BỘ THÔNG TIN VÀ TRUYỀN THÔNG

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÁO CÁO BÀI TẬP LỚN

MÔN HỌC: CƠ SỞ DỮ LIỆU PHÂN TÁN

Đề tài: Mô phỏng các phương pháp phân mảnh dữ liệu trên PostgreSQL

Nhóm: 17 – Thành viên nhóm:

STT	Họ và tên	Mã sinh viên
1	Nguyễn Duy Anh	B22DCCN025
2	Nguyễn Thị Ngân	B22DCCN582
3	Nguyễn Nhật Thành	B22DCCN795

Giảng viên: TS. Kim Ngọc Bách

Hà Nội, 2025

BẢNG PHÂN CHIA CÔNG VIỆC

Thành viên	Lập trình	Viết báo cáo
Nguyễn Duy Anh	Range_Insert, count_partitions	Chương 2, 5
Nguyễn Thị Ngân	Loadratings, range_partition	Chương 1, 4, tổng hợp báo cáo
Nguyễn Nhật Thành	RoundRobin_partition, RoundRobin_insert	Chương 3, 6, 7

MỤC LỤC

LỜI CẢM ƠN

Trước khi khép lại báo cáo bài tập nhóm cuối kỳ môn *Cơ sở dữ liệu phân tán*, nhóm chúng em xin trân trọng gửi lời cảm ơn sâu sắc đến **Thầy Kim Ngọc Bách**, giảng viên đã trực tiếp giảng dạy và hướng dẫn chúng em trong suốt học kỳ vừa qua.

Bài tập cuối kỳ với nội dung *mô phỏng các phương pháp phân mảnh dữ liệu trong cơ sở dữ liệu phân tán* đã mang lại cho chúng em cơ hội được vận dụng kiến thức lý thuyết vào một tình huống thực tiễn. Trong quá trình triển khai bài toán, nhóm đã có dịp tìm hiểu sâu hơn về các kỹ thuật phân mảnh dữ liệu, đặc biệt là phân mảnh ngang – một kỹ thuật có ý nghĩa lớn trong việc tối ưu hóa hiệu năng, khả năng mở rộng và quản lý dữ liệu phân tán. Đây thực sự là một trải nghiệm học thuật bổ ích, giúp chúng em hiểu rõ hơn về mối liên hệ giữa lý thuyết và ứng dụng thực tế.

Chúng em ý thức được rằng những kết quả đạt được trong báo cáo này có sự đóng góp rất lớn từ sự tận tâm và nghiêm túc trong giảng dạy của Thầy. Thầy không chỉ truyền đạt kiến thức mà còn truyền cảm hứng để chúng em nỗ lực hơn trong quá trình học tập và nghiên cứu.

Một lần nữa, chúng em xin được bày tỏ lòng biết ơn chân thành đến Thầy. Kính chúc Thầy luôn dồi dào sức khỏe, hạnh phúc và tiếp tục thành công trên sự nghiệp trồng người, nghiên cứu và truyền cảm hứng cho các thế hệ sinh viên.

Xin trân trọng cảm ơn.

PHẦN I: GIỚI THIỆU

1. Mục tiêu bài tập

Bài tập lớn này nhằm mục đích **mô phỏng phân mảnh ngang** trên cơ sở dữ liệu PostgreSQL, cụ thể là phân mảnh theo hai phương pháp phổ biến: **Range Partitioning** (phân mảnh theo khoảng) và **Round Robin Partitioning** (phân mảnh vòng tròn). Qua đó, bài tập này giúp người thực hiện hiểu rõ cơ chế phân chia dữ liệu thành các phân mảnh nhỏ hơn, từ đó nâng cao hiệu quả truy vấn và quản lý dữ liệu trong các hệ thống cơ sở dữ liệu phân tán hoặc quy mô lớn.

2. Dữ liệu đầu vào

Dữ liệu được sử dụng là tệp *ratings.dat* thuộc bộ dữ liệu *MovieLens* – một tập dữ liệu tiêu chuẩn được sử dụng rộng rãi trong lĩnh vực đề xuất phim và khai phá dữ liệu. Định dạng mỗi dòng dữ liệu trong tệp có dạng như sau:

UserID::MovieID::Rating::Timestamp

- Trong đó:
  - **UserID**: mã định danh của người dùng (kiểu số nguyên)
  - **MovieID**: mã định danh của phim (kiểu số nguyên)
  - **Rating**: điểm đánh giá của người dùng dành cho phim (kiểu số thực, từ 0 đến 5)
  - **Timestamp**: thời gian đánh giá (kiểu số nguyên, đại diện thời gian UNIX)

Tuy nhiên, trong các yêu cầu của bài tập (mục 4), chỉ cần sử dụng ba trường đầu (UserID, MovieID, Rating), không sử dụng trường Timestamp.

3. Yêu cầu kỹ thuật

Để đáp ứng mục tiêu đề ra, bài tập yêu cầu thực hiện các công việc chính sau:

- Tạo bảng *Ratings* trong cơ sở dữ liệu PostgreSQL Bảng Ratings là bảng gốc để lưu trữ toàn bộ dữ liệu đánh giá phim được tải từ tệp ratings.dat. Cấu trúc bảng bao gồm các cột: UserID (integer), MovieID (integer), Rating (float).

- Cài đặt 6 hàm Python tương tác với PostgreSQL, phục vụ việc tải dữ liệu và phân mảnh dữ liệu. Các hàm phải tuân thủ nghiêm ngặt quy ước đặt tên bảng và cấu trúc dữ liệu, bao gồm:

  a. LoadRatings(ratingstablename, ratingsfilepath, openconnection): Hàm thực hiện đọc tệp dữ liệu và tải toàn bộ thông tin vào bảng Ratings trong cơ sở dữ liệu.

  b. Range_Partition(ratingstablename, numberofpartitions, openconnection): Hàm tạo ra các bảng phân mảnh theo phương pháp Range, chia dữ liệu dựa trên giá trị Rating thành các khoảng đều nhau.

  c. RoundRobin_Partition(ratingstablename, numberofpartitions, openconnection): Hàm tạo các bảng phân mảnh theo phương pháp Round Robin, phân phối dữ liệu đều theo vòng tròn.

  d. Range_Insert(ratingstablename, userid, movieid, rating, openconnection): Hàm thực hiện chèn một bản ghi mới vào bảng Ratings và vào bảng phân mảnh Range tương ứng.

  e. RoundRobin_Insert(ratingstablename, userid, movieid, rating, openconnection): Hàm thực hiện chèn một bản ghi mới vào bảng Ratings và bảng phân mảnh Round Robin tiếp theo trong chu kỳ.

  f. count_partitions(openconnection): Hàm đếm số lượng bảng phân mảnh hiện có trong cơ sở dữ liệu để quản lý và kiểm tra.

PHẦN II: CƠ SỞ LÝ THUYẾT

1. Phân mảnh dữ liệu

1.1. Định nghĩa

- Phân mảnh dữ liệu là kỹ thuật chia một bảng cơ sở dữ liệu lớn thành nhiều thành phần nhỏ hơn sao cho mỗi thành phần đó vẫn chứa đủ thông tin để phục vụ một số truy vấn nhất định. Có hai kiểu phân mảnh cơ bản:
  - Phân mảnh ngang (Horizontal fragmentation).
  - Phân mảnh dọc (Vertical fragmentation).

1.2. Phân mảnh ngang (Horizontal fragmentation)

- Phân mảnh ngang là kỹ thuật chia một bảng thành các mảnh dựa trên các bản ghi, sao cho mỗi mảnh chứa một tập hợp các bản ghi thoả mãn một điều kiện nhất định.
- Đặc điểm:
  - Mỗi mảnh có cùng cấu trúc với bảng gốc.
  - Dữ liệu được chia dựa trên giá trị của thuộc tính.
  - Phù hợp khi số lượng bản ghi lớn, truy vấn thường chỉ lấy một phần dữ liệu dựa trên điều kiện lọc.

1.3. Phân mảnh dọc (Vertical fragmentation)

- Là kỹ thuật chia một bảng thành các mảnh dựa trên các thuộc tính sao cho mỗi mảnh chứa một tập hợp các thuộc tính nhất định của bảng, có thể kèm theo khóa chính để liên kết lại dữ liệu khi cần thiết.
- Đặc điểm:
  - Mỗi mảnh chứa ít hơn toàn bộ cột (thuộc tính), giảm kích thước bản ghi.
  - Phù hợp khi có nhiều truy vấn chỉ cần một nhóm các thuộc tính con.
  - Cần đảm bảo mỗi mảnh vẫn có cột đủ để liên kết lại.

1.4. Lợi ích của phân mảnh trong hệ thống phân tán

- Kỹ thuật phân mảnh dữ liệu đem lại một số lợi ích:
  - Tăng hiệu suất: Phân mảnh giúp giảm khối lượng dữ liệu mỗi truy vấn cần xử lý, từ đó tăng tốc độ truy xuất và xử lý dữ liệu.

- Cân bằng tải: Dữ liệu được phân bố trên nhiều máy chủ hoặc địa điểm khác nhau, giúp tránh quá tải ở một nút dữ liệu duy nhất.
- Tính sẵn sàng và khả năng mở rộng: Nếu một mảnh dữ liệu gặp sự cố, hệ thống vẫn có thể hoạt động với các mảnh khác.
- Tối ưu hóa truy vấn: Chỉ truy vấn trên các mảnh cần thiết, không phải toàn bộ bảng.
- Bảo mật và quản lý: Dễ dàng phân chia dữ liệu theo các mức độ bảo mật khác nhau và quản lý dữ liệu hiệu quả hơn.

2. Phân mảnh theo khoảng

Khi áp dụng phân mảnh ngang theo khoảng, ta chia các bản ghi của bảng gốc thành N bảng con dựa trên khoảng giá trị của một thuộc tính (hiện tại đang xét Rating), sao cho mỗi khoảng có cùng độ rộng về lý thuyết.

2.1. Mô tả chung

- Đầu vào:
  - Bảng Ratings từ file "ratings.dat" có cột Rating với giá trị trong đoạn [minRating, maxRating].
  - Số phân mảnh N ($N > 0$).
- Mục tiêu: Tạo thành N bảng con "range_part0", "range_part1",... sao cho:
  - Partition 0 chứa tất cả bản ghi mà Rating nằm trong đoạn [minRating, minRating + x].
  - Partition 1 chứa tất cả bản ghi mà Rating nằm trong đoạn [minRating + x, minRating + $2x$].
  - Partition N - 1 chứa tất cả bản ghi mà Rating nằm trong đoạn [minRating + (N - 1)x, maxRating].

2.2. Công thức tính x

$$x = maxRating – minRatingN$$

- Sau khi tính x, ta xác định cho mỗi Partition i ($0 < i < N$ - 1):
  - Giá trị biên dưới: lowerBound = minRating + ix.
  - Giá trị biên trên:

- Nếu $i < N - 1$: upperBound = minRating + $(i + 1)x$.
- Nếu $i = N - 1$: upperBound = maxRating.

3. Phân mảnh vòng tròn

- Định nghĩa: Mỗi bản ghi của bảng gốc được đưa lần lượt vào các partition theo thứ tự cố định, bất kể giá trị của các cột.
- Phân phối dữ liệu luân phiên đều vào các mảnh: Với *N* partiiton rrobin_part0, rrobin_part1,...:
  - Duyệt từng bản ghi thứ *i* (tính từ 0) trong bảng gốc:
    - Tính *partition_index = i mod N*.
    - Chèn bản ghi vào partition rrobin_part{*partition_index*}.
  - Kết quả: Dữ liệu được chia luân phiên vào *N* mảnh, mỗi mảnh chứa xấp xỉ bằng nhau số bản ghi.

PHẦN III: MÔI TRƯỜNG VÀ CÔNG CỤ

1. Giới thiệu ngôn ngữ lập trình

- Ngôn ngữ lập trình: **Python 3.12.5**
- Python là ngôn ngữ lập trình cấp cao, phổ biến trong các ứng dụng xử lý dữ liệu, khoa học máy tính và tích hợp hệ thống. Trong bài tập này, Python được sử dụng để:
    - Kết nối và tương tác với hệ quản trị cơ sở dữ liệu PostgreSQL.
    - Thực hiện logic phân mảnh ngang (range partition và round robin).
    - Tự động tạo các bảng phân mảnh và chèn dữ liệu phù hợp vào từng phân mảnh.
    - Thực hiện kiểm thử phân mảnh, chèn dữ liệu vào các bảng phân mảnh.

2. Giới thiệu thư viện sử dụng

- Thư viện sử dụng: **psycopg2**
- psycopg2 là một thư viện Python cho phép kết nối và thao tác với cơ sở dữ liệu PostgreSQL một cách hiệu quả. Đây là thư viện client phổ biến nhất để làm việc với PostgreSQL trong Python.
- Một số chức năng sử dụng:
    - Kết nối đến PostgreSQL: psycopg2.connect(...)
    - Thực thi truy vấn SQL: cursor.execute(...)
    - Giao dịch dữ liệu: commit/rollback khi cần thiết.
    - Đóng kết nối sau khi hoàn tất.

3. Giới thiệu hệ quản trị cơ sở dữ liệu

- Hệ quản trị cơ sở dữ liệu: **PostgreSQL phiên bản 15 trở lên**
- PostgreSQL là một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở mạnh mẽ, hỗ trợ các tính năng nâng cao như:
    - Phân vùng bảng (table partitioning)

- Giao dịch (ACID compliance)
- Tích hợp với nhiều ngôn ngữ và thư viện

Trong bài tập này, PostgreSQL đóng vai trò lưu trữ dữ liệu gốc và các bảng phân mảnh sau khi xử lý. Các bảng sẽ được tạo ra tự động bởi script Python, sau đó dữ liệu được chèn vào theo đúng thuật toán phân mảnh.

4. Môi trường triển khai

Có thể sử dụng một trong hai hệ điều hành sau để triển khai:

a. Ubuntu 22.04 LTS

- Hệ điều hành mã nguồn mở, ổn định, được sử dụng rộng rãi cho môi trường server và học thuật.
- Dễ dàng cài đặt PostgreSQL thông qua apt.
- Dễ thiết lập môi trường Python và thư viện psycopg2 qua pip.

b. Windows 10

- Phù hợp cho các bạn học sinh/sinh viên quen dùng Windows.
- PostgreSQL có thể cài đặt dễ dàng qua trình cài đặt chính thức.
- Python và psycopg2 có thể được cài đặt qua Microsoft Store hoặc từ python.org.

PHẦN IV: THIẾT KẾ HỆ THỐNG

1. Sơ đồ hàm

Hệ thống bao gồm các hàm xử lý chính dùng để phân mảnh bảng Ratings theo hai phương pháp: Range Partitioning và Round Robin Partitioning. Dưới đây là mô tả sơ đồ từng hàm:

1.1. Hàm loadratings

Hàm loadratings dùng để nạp dữ liệu ban đầu từ file ratings.dat (hoặc các file tương tự) vào bảng ratings trong cơ sở dữ liệu PostgreSQL. Hàm thực hiện các bước chính sau theo trình tự logic:

Hàm loadratings dùng để nạp dữ liệu ban đầu từ file ratings.dat (hoặc các file tương tự) vào bảng ratings trong cơ sở dữ liệu PostgreSQL. Hàm thực hiện các bước chính sau theo trình tự logic:

a. Khởi tạo và chuẩn bị

- Bắt đầu đếm thời gian thực thi bằng time.time().
- Tạo một con trỏ (cursor) từ kết nối đã mở (openconnection) để thực hiện các truy vấn SQL.

b. Xóa bảng cũ (nếu có)

- Thực thi câu lệnh SQL DROP TABLE IF EXISTS để xóa bảng ratingstablename nếu đã tồn tại nhằm tránh lỗi khi tạo mới.

c. Tạo bảng mới

- Tạo bảng ratingstablename mới với cấu trúc:
  - userid (kiểu INT)
  - movieid (kiểu INT)
  - rating (kiểu FLOAT)
- Không tạo bảng tạm riêng biệt, mà ghi dữ liệu trực tiếp vào bảng chính.
- Sau khi nạp dữ liệu, sử dụng ALTER TABLE để đặt khóa chính là tổ hợp (userid, movieid), đảm bảo mỗi người dùng chỉ đánh giá mỗi phim một lần.

d. Đọc và xử lý dữ liệu từ file

- Mở file dữ liệu tại đường dẫn ratingsfilepath.
- Đọc từng dòng của file và tách chuỗi theo dấu "::".
- Lấy các trường:
    - userid: chỉ số 0
    - movieid: chỉ số 1
    - rating: chỉ số 2
- Ghi lại các trường này vào một bộ đệm (StringIO) dưới định dạng phân tách bằng tab (\t) để dễ dàng nạp vào cơ sở dữ liệu.

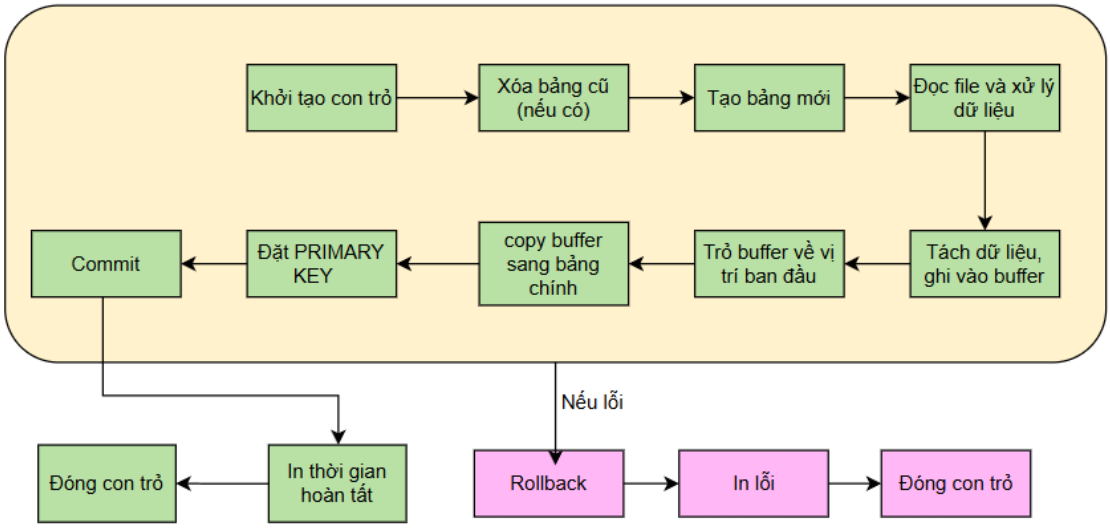
e. Nạp dữ liệu trực tiếp vào bảng chính

- Trỏ con trỏ của bộ đệm về đầu (buffer.seek(0)).
- Sử dụng phương thức copy_from của con trỏ PostgreSQL để sao chép toàn bộ dữ liệu từ buffer vào bảng ratingstablename. Phương pháp này giúp tối ưu hiệu năng và nhanh hơn nhiều so với INSERT từng dòng.

f. Kết thúc và xử lý lỗi

- Sau khi nạp dữ liệu xong và thiết lập khóa chính, thực hiện commit() để lưu lại tất cả thay đổi vào cơ sở dữ liệu.
- In ra thời gian thực thi của toàn bộ hàm.
- Trong trường hợp xảy ra lỗi ở bất kỳ bước nào:
    - Gọi rollback() để hủy bỏ các thay đổi chưa hoàn tất, tránh dữ liệu không đồng nhất.
    - In ra thông báo lỗi chi tiết để tiện gỡ lỗi.
- Cuối cùng, đóng con trỏ (cur.close()), nhưng không đóng kết nối theo đúng yêu cầu của đề bài.

g. Tóm tắt luồng hoạt động tổng quát:



Sơ đồ luồng hoạt động tóm tắt của hàm Loadratings

1.2. Hàm rangepartition

Hàm rangepartition được dùng để **phân mảnh ngang dữ liệu bảng ratings theo giá trị rating** bằng cách chia toàn bộ dữ liệu trong bảng gốc thành nhiều bảng nhỏ hơn (các phân mảnh), **mỗi phân mảnh chứa các bản ghi nằm trong một khoảng giá trị rating nhất định**. Hàm thực hiện các bước sau theo trình tự logic:

a. Khởi tạo và chuẩn bị

- Bắt đầu đếm thời gian thực thi để đo hiệu suất.
- Kiểm tra numberofpartitions. Nếu giá trị nhỏ hơn hoặc bằng 0, hàm sẽ dừng và đưa ra lỗi (raise ValueError).
- Tạo con trỏ (cursor) để thao tác với cơ sở dữ liệu thông qua kết nối đã mở (openconnection).
- Đặt tên tiền tố cho các bảng phân mảnh là range_part.
- Tính kích thước bước (step) cho mỗi phân mảnh bằng công thức 5.0 / numberofpartitions, vì giá trị rating nằm trong khoảng từ 0.0 đến 5.0.

b. Xóa bảng phân mảnh cũ (nếu có)

- Với mỗi chỉ số phân mảnh i, thực hiện câu lệnh DROP TABLE IF EXISTS range_part<i> để xóa bảng nếu đã tồn tại, tránh xung đột dữ liệu.

c. Tạo các bảng phân mảnh

- Lặp qua numberofpartitions, tạo mới bảng range_part<i> với các cột:
    - userid (kiểu INTEGER)
    - movieid (kiểu INTEGER)
    - rating (kiểu FLOAT)
- Đặt khóa chính là tổ hợp (userid, movieid) để đảm bảo mỗi người dùng chỉ đánh giá một phim duy nhất.

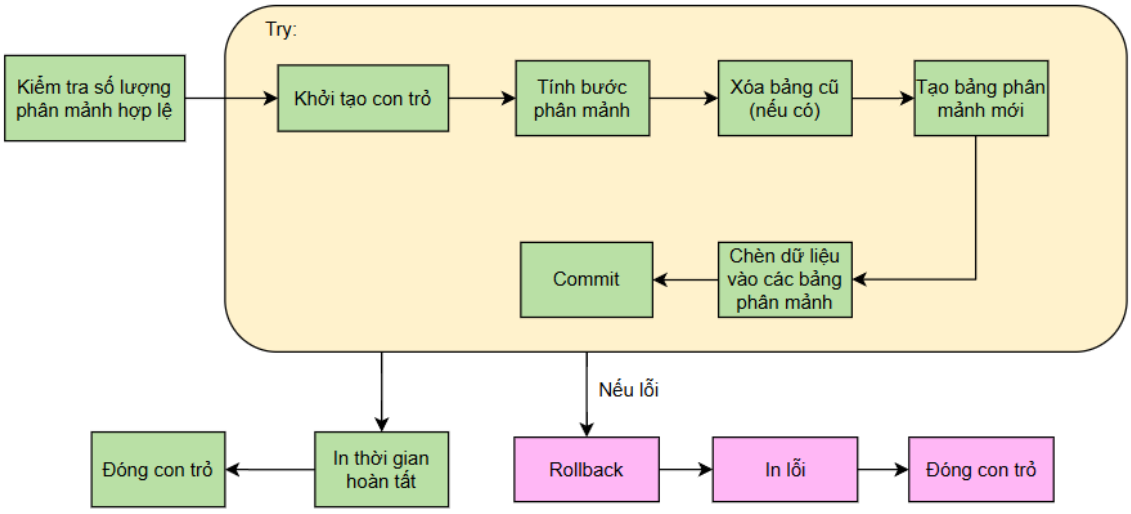
d. Phân phối dữ liệu vào các phân mảnh

- Tiếp tục lặp qua từng phân mảnh i, xác định điều kiện rating thuộc phân đoạn tương ứng:
    - Với phân mảnh đầu tiên (i == 0): điều kiện là rating >= lower AND rating <= upper
    - Với các phân mảnh còn lại: điều kiện là rating > lower AND rating <= upper
- Thực hiện truy vấn SQL INSERT INTO range_part<i> để chọn dữ liệu phù hợp từ bảng chính ratingstablename và chèn vào bảng phân mảnh tương ứng.

e. Kết thúc và xử lý lỗi

- Gọi commit() để lưu các thay đổi vào cơ sở dữ liệu nếu mọi thao tác thành công.
- In ra thời gian thực thi hoàn tất việc phân mảnh.
- Nếu có lỗi trong bất kỳ bước nào:
    - Rollback giao dịch để đảm bảo dữ liệu nhất quán.
    - In thông báo lỗi chi tiết để phục vụ việc kiểm tra và sửa lỗi.
- Cuối cùng, đóng con trỏ cơ sở dữ liệu để giải phóng tài nguyên.

f. Tóm tắt luồng hoạt động tổng quát:



Sơ đồ luồng hoạt động tóm tắt của hàm rangepartition

1.3. Hàm rangeinsert

Hàm rangeinsert thực hiện việc **chèn một dòng dữ liệu mới** (bao gồm userid, itemid, rating) vào **bảng phân mảnh theo dải giá trị (range)** tương ứng dựa trên giá trị rating. Hàm thực hiện các bước sau theo trình tự logic:

a. Khởi tạo và chuẩn bị

- Bắt đầu đếm thời gian thực thi để thống kê thời gian chạy hàm.
- Tạo con trỏ (cursor) để thực hiện các truy vấn SQL trên kết nối PostgreSQL (openconnection).

b. Xác định số lượng phân mảnh

- Gọi hàm count_partitions(prefix, openconnection) để đếm tổng số bảng phân mảnh có tiền tố là 'range_part'.
- Nếu không có phân mảnh nào, đóng con trỏ và raise lỗi.

c. Tính chỉ số phân mảnh phù hợp

- Tính **khoảng cách (delta)** cho mỗi phân mảnh: delta = 5 / number_of_partitions.
- Dựa trên giá trị rating, tính toán chỉ số phân mảnh idx = int(rating / delta).

- Nếu rating chia hết cho delta và idx > 0, thì trừ 1 để tránh đưa nhầm vào phân mảnh bên phải.
- Đảm bảo idx không vượt quá số phân mảnh (nếu bằng hoặc lớn hơn, giảm về partitions_number - 1).

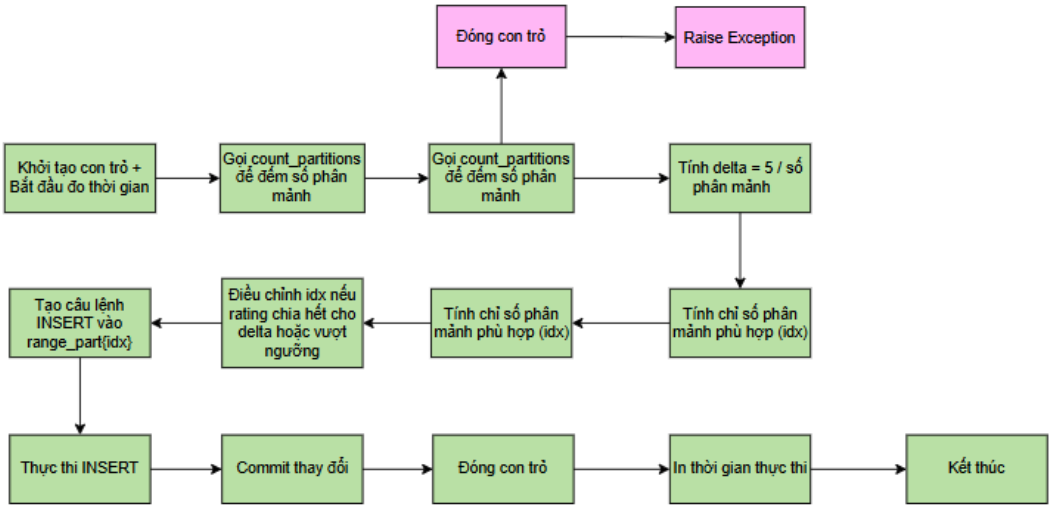
d. Chèn dữ liệu

- Tạo câu lệnh SQL INSERT để chèn dòng dữ liệu vào bảng phân mảnh range_part{idx} tương ứng.
- Thực thi câu lệnh INSERT với con trỏ.

e. Kết thúc

- Ghi nhận thay đổi với commit().
- Đóng con trỏ.
- In thời gian hoàn tất thao tác.

f. Tóm tắt luồng hoạt động tổng quát



Sơ đồ luồng hoạt động tóm tắt của hàm rangeinsert

1.4. Hàm roundrobinpartition

Hàm roundrobinpartition thực hiện phân mảnh bảng dữ liệu đầu vào theo phương pháp **vòng tròn (Round Robin)**, chia đều các bản ghi vào N bảng con. Luồng hoạt động chính của hàm được mô tả như sau:

a. Khởi tạo và chuẩn bị

- Bắt đầu đếm thời gian thực thi để phục vụ thống kê hiệu suất.
- Tạo con trỏ (cursor) để thao tác với cơ sở dữ liệu thông qua openconnection.
- Kiểm tra số lượng phân mảnh phải lớn hơn 0, nếu không thì raise lỗi.

b. Tạo bảng phân mảnh

- Lặp qua numberofpartitions, mỗi lần:
    - Xóa bảng con cũ tương ứng (rrobin_part{i}) nếu đã tồn tại.
    - Tạo bảng mới rrobin_part{i} với các cột: userid, movieid, rating và khóa chính (userid, movieid).

c. Phân phối dữ liệu theo vòng tròn

- Sử dụng truy vấn ROW_NUMBER() để đánh số dòng từ bảng gốc.
- Dựa vào chỉ số dòng (rn) và phép chia dư (MOD rn % numberofpartitions) để phân chia đều các dòng vào các bảng con tương ứng.

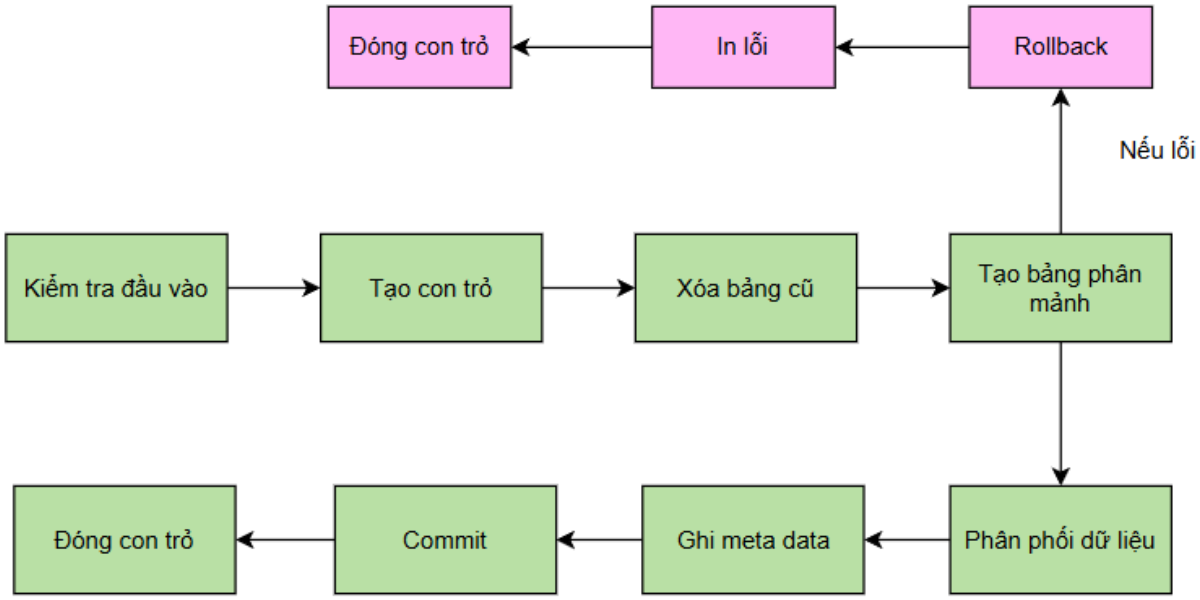
d. Ghi thông tin phân mảnh vào bảng metadata

- Chèn thông tin vào bảng partition_metadata với:
    - partition_type = 'rrobin'
    - partition_count = numberofpartitions
    - last_used = timestamp (thời điểm hiện tại, dùng như một dấu mốc).

e. Kết thúc

- Commit giao dịch để lưu lại toàn bộ thay đổi.
- Đóng con trỏ sau khi hoàn tất.
- In ra thời gian thực thi của hàm để theo dõi hiệu suất.

f. Tóm tắt luồng hoạt động tổng quát



*Sơ đồ luồng hoạt động tóm tắt của hàm r*oundrobinpartition

1.5. Hàm roundrobininsert

Hàm roundrobininsert thực hiện chức năng chèn một dòng dữ liệu mới (gồm userid, itemid, rating) vào bảng chính và phân phối dòng đó vào bảng phân mảnh phù hợp theo phương pháp **vòng tròn (Round Robin)**. Luồng hoạt động chính của hàm được mô tả như sau:

a. Khởi tạo và chuẩn bị

- Tạo con trỏ (cursor) để thao tác với cơ sở dữ liệu thông qua kết nối mở (openconnection).

b. Xác định số lượng phân mảnh

- Gọi hàm count_partitions('rrobin', openconnection) để đếm số lượng bảng phân mảnh có tiền tố 'rrobin_part'.
- Nếu không có phân mảnh nào, raise lỗi để ngắt quá trình chèn.

c. Chèn dữ liệu vào bảng chính và xác định vị trí phân mảnh

- Thực hiện chèn dữ liệu vào bảng chính ratingstablename.

- Đồng thời sử dụng truy vấn RETURNING để đếm tổng số dòng hiện có trong bảng chính.
- Tính chỉ số phân mảnh phù hợp: index = (total_rows - 1) % numberofpartitions.

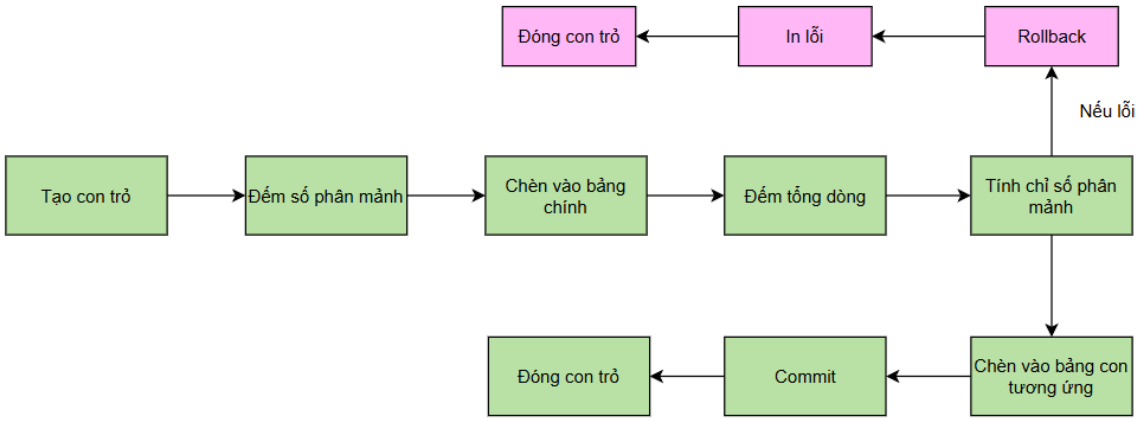
d. Chèn dữ liệu vào bảng phân mảnh

- Tạo câu lệnh SQL INSERT để chèn dữ liệu vào bảng con rrobin_part{index} tương ứng với chỉ số vừa tính được.
- Thực thi lệnh chèn bằng con trỏ SQL.

e. Kết thúc

- Ghi nhận thay đổi với commit() để đảm bảo tính bền vững.
- Đóng con trỏ sau khi hoàn tất thao tác.
- In ra thông báo chèn thành công và chỉ số phân mảnh tương ứng.

f. Tóm tắt luồng hoạt động tổng quát



Sơ đồ luồng hoạt động tóm tắt của hàm roundrobininsert

1.6. Hàm count_partitions

Đếm số phân mảnh đã được tạo dựa trên kiểu phân mảnh (range hoặc roundrobin) được lưu trong bảng partition_metadata. Hàm thực hiện các bước sau theo trình tự logic:

a. Khởi tạo

- Tạo con trỏ để tương tác với cơ sở dữ liệu qua kết nối openconnection.

b. Tạo câu lệnh SQL

- Xây dựng câu lệnh SQL:

SELECT partition_count

FROM partition_metadata

WHERE partition_type = '<type>';

- type là chuỗi đầu vào chỉ định loại phân mảnh cần đếm (range hoặc roundrobin).

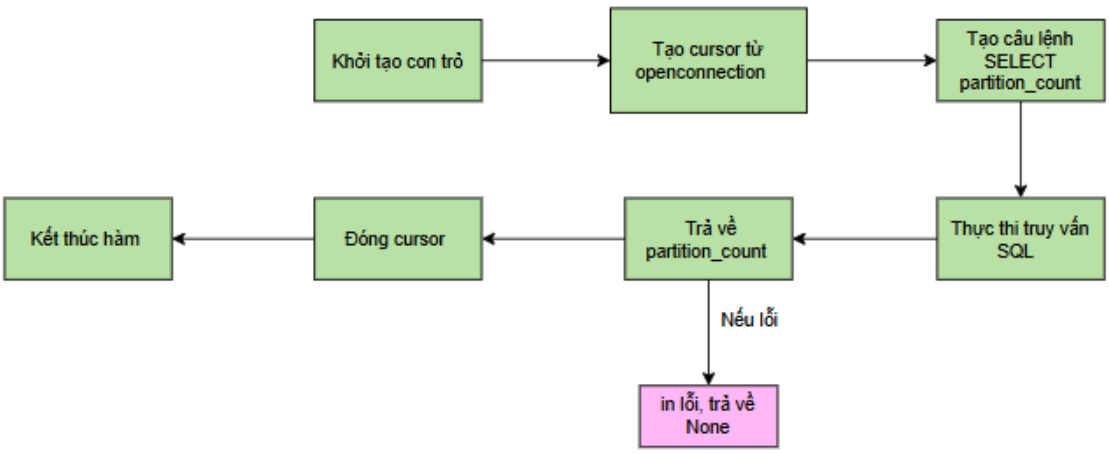
c. Thực thi và trả kết quả

- Thực thi câu lệnh SQL.
- Nếu có kết quả, trả về partition_count.
- Nếu lỗi, in lỗi và trả về None.

d. Dọn dẹp

- Dù thành công hay thất bại, con trỏ cũng được đóng trong khối finally.

f. Tóm tắt hoạt động luồng tổng quát



Sơ đồ luồng hoạt động tóm tắt của hàm count_partitions

1.7. Hàm create_metadata_table_if_not_exists

Hàm create_metadata_table_if_not_exists đảm bảo rằng bảng metadata (partition_metadata) dùng để lưu thông tin quản lý phân mảnh tồn tại trong cơ sở dữ liệu. Nếu bảng chưa tồn tại, hàm sẽ tạo mới bảng với cấu trúc phù hợp.

a. Khởi tạo và chuẩn bị

- Hàm nhận vào một đối tượng cursor từ PostgreSQL, cho phép thực thi truy vấn SQL trực tiếp.
- Không cần mở kết nối vì cursor đã được truyền từ hàm gọi.

b. Tạo bảng nếu chưa tồn tại

- Xây dựng câu lệnh SQL CREATE TABLE IF NOT EXISTS để đảm bảo bảng partition_metadata được tạo nếu chưa tồn tại.
- Bảng gồm các trường:
  - partition_type: kiểu phân mảnh ('range', 'rrobin'), là **khóa chính**.
  - partition_count: số lượng phân mảnh hiện có (**bắt buộc**).
  - last_used: chỉ số phân mảnh được sử dụng gần nhất (dùng cho Round Robin).

c. Thực thi truy vấn

- Thực hiện câu lệnh CREATE TABLE bằng con trỏ SQL.
- In câu lệnh ra màn hình để dễ kiểm tra hoặc debug trong quá trình thực thi.

d. Tóm tắt luồng hoạt động tổng quát

Sơ đồ luồng hoạt động tóm tắt của hàm create_metadata_table_if_not_exists

2. Thiết kế bảng cơ sở dữ liệu

2.1. Bảng Ratings:

Bảng Ratings là bảng dữ liệu chính chứa thông tin đánh giá (rating) mà người dùng (user) đã đưa ra cho từng bộ phim (movie). Bảng này được sử dụng làm dữ liệu đầu vào cho quá trình phân mảnh ngang.

```
CREATE TABLE Ratings (
UserID INT,
MovieID INT,
Rating FLOAT,
);
```

- **Mô tả các trường:**
  - UserID: ID của người dùng (kiểu số nguyên).
  - MovieID: ID của bộ phim (kiểu số nguyên).
  - Rating: Điểm đánh giá do người dùng đưa ra cho bộ phim, là một số thực (FLOAT).
- **Ghi chú:**
  - Dữ liệu trong bảng này được sử dụng để thực hiện phân mảnh theo nhiều chiến lược khác nhau như Range hoặc Round Robin.
  - Trong trường hợp phân mảnh, các bản sao của bảng Ratings sẽ được lưu vào các bảng phân mảnh tương ứng.

2.2. Bảng partition_metadata

Bảng partition_metadata được sử dụng để lưu trữ trạng thái của quá trình phân mảnh, thay thế cho việc sử dụng biến toàn cục trong chương trình Python. Điều này đảm bảo tính nhất quán và giúp quản lý thông tin phân mảnh một cách rõ ràng trong cơ sở dữ liệu.

```
CREATE TABLE partition_metadata (
 partition_type VARCHAR(20) PRIMARY KEY,
 partition_count INT,
 last_used INT
);
```

- **Mô tả các trường:**
  - **partition_type:** Kiểu phân mảnh được sử dụng, ví dụ 'range' hoặc 'round_robin' (dùng làm khóa chính để phân biệt loại phân mảnh).
  - **partition_count:** Số lượng phân mảnh hiện đang tồn tại đối với kiểu phân mảnh này.
  - **last_used:** Chỉ số của phân mảnh được sử dụng gần nhất (chỉ áp dụng cho phương pháp Round Robin để biết lần chèn kế tiếp sẽ vào phân mảnh nào).

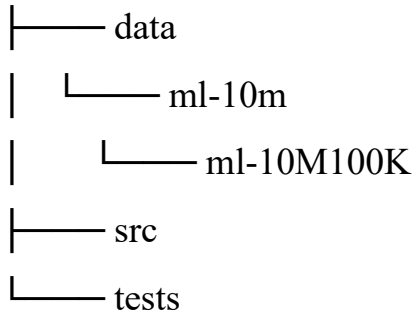
- **Ví dụ dữ liệu mẫu:**

partition_type	partition_count	last_used
round_robin	4	2
range	4	NULL

- **Ghi chú:**
  - Với phương pháp **Round Robin**, last_used sẽ giúp xác định phân mảnh kế tiếp cần chèn khi thêm bản ghi mới.
  - Với phương pháp **Range**, trường last_used có thể không cần thiết, nhưng vẫn giữ để đồng nhất cấu trúc.

PHẦN V: CÀI ĐẶT CHƯƠNG TRÌNH

1. Cấu trúc repository



- data\m1-10m\m1-10M100K: Package chứa tập các dữ liệu phục vụ cho việc phân mảnh.
- src: Package chứa file các hàm phục vụ cho việc phân mảnh.
- tests: Package chứa các file phục vụ cho việc kiểm tra phân mảnh dữ liệu.

2. Cài đặt giải thuật

2.1. Hàm loadratings()

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

    try:
        cur.execute(f"DROP TABLE IF EXISTS {ratingstablename};")
        cur.execute(f"""
            CREATE TABLE {ratingstablename} (
                userid INT,
                movieid INT,
                rating FLOAT
            );
        """)

        buffer = StringIO()
        with open(ratingsfilepath, 'r') as f:
            for line in f:
                parts = line.strip().split('::')
                if len(parts) >= 3:
                    buffer.write(f"{parts[0]}\t{parts[1]}\t{parts[2]}\n")
        buffer.seek(0)

        cur.copy_from(buffer, ratingstablename, sep='\t', columns=('userid', 'movieid', 'rating'))

        cur.execute(f"ALTER TABLE {ratingstablename} ADD PRIMARY KEY (userid, movieid);")

        openconnection.commit()
        print(f"[loadratings] Completed in {time.time() - start_time:.2f} seconds")

    except Exception as e:
        openconnection.rollback()
        print(f"[loadratings] Error: {e}")
        raise
    finally:
        cur.close()
```

Hàm loadrating()

- **Mục đích:** Tạo bảng gốc gồm các thuộc tính: *userid, movieid, rating* và nạp toàn bộ dữ liệu từ file *ratings.dat* vào bảng.
- **Thực thi:**
  - **Xóa (nếu có) và tạo mới bảng gốc:**
    - Sử dụng lệnh DROP TABLE IF EXISTS để đảm bảo không có bảng cũ trùng tên tồn tại.

- Tạo mới bảng ratingstablename với ba cột: userid INT, movieid INT, rating FLOAT.

- **Đọc file dữ liệu đầu vào:**
  - Mở và đọc từng dòng trong file ratings.dat.
  - Mỗi dòng có định dạng: userid::movieid::rating::timestamp.
  - Tách chuỗi theo ký hiệu "::", chỉ giữ lại ba trường userid, movieid, rating.
  - Ghi dữ liệu vào một bộ đệm (StringIO) theo định dạng: userid \t movieid \t rating.
- **Tải nhanh dữ liệu vào bảng:**
  - Sử dụng phương thức cur.copy_from(...) để sao chép toàn bộ nội dung từ buffer vào bảng gốc.
  - copy_from giúp tăng tốc độ chèn dữ liệu so với INSERT từng dòng riêng lẻ.
- **Thiết lập khóa chính và lưu lại:**
  - Thêm khóa chính (userid, movieid) để đảm bảo tính duy nhất cho mỗi bản ghi.
  - Ghi lại các thay đổi bằng commit.

2.2. Hàm rangepartition()

```python
def rangepartition(ratingstablename, numberofpartitions, openconnection):
    if numberofpartitions <= 0:
        raise ValueError("Number of partitions must be positive")

    start_time = time.time()
    try:
        cur = openconnection.cursor()
        RANGE_TABLE_PREFIX = 'range_part'

        step = 5.0 / numberofpartitions

        # Tạo các bảng phân mảnh
        for i in range(numberofpartitions):
            cur.execute(f"""
                DROP TABLE IF EXISTS {RANGE_TABLE_PREFIX}{i};
                CREATE TABLE {RANGE_TABLE_PREFIX}{i} (
                    userid INTEGER,
                    movieid INTEGER,
                    rating FLOAT,
                    PRIMARY KEY (userid, movieid)
                );
            """)

        # Sử dụng CASE WHEN để chia vào nhiều bảng trong một truy vấn duy nhất
        for i in range(numberofpartitions):
            lower = i * step
            upper = (i + 1) * step
            if i == 0:
                condition = f"rating >= {lower} AND rating <= {upper}"
            else:
                condition = f"rating > {lower} AND rating <= {upper}"

            cur.execute(f"""
                INSERT INTO {RANGE_TABLE_PREFIX}{i}
                SELECT userid, movieid, rating
                FROM {ratingstablename}
                WHERE {condition};
            """)

        openconnection.commit()
        print(f"[rangepartition] Completed {numberofpartitions} partitions in {time.time() - start_time:.2f} seconds")

    except Exception as e:
        openconnection.rollback()
        print(f"[rangepartition] Error: {e}")
        raise
    finally:
        cur.close()
```

Hàm rangepartition()

- **Mục đích:** Phân mảnh ngang theo khoảng giá trị *rating*, tạo *numberofpartitions* bảng con *range_part0, range_part1,...* và ghi vào metadata số phân mảnh.
- **Thực thi:**
    - Tạo bảng *partition_metadata* nếu chưa tồn tại.
    - Tính bước *step = 5.0 / numberofpartitions* (*min_rating=0, max_rating=5*).

- Với mỗi *i = 0, 1,..., numberofpartitions - 1*:
  - Xóa (nếu có) và tạo bảng *range_part{i}*.
  - Tính bước *step = 5.0 / numberofpartitions* (*min=0, max=5.0*).
  - Xác định cận dưới *lower*, cận trên *upper* và điều kiện *condition*.
  - Thực hiện chèn dữ liệu từ bảng *ratings* vào bảng *range_part{i}* thỏa mãn điều kiện *condition*.
  - Thêm vào bảng *partition_metadata* với *partiton_type=”range”, partition_count=numberofpartitions*.
  - Commit mọi thay đổi và đóng con trỏ *cur*.

2.3. Hàm count_partitions()

```
287 def count_partitions(type, openconnection: psycopg2.extensions.connection) -> int | None:
288     """
289     Function to count the number of partitions which type is @type.
290     """
291     try:
292         cursor = openconnection.cursor()
293         command = f"""
294             SELECT partition_count
295             FROM partition_metadata
296             WHERE partition_type = '{type}';
297         """
298 
299         cursor.execute(command)
300         return cursor.fetchone()[0]
301     except Exception as e:
302         print (f"[count_partitions] Error: {e}")
303         return None
304     finally:
305         cursor.close()
```

Hàm count_partitions()

- **Mục đích:** Đếm số phân mảnh đang có với kiểu phân mảnh type (range hoặc roundrobin), dựa trên thông tin được lưu trong bảng partition_metadata.
- **Thực thi:**
  - Truy vấn bảng partition_metadata để lấy giá trị của thuộc tính partition_count thỏa mãn điều kiện partition_type = type.

- Lấy kết quả truy vấn đầu tiên và trả về giá trị đếm phân mảnh.
- Nếu có lỗi trong quá trình truy vấn, in thông báo lỗi và trả về None.

2.4. Hàm rangeinsert()

```python
def rangeinsert(_, userid, itemid, rating, openconnection: psycopg2.extensions.connection) -> None:
    """
    Function to insert a new row into the main table and specific partition based on range rating.
    """
    cursor = openconnection.cursor()
    start_time = time.time()

    prefix = "range_part"
    partitions_number = count_partitions(prefix, openconnection)
    print (f"[rangeinsert] Number of partitions: {partitions_number}")

    if not partitions_number:
        cursor.close()
        raise Exception(f"No partitions found with prefix '{prefix}'")

    delta = 5 / partitions_number
    idx = int(rating / delta)

    if rating % delta == 0 and idx:
        idx -= 1
    if idx >= partitions_number:
        idx = partitions_number - 1

    table_name = f"{prefix}{idx}"
    command = f"INSERT INTO {table_name} (userid, movieid, rating) VALUES ('{userid}', '{itemid}', '{rating}');"
    cursor.execute(command)
    print (command)

    openconnection.commit()
    cursor.close()
    print (f"[rangeinsert] Done in {time.time() - start_time:.2f} seconds")
```

Hàm rangeinsert()

- **Mục đích:** Thực hiện chèn một bản ghi mới vào đúng phân mảnh dựa trên giá trị *rating*.
- **Thực thi:**
    - Đếm số phân mảnh *range* từ bảng *partition_metadata* sử dụng hàm *count_partition*.
    - Tính độ rộng mỗi khoảng *delta*.
    - Tính chỉ số bảng mà chúng ta cần chèn dữ liệu mới: int(*rating / delta*).
    - Nếu chỉ số *idx* có giá trị nguyên dương và rơi đúng biên, giảm *idx* đi 1.

- Nếu *idx* vượt quá số phân mảnh, gán *idx = partition_number - 1*.
- Thực hiện chèn dữ liệu mới vào bảng *range_part{idx}*.
- Ghi lại mọi thay đổi và đóng con trỏ *cursor*.

2.5. Hàm roundrobinpartition()

```python
def roundrobinpartition(ratingstablename, numberofpartitions, openconnection):
    if numberofpartitions <= 0:
        raise ValueError("Number of partitions must be positive")

    start_time = time.time()
    try:
        cur = openconnection.cursor()
        create_metadata_table_if_not_exists(cur)
        RROBIN_TABLE_PREFIX = 'rrobin_part'

        # Tạo các bảng phân mảnh
        for i in range(numberofpartitions):
            cur.execute(f"""
                DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i};
                CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (
                    userid INTEGER,
                    movieid INTEGER,
                    rating FLOAT,
                    PRIMARY KEY (userid, movieid)
                );
            """)
            cur.execute(f"""
                INSERT INTO {RROBIN_TABLE_PREFIX}{i}(userid, movieid, rating)
                SELECT userid, movieid, rating
                FROM (
                    SELECT userid, movieid, rating,
                           row_number() over () - 1 as rn
                    FROM {ratingstablename}
                ) t
                WHERE mod(rn, {numberofpartitions}) = {i};
            """)

        command = (f"""
            INSERT INTO partition_metadata (partition_type, partition_count, last_used)
            VALUES ('rrobin', {numberofpartitions}, {time.time()})
        """)

        cur.execute(command)
        print (command)

        openconnection.commit()
        print(f"[roundrobinpartition] Completed {numberofpartitions} partitions in {time.time() - start_time:.2f} seconds")

    except Exception as e:
        openconnection.rollback()
        print(f"[roundrobinpartition] Error: {e}")
        raise
    finally:
        cur.close()
```

Hàm roundrobinpartition()

- **Mục đích:** Phân mảnh ngang theo vòng tròn, chia đều lần lượt mỗi bản ghi qua *numberofpartitions* mảng, lưu lại số phân mảnh vào bảng *partition_metadata*.

- **Thực thi:**
  - Tạo bảng *partition_metadata* nếu chưa tồn tại.
  - Với mỗi *i = 0, 1,..., numberofpartitions - 1*:
    - Xóa (nếu có) và tạo bảng *rrobin_part{i}*.
    - *SELECT* tất cả bản ghi từ bảng *ratings*, duyệt bằng thứ tự *rownum* bắt đầu từ 0.
    - Với mỗi bản ghi thứ *rownum*, tính chỉ số *idx = rownum mod numberofpartitions*, chèn dữ liệu vào mảnh *rrobin_part{idx}*.
    - Thêm vào bảng *partition_metadata* với *partiton_type=”rrobin_part”, partition_count=numberofpartitions.*
    - Ghi lại thay đổi và đóng con trỏ *cur*.

2.6. Hàm roundrobininsert()

```python
def roundrobininsert(ratingstablename, userid, itemid, rating, openconnection):
    try:
        cur = openconnection.cursor()
        RROBIN_TABLE_PREFIX = 'rrobin_part'

        # Kiểm tra số partition
        numberofpartitions = count_partitions('rrobin', openconnection)
        if not numberofpartitions:
            raise ValueError("No round-robin partitions found")

        # Insert vào bảng chính và lấy số hàng trong một query
        cur.execute(f"""
            WITH inserted AS (
                INSERT INTO {ratingstablename} (userid, movieid, rating)
                VALUES (%s, %s, %s)
                RETURNING (SELECT COUNT(*) FROM {ratingstablename})
            )
            SELECT * FROM inserted;
        """, (userid, itemid, rating))

        total_rows = cur.fetchone()[0]
        index = (total_rows - 1) % numberofpartitions
        table_name = f"{RROBIN_TABLE_PREFIX}{index}"

        # Insert vào partition tương ứng
        cur.execute(f"""
            INSERT INTO {table_name} (userid, movieid, rating)
            VALUES (%s, %s, %s);
        """, (userid, itemid, rating))

        openconnection.commit()
        print(f"[roundrobininsert] Successfully inserted into partition {index}")

    except Exception as e:
        openconnection.rollback()
        print(f"[roundrobininsert] Error: {e}")
        raise
    finally:
        cur.close()
```

Hàm roundrobininsert()

- **Mục đích:** Chèn một bản ghi mới theo vòng tròn vào đúng mảnh tiếp theo.
- **Thực thi:**
  - Đếm số phân mảnh *rrobin_part* từ bảng *partition_metadata* sử dụng hàm *count_partition*.
  - Đếm số bản ghi có trong bảng *ratings*, lưu vào biến *total_rows*.
  - Tính chỉ số *index = (total_rows - 1) % numberofpartitions* để xác định chèn dữ liệu vào mảnh nào.
  - Thực hiện chèn dữ liệu mới vào mảnh *rrobin_part{index}*.

- Ghi lại thay đổi và đóng con trỏ *cur*.

PHẦN VI: KIỂM THỬ VÀ ĐÁNH GIÁ

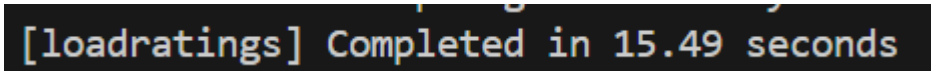
Dưới đây là các trường hợp cần kiểm thử, kèm theo truy vấn kiểm tra và đánh giá về mặt thời gian thực thi.

1. Test case 1: Load dữ liệu vào bảng

- Tiến hành tải dữ liệu từ file ratings.dat vào bảng dữ liệu tương ứng:
  - **Mục tiêu:** kiểm tra số dòng dữ liệu được nạp từ file vào bảng có đủ hay không.
  - **Tiêu chí pass:** số dòng trong bảng sau khi tải dữ liệu bằng với số dòng của file dữ liệu.
- **Truy vấn kiểm tra:**

SELECT COUNT(*) FROM ratings;

  - Truy vấn để lấy tổng số dòng của bảng sau khi tải dữ liệu, và so sánh với số dòng của file.
- **Đánh giá:**
  - Thời gian thực thi: 15.49s



2. Test case 2: Kiểm tra phân mảnh theo khoảng

- Tiến hành phân mảnh ngang dữ liệu bảng đã tải dữ liệu bằng phương pháp phân mảnh theo khoảng.
  - **Mục tiêu:** kiểm tra thuật toán phân mảnh theo khoảng hoạt động đúng hay không.
  - **Tiêu chí pass:** các bảng phân mảnh có kết quả đúng như dự đoán.
- **Truy vấn kiểm tra:**
  - Kiểm tra số bảng phân mảnh được tạo:

SELECT COUNT(*) FROM information_schema.tables

WHERE table_name LIKE 'range_part%';

- Kiểm tra tổng số dòng trong phân mảnh bằng số dòng trong bảng gốc

```
SELECT (
    (SELECT COUNT(*) FROM range_part0) +
    (SELECT COUNT(*) FROM range_part1) +
    (SELECT COUNT(*) FROM range_part2) +
    (SELECT COUNT(*) FROM range_part3)
) = (SELECT COUNT(*) FROM ratings);
```

- Kiểm tra không có bản ghi trùng giữa các mảnh

```
SELECT COUNT(*) FROM (
SELECT userid, movieid, rating, timestamp FROM range_part0
INTERSECT
SELECT userid, movieid, rating, timestamp FROM range_part1
) AS common;
```

- **Thời gian thực hiện:** 34.56s trong trường hợp tạo 5 phân mảnh

```
[rangepartition] Completed 5 partitions in 34.56 seconds
```

3. Test case 3: Kiểm tra thao tác chèn vào phân mảnh sau khi đã phân mảnh bằng thuật toán phân mảnh theo khoảng

- Tiến hành chèn dữ liệu vào bảng chính và bảng phân mảnh
  - **Mục tiêu:** kiểm tra xem dữ liệu có được chèn vào bảng phân mảnh như dự đoán không.
  - **Tiêu chí pass:** dữ liệu được chèn vào bảng phân mảnh như dự đoán.
- **Truy vấn kiểm tra:**

- Cần biết được trước bảng phân mảnh và dữ liệu sẽ được chèn vào, và thực hiện truy vấn tới bảng tương ứng xem dữ liệu mới có tồn tại không, giả sử nằm ở phân mảnh 1.

*SELECT * FROM range_part1*

WHERE userid = 10 AND movieid = 100 AND rating = 2.5

- **Kết quả:**

```
[rangeinsert] Number of partitions: 5
INSERT INTO range_part2 (userid, movieid, rating) VALUES (100, 2, 3);
[rangeinsert] Completed in 0.01 seconds
```

4. Test case 4: Kiểm tra phân mảnh vòng

- Tiến hành phân mảnh ngang dữ liệu bảng đã tải dữ liệu bằng phương pháp phân mảnh vòng
  - **Mục tiêu:** kiểm tra thuật toán phân mảnh vòng hoạt động đúng hay không
  - **Tiêu chí pass:** các bảng phân mảnh có kết quả đúng như dự đoán
- **Truy vấn kiểm tra:**
  - Kiểm tra số bảng phân mảnh được tạo:

SELECT COUNT(*) FROM information_schema.tables

WHERE table_name LIKE 'rrobin_part%';

  - Tổng số dòng của các phân mảnh bằng số dòng của bảng gốc:

SELECT (

(SELECT COUNT(*) FROM rrobin_part0) +

(SELECT COUNT(*) FROM rrobin_part1) +

(SELECT COUNT(*) FROM rrobin_part2) +

(SELECT COUNT(*) FROM rrobin_part3)

) = (SELECT COUNT(*) FROM ratings);

  - Kiểm tra phân phối tương đối đều giữa các phân mảnh, chênh lệch số lượng dòng không quá 1:

SELECT COUNT(*) FROM rrobin_part0;

SELECT COUNT(*) FROM rrobin_part1;

- **Thời gian thực hiện:** 42.86s với trường hợp có 5 phân mảnh.

```
[roundrobinpartition] Completed 5 partitions in 42.86 seconds
```

5. Test case 5: Kiểm tra thao tác chèn vào phân mảnh sau khi đã phân mảnh bằng thuật toán phân mảnh vòng

- Tiến hành chèn dữ liệu vào bảng chính và bảng phân mảnh:
  - Mục tiêu: kiểm tra xem dữ liệu có được chèn vào bảng phân mảnh như dự đoán không.
  - Tiêu chí pass: dữ liệu được chèn vào bảng phân mảnh như dự đoán.
- **Truy vấn kiểm tra:**
  - Cần biết được trước bảng phân mảnh và dữ liệu sẽ được chèn vào, và thực hiện truy vấn tới bảng tương ứng xem dữ liệu mới có tồn tại không, giả sử dự đoán dữ liệu sẽ vào phân mảnh 1.

*SELECT * FROM range_part1*

WHERE userid = 10 AND movieid = 100 AND rating = 2.5

- **Kết quả**

```
[roundrobininsert] Successfully inserted into partition 3
roundrobininsert function pass!
```

Phần kiểm thử đã xây dựng và thực hiện các test case tiêu biểu nhằm đảm bảo tính **đúng đắn**, **toàn vẹn** và **đủ bao phủ** cho hệ thống phân mảnh cơ sở dữ liệu. Các kiểm thử đã được thiết kế để xác minh:

- Việc **tải dữ liệu** từ tệp đầu vào vào bảng cơ sở dữ liệu diễn ra đầy đủ và chính xác.

- Hai thuật toán phân mảnh chính là **Range Partitioning** và **Round Robin Partitioning** hoạt động đúng theo logic mong đợi:
    - Các bảng phân mảnh được tạo đúng số lượng.
    - Dữ liệu được phân phối chính xác, không trùng lặp và có thể tái cấu trúc lại bảng gốc.
- Các hàm **insert** (chèn dữ liệu) vào các bảng phân mảnh hoạt động chính xác, theo đúng nguyên tắc phân mảnh đã chọn.

Các truy vấn SQL kiểm tra đi kèm đã giúp xác thực trực tiếp các kết quả của mỗi test case, đồng thời hỗ trợ việc phát hiện lỗi nếu có sai sót xảy ra.

Thông qua quá trình kiểm thử này, ta có thể tự tin rằng hệ thống phân mảnh được triển khai đáp ứng đầy đủ các yêu cầu về mặt **chức năng**, đồng thời sẵn sàng để tích hợp vào các ứng dụng thực tế đòi hỏi khả năng **chia nhỏ dữ liệu lớn** một cách hiệu quả.

PHẦN VII: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

1. Kết quả đạt được:

1.1. Thành công mô phỏng 2 cơ chế phân mảnh

- Một trong những kết quả nổi bật của quá trình triển khai là **nhóm đã mô phỏng thành công hai kỹ thuật phân mảnh phổ biến trong hệ quản trị cơ sở dữ liệu**, bao gồm:

a. Phân mảnh theo Range (Range Partitioning)

- Hệ thống cho phép chia dữ liệu theo khoảng giá trị cụ thể của thuộc tính rating, ví dụ như:
  - Bản ghi có rating từ 0 đến dưới 2 được đưa vào range_part0
  - Từ 2 đến dưới 4 vào range_part1, v.v...

  Việc xác định điều kiện chia range, số lượng phân mảnh, cũng như quá trình kiểm tra xem mỗi bản ghi có nằm đúng trong khoảng tương ứng đã được thực hiện chính xác.

b. Phân mảnh theo Round Robin (Round Robin Partitioning

- Đối với Round Robin, hệ thống đã thực hiện phân phối bản ghi theo vòng lặp lần lượt giữa các bảng phân mảnh. Điều này đảm bảo:
  - Mỗi bảng có số lượng bản ghi gần như đồng đều.
  - Không có bản ghi nào bị phân bổ trùng lặp.

1.2. Đảm bảo tính chính xác của hệ thống phân mảnh

- Thông qua bộ test case tiêu biểu được thiết kế xoay quanh các chức năng cốt lõi, bao gồm: tải dữ liệu, phân mảnh dữ liệu (Range và Round Robin), và chèn dữ liệu mới vào các bảng phân mảnh — nhóm đã xác minh được rằng:

- Tất cả các bảng phân mảnh được tạo đúng số lượng và tuân thủ đúng quy ước đặt tên.
- Dữ liệu sau khi phân mảnh **không bị mất mát**, **không bị trùng lặp**, và có thể kết hợp lại chính xác như dữ liệu gốc.
- Với phân mảnh theo Range, mỗi bản ghi đều được đặt đúng theo điều kiện về giá trị rating.
- Với phân mảnh Round Robin, dữ liệu được chia đều và tuần tự giữa các bảng phân mảnh.
- Các thao tác **INSERT** cho từng loại phân mảnh đã hoạt động chính xác, ghi nhận được dữ liệu ở đúng bảng cần thiết mà không gây lỗi hệ thống hay sai lệch dữ liệu.

1.3. Tự động hóa quá trình kiểm thử

Các test case đã được thiết kế thành một tập lệnh Python dễ chạy, kiểm thử đầu-cuối (end-to-end), giúp:

- Kiểm tra được nhiều khía cạnh chỉ với một lần thực thi.
- Dễ dàng tái sử dụng, mở rộng để kiểm thử với dữ liệu mới hoặc logic mới.
- Có thể dùng làm một công cụ kiểm thử hồi quy khi hệ thống được cập nhật.

1.4. Hiểu rõ hơn về kỹ thuật phân mảnh và hệ quản trị cơ sở dữ liệu

- Thông qua việc triển khai và kiểm thử hệ thống, nhóm đã hiểu sâu hơn về các kỹ thuật phân mảnh dữ liệu (data partitioning), cách ánh xạ chúng vào hệ quản trị cơ sở dữ liệu PostgreSQL, và xử lý các thao tác như INSERT, SELECT, COUNT, kiểm tra tồn tại, v.v… khi có nhiều bảng con.

2. Hướng mở rộng

2.1. Tự động cân bằng tải khi thêm/xóa phân mảnh

- Hiện tại, hệ thống yêu cầu xác định trước số lượng phân mảnh, và khi cần mở rộng hoặc thu hẹp hệ thống, người dùng phải can thiệp thủ công vào cấu trúc phân mảnh. Hướng mở rộng này nhằm:

  - **Tự động phát hiện mất cân bằng** (ví dụ: một phân mảnh có quá nhiều bản ghi so với các phân mảnh khác).
  - Khi thêm phân mảnh mới, hệ thống tự **phân phối lại dữ liệu** một cách tối ưu để đảm bảo phân phối đồng đều, giảm hiện tượng nghẽn dữ liệu hoặc điểm nghẽn truy vấn.
  - Khi xóa phân mảnh, các bản ghi trong phân mảnh đó sẽ được tự động di chuyển sang các phân mảnh còn lại.
  - Cập nhật các chỉ mục và ánh xạ liên quan để đảm bảo tính nhất quán và hiệu năng truy vấn.
  - Việc thực hiện yêu cầu này đòi hỏi bổ sung thêm các **thuật toán phân tích thống kê** và **cơ chế di chuyển dữ liệu động**, hướng đến một hệ thống **partitioning-aware optimizer**.

2.2. Tích hợp cơ chế replication (nhân bản dữ liệu)

- Để đảm bảo **tính sẵn sàng cao** và **khả năng khôi phục sau lỗi**, cơ chế replication có thể được tích hợp để sao lưu dữ liệu giữa các phân mảnh hoặc giữa các node khác nhau. Mục tiêu của hướng mở rộng này bao gồm:

- **Replication theo phân mảnh**: mỗi phân mảnh được nhân bản sang một hoặc nhiều bản sao, đặt tại các nút khác nhau trong hệ thống phân tán.
- **Tự động đồng bộ bản ghi** giữa các bản sao theo thời gian thực hoặc theo lịch.
- Cung cấp khả năng **chuyển đổi dự phòng (failover)** trong trường hợp node chính gặp sự cố.
- Có thể hỗ trợ các mô hình như **master-slave** hoặc **multi-master replication**, tùy theo nhu cầu.

- Việc tích hợp replication không chỉ nâng cao độ tin cậy của hệ thống mà còn hỗ trợ mở rộng theo chiều ngang, phù hợp với các ứng dụng quy mô lớn.

2.3. Xây dựng giao diện quản lý trực quan

- Hiện tại, quá trình thao tác và kiểm thử chủ yếu thông qua dòng lệnh và file Python. Một giao diện đồ họa trực quan (GUI) sẽ giúp:
  - **Quản lý và giám sát phân mảnh dễ dàng hơn**: hiển thị biểu đồ số lượng bản ghi trong từng phân mảnh, trạng thái hoạt động, tốc độ truy vấn, v.v.
  - **Thao tác tạo, xóa, điều chỉnh phân mảnh bằng giao diện** thay vì viết lệnh thủ công.
  - **Xem log truy vấn**, hiển thị các cảnh báo về phân mảnh mất cân bằng hoặc lỗi đồng bộ hóa.
  - Có thể xây dựng bằng các framework web phổ biến như **React, Vue hoặc Django Admin**, kết nối với backend Python để thao tác trên hệ thống cơ sở dữ liệu.
- Giao diện này không chỉ hỗ trợ kiểm thử và demo hiệu quả hơn mà còn là bước đệm hướng tới triển khai thực tế.

TÀI LIỆU THAM KHẢO

1. "Distributed Database Systems" – *Özsu, M. Tamer & Valduriez, Patrick*

2. "Database System Concepts" (7th Edition) – *Silberschatz, Korth & Sudarshan*

3. Psycopg2 Tutorial (PostgreSQL Wiki)