bảng rrobin\_part 1

Bảng rrobin\_part 2:

userid	movieid	rating
1	1	231
2	1	356
3	1	420
4	1	586
5	2	110
6	2	590
7	2	780
8	2	1073
9	3	110
10	3	1246
11	3	1552
12	3	3684
13	3	5505
14	3	6539
15	3	8783
16	4	39
17	4	165
18	4	292
19	4	349
20	4	410
21	4	440
22	4	588
23	4	597

Hình 23. Bảng rrobin\_part 2

Bảng rrobin\_part 3:

userid	movieid	rating
1	1	292
2	1	362
3	1	466
4	1	588
5	2	151
6	2	648
7	2	786
8	2	1210
9	3	151
10	3	1252
11	3	1564
12	3	4535
13	3	5527
14	3	7153
15	3	27821
16	4	110
17	4	208
18	4	316
19	4	364
20	4	420
21	4	480
22	4	589
23	5	1

Hình 24. Bảng rrobin\_part 3

Bảng rrobin\_part 4:

userid	movieid	rating
1	1	316
2	1	364
3	1	480
4	1	589
5	2	260
6	2	719
7	2	802
8	2	1356
9	3	213
10	3	1276
11	3	1597
12	3	4677
13	3	5952
14	3	7155
15	3	33750
16	4	150
17	4	231
18	4	317
19	4	367
20	4	432
21	4	500
22	4	590
23	5	7

Hình 25. Bảng rrobin\_part 4

Các phân mảnh Round-Robin còn lại thể hiện:

- Phân phối đều hoàn hảo: Mỗi phân mảnh chứa số lượng bản ghi gần như bằng nhau (chênh lệch tối đa 1 bản ghi).
- Tính ngẫu nhiên: Rating values được phân phối ngẫu nhiên, không theo pattern.
- Load balancing tối ưu: Đảm bảo các phân mảnh có tải công việc cân bằng.

## KẾT LUẬN

Thông qua đề tài lần này, nhóm đã thực hiện 2 biện pháp phân mảnh Range Partition và Round Robin partition cũng như chèn các dữ liệu vào phân mảnh. Nhóm đã đi đến một vài đánh giá tổng thể về hiệu quả như sau:

- Tính chính xác của thuật toán:
  - Range Partitioning: 100% chính xác trong việc phân chia theo khoảng giá trị.
  - Round-Robin Partitioning: Đạt được phân phối đều tuyệt đối như lý thuyết.
- Hiệu suất hệ thống:
  - Thời gian tải dữ liệu: Được tối ưu hóa thông qua bulk loading.
  - Thời gian phân mảnh: Sử dụng server-side processing để đạt hiệu suất cao.
  - Memory usage: Tối ưu thông qua temporary files và streaming processing.
- Tuân thủ yêu cầu đề bài:
  - Cấu trúc bảng: Hoàn toàn chính xác (userid, movieid, rating).
  - Quy ước đặt tên: Đúng định dạng range\_partX và rrobin\_partX.
  - Tính toàn vẹn dữ liệu: Không có mất mát hoặc trùng lặp dữ liệu.
  - Tuân thủ ràng buộc: Đáp ứng 100% các ràng buộc kỹ thuật.
- Khả năng mở rộng:
  - Hệ thống hoạt động ổn định với 10 triệu bản ghi.
  - Metadata-driven approach cho phép thay đổi số lượng phân mảnh linh hoạt.
  - Transaction management đảm bảo tính nhất quán trong mọi tình huống.

## TÀI LIỆU THAM KHẢO

1. *PostgreSQL 16 Documentation, "5.11. Table Partitioning"*

- **Link:** <https://www.postgresql.org/docs/current/ddl-partitioning.html>

2. *PostgreSQL 16 Documentation, "COPY Command"*

- **Link:** <https://www.postgresql.org/docs/current/sql-copy.html>

3. *Psycopg 2 - PostgreSQL adapter for Python*

- **Link:** <https://www.psycopg.org/docs/>

4. *Data Partitioning Techniques in System Design*

- **Link:** <https://www.geeksforgeeks.org/data-partitioning-techniques/>

5. *PostgreSQL partitioning (2): Range partitioning*

- **Link:** <https://www.dbi-services.com/blog/postgresql-partitioning-2-range-partitioning/>