

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



BÁO CÁO BÀI TẬP LỚN

HỌC PHẦN: CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên hướng dẫn	: Kim Ngọc Bách
Nhóm lớp học	: 09
Nhóm bài tập lớn	: 19
Thành viên nhóm	: Trần Quang Huy - B22DCCN397
	Đinh Hữu Minh - B22DCCN526
	Hoàng Minh Hoà - B22DCCN325

HÀ NỘI 06/2025

PHÂN CÔNG NHIỆM VỤ NHÓM THỰC HIỆN

TT	Công việc / Nhiệm vụ	SV thực hiện
1	- Cài đặt hàm LoadRatings() và RoundRobin_Insert()	Trần Quang Huy
2	- Cài đặt hàm: Range_Partition()	Đinh Hữu Minh
3	- Cài đặt hàm: RoundRobin_Partition() và Range_Insert()	Hoàng Minh Hoà

MỤC LỤC

PHÂN CÔNG NHIỆM VỤ NHÓM THỰC HIỆN	2
MỤC LỤC	3
DANH MỤC CÁC HÌNH VẼ	5
CHƯƠNG 1. TỔNG QUAN BÀI TOÁN	6
1.1 Bối cảnh và mục tiêu	6
1.1.1 Bối cảnh	6
1.1.2 Mục tiêu của bài toán	6
1.2 Dữ liệu đầu vào	6
1.2.1 Cấu trúc dữ liệu	6
1.2.2 Mô tả các trường dữ liệu	7
1.2.3 Đặc điểm dữ liệu	7
1.2.4 Yêu cầu xử lý	7
CHƯƠNG 2. Cơ sở lý thuyết	8
2.1 Phân mảnh dữ liệu trong cơ sở dữ liệu	8
2.1.1 Định nghĩa	8
2.1.2 Các loại phân mảnh	8
2.1.3 Lợi ích	8
2.2 Phân mảnh ngang	9
2.2.1 Phân mảnh theo khoảng (Range Partitioning) :	9
2.2.2 Phân mảnh vòng tròn (Round-Robin Partitioning):	9
2.3 Vai trò của hệ quản trị cơ sở dữ liệu quan hệ	10
CHƯƠNG 3. PHƯƠNG PHÁP GIẢI QUYẾT	11
3.1 Tổng quan quy trình	11
3.2 Cài đặt môi trường	11
3.2.1 Cấu hình phần mềm	11
3.2.2 Tải dữ liệu MovieLens	13
3.3 Triển khai các hàm Python	13
3.3.1 Hàm LoadRatings()	13
3.3.2 Hàm Range_Partition()	16

3.3.3 Hàm RoundRobin_Partition():	18
3.3.4 Hàm RoundRobin_Insert()	21
3.3.5 Hàm Range_Insert()	24
CHƯƠNG 4. Kiểm tra và đánh giá	26
4.1 Dữ liệu test	26
4.2 Kết quả kiểm tra	26
4.3 Đánh giá kết quả:.....	30
KẾT LUẬN	31
Phụ lục	32
Phụ lục A:.....	32
Phụ lục B:	33
Phụ lục C	35
Phụ lục D:.....	36
Phụ lục E:	37
Phụ lục F:	Error! Bookmark not defined.

DANH MỤC CÁC HÌNH VẼ

Hình 2: Đoạn code kiểm tra đầu vào và tính toán khoảng phân mảnh trong hàm Range_Partition.	17
Hình 3 Đoạn code tạo bảng phân mảnh trong hàm Range_Partition.	17
Hình 4 Đoạn code phân phối dữ liệu vào phân mảnh đầu tiên trong hàm Range_Partition.	18
Hình 5 Đoạn code tạo bảng phân mảnh trong hàm RoundRobin_Partition.	20
Hình 6 Đoạn code đếm tổng số bản ghi và lưu chỉ số vòng tròn trong hàm RoundRobin_Partition.	20
Hình 7 Đoạn code phân phối dữ liệu theo phương pháp vòng tròn trong hàm RoundRobin_Partition.	21
Hình 8 Đoạn code kiểm tra và xác định chỉ số phân mảnh trong hàm RoundRobin_Insert.	23
Hình 9: Đoạn code chèn bản ghi vào bảng Ratings và bảng phân mảnh trong hàm RoundRobin_Insert.	23
Hình 10 Đoạn code cập nhật chỉ số vòng tròn trong hàm RoundRobin_Insert.	23
Hình 11:Đoạn code chèn bản ghi vào bảng Ratings trong hàm Range_Insert.	25
Hình 12 Đoạn code tính toán chỉ số phân mảnh trong hàm Range_Insert.	25
Hình 13 Đoạn code chèn bản ghi vào bảng phân mảnh trong hàm Range_Insert.	25

CHƯƠNG 1. TỔNG QUAN BÀI TOÁN

1.1 Bối cảnh và mục tiêu

1.1.1 Bối cảnh

Trong thời đại big data hiện nay, việc quản lý và xử lý khối lượng dữ liệu khổng lồ đã trở thành một thách thức lớn đối với các hệ thống cơ sở dữ liệu truyền thống. Một cơ sở dữ liệu tập trung không thể đáp ứng được các yêu cầu về:

- **Hiệu suất:** Khi dữ liệu tăng lên hàng triệu, hàng tỷ bản ghi, việc truy vấn và cập nhật trở nên chậm chạp
- **Khả năng mở rộng:** Khó khăn trong việc tăng cường phần cứng và mở rộng hệ thống
- **Độ tin cậy:** Một điểm lỗi có thể làm sập toàn bộ hệ thống
- **Tính sẵn sàng:** Khó đảm bảo hoạt động liên tục 24/7

Cơ sở dữ liệu phân tán với kỹ thuật phân mảnh dữ liệu (data fragmentation/partitioning) ra đời như một giải pháp hiệu quả để giải quyết những vấn đề trên.

1.1.2 Mục tiêu của bài toán

Bài tập lớn này nhằm mục đích:

Mục tiêu chính:

- Hiểu và thực hiện các kỹ thuật phân mảnh dữ liệu cơ bản trong cơ sở dữ liệu phân tán
- Cài đặt hai phương pháp phân mảnh ngang: Range Partitioning và Round-Robin Partitioning
- Xây dựng các thao tác cơ bản trên dữ liệu đã được phân mảnh

Mục tiêu cụ thể:

1. **Tải dữ liệu:** Cài đặt chức năng load dữ liệu từ file vào bảng chính
2. **Phân mảnh theo khoảng:** Chia dữ liệu rating theo các khoảng giá trị
3. **Phân mảnh vòng tròn:** Phân bổ dữ liệu đều các partition theo thứ tự
4. **Chèn dữ liệu:** Thực hiện thao tác insert vào cả bảng chính và partition tương ứng
5. **Kiểm tra tính đúng đắn:** Đảm bảo các tính chất completeness, disjointness và reconstruction

Đối tượng áp dụng:

- Hệ thống rating phim với dữ liệu gồm UserID, MovieID và Rating
- Sử dụng PostgreSQL làm hệ quản trị cơ sở dữ liệu
- Ngôn ngữ lập trình Python với thư viện psycopg2

1.2 Dữ liệu đầu vào

1.2.1 Cấu trúc dữ liệu

Bài toán sử dụng dữ liệu rating phim với định dạng chuẩn MovieLens:

Định dạng file dữ liệu:

UserID::MovieID::Rating::Timestamp

Ví dụ dữ liệu thực tế:

1::122::5::838985046

1::185::4.5::838983525

1::231::4::838983392

1::292::3.5::838983421

1::316::3::838983392

1.2.2 Mô tả các trường dữ liệu

Trường	Kiểu dữ liệu	Mô tả	Ràng buộc
UserID	INTEGER	Mã định danh người dùng	> 0
MovieID	INTEGER	Mã định danh phim	> 0
Rating	FLOAT	Điểm đánh giá	$0.0 \leq \text{Rating} \leq 5.0$
Timestamp	INTEGER	Thời gian đánh giá (Unix timestamp)	

1.2.3 Đặc điểm dữ liệu

- **Kích thước:** File test chứa 20 bản ghi, có thể mở rộng lên hàng triệu bản ghi
- **Phân bố Rating:** Từ 0.0 đến 5.0 với bước 0.5
- **Mô hình:** Quan hệ nhiều-nhiều giữa User và Movie
- **Tính chất:** Dữ liệu thời gian thực, có thể được cập nhật liên tục

1.2.4 Yêu cầu xử lý

1. **Parsing:** Tách các trường từ định dạng "::" delimiter
2. **Validation:** Kiểm tra tính hợp lệ của dữ liệu
3. **Transformation:** Chuyển đổi kiểu dữ liệu phù hợp
4. **Loading:** Nạp vào bảng chính trong cơ sở dữ liệu

mang tính ứng dụng cao, hỗ trợ người dùng trong việc số hóa và quản lý thông tin hiệu quả.

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1 Phân mảnh dữ liệu trong cơ sở dữ liệu

Phân mảnh (Fragmentation) là một khái niệm quan trọng trong thiết kế cơ sở dữ liệu (CSDL) phân tán. Đây là quá trình chia một CSDL hoặc một quan hệ (relation) thành các đơn vị logic nhỏ hơn, gọi là các mảnh (fragments). Các mảnh này sau đó có thể được phân phối và lưu trữ tại các địa điểm khác nhau trong một hệ thống CSDL phân tán.

2.1.1 Định nghĩa

Phân mảnh dữ liệu là hành động chia một quan hệ toàn cục (global relation) thành các tập hợp con (fragments) nhỏ hơn. Mỗi mảnh dữ liệu có thể được xử lý độc lập, cho phép tăng cường tính song song trong quá trình xử lý truy vấn. Trong CSDL phân tán, mục tiêu chính của phân mảnh là tăng cường tính cục bộ dữ liệu (data locality) và tính song song để cải thiện hiệu suất.

2.1.2 Các loại phân mảnh

Có ba loại phân mảnh chính trong CSDL quan hệ:

- **Phân mảnh ngang (Horizontal Fragmentation):** Chia một quan hệ thành các tập con của các bộ (rows/tuples) dựa trên một vị từ (selection predicate). Mỗi mảnh ngang chứa các bộ thỏa mãn một điều kiện lựa chọn cụ thể.
Ví dụ: Quan hệ PROJ có thể được phân mảnh ngang thành PROJ1 (chứa các dự án có ngân sách < 200,000) và PROJ2 (chứa các dự án có ngân sách \geq 200,000).
- **Phân mảnh dọc (Vertical Fragmentation):** Chia một quan hệ thành các tập con của các thuộc tính (columns). Mỗi mảnh dọc bao gồm một tập hợp các thuộc tính của quan hệ gốc và khóa chính (primary key) của quan hệ đó để đảm bảo khả năng tái tạo lại quan hệ gốc.
Ví dụ (từ slide 38): Quan hệ PROJ có thể được phân mảnh dọc thành PROJ1 (chứa PNO, PNAME, BUDGET) và PROJ2 (chứa PNO, LOC).
- **Phân mảnh hỗn hợp (Hybrid Fragmentation):** Là sự kết hợp của phân mảnh ngang và phân mảnh dọc, trong đó một quan hệ có thể được phân mảnh ngang trước, sau đó mỗi mảnh ngang lại được phân mảnh dọc, hoặc ngược lại.

2.1.3 Lợi ích

Việc phân mảnh dữ liệu mang lại nhiều lợi ích quan trọng cho hệ thống CSDL phân tán và song song:

Cải thiện hiệu suất (Improved Performance):

Tính cục bộ dữ liệu (Data Locality): Dữ liệu được lưu trữ gần nơi nó được sử dụng nhất, giảm đáng kể chi phí truyền thông và độ trễ truy cập dữ liệu từ xa.

Tính song song (Parallelism): Cho phép nhiều truy vấn (interquery parallelism) hoặc nhiều phần của cùng một truy vấn (intraquery parallelism) được thực thi đồng thời trên các mảnh khác nhau, tăng thông lượng và giảm thời gian phản hồi.

Cân bằng tải :

Phân phối công việc đều hơn giữa các nút xử lý, tránh tình trạng quá tải ở một số nút và giúp hệ thống hoạt động hiệu quả hơn.

Tăng tính sẵn sàng:

Khi dữ liệu được sao chép và phân mảnh, nếu một nút bị lỗi, dữ liệu vẫn có thể được truy cập từ các bản sao ở các nút khác, đảm bảo tính liên tục của dịch vụ.

Dễ quản lý hơn:

Việc quản lý các mảnh dữ liệu nhỏ hơn thường đơn giản hơn so với quản lý toàn bộ CSDL lớn.

Khả năng mở rộng:

Cho phép hệ thống dễ dàng mở rộng bằng cách thêm các nút mới và phân phối dữ liệu trên chúng, đáp ứng nhu cầu tăng trưởng về kích thước CSDL và khối lượng công việc.

2.2 Phân mảnh ngang

Phân mảnh ngang là phương pháp chia một quan hệ thành các tập hợp con của các bộ (hàng) dựa trên các điều kiện chọn. Có hai phương pháp phổ biến để thực hiện phân mảnh ngang:

2.2.1 Phân mảnh theo khoảng (Range Partitioning) :

Mô tả: Trong phương pháp này, các bộ dữ liệu được phân phối dựa trên các khoảng giá trị của một thuộc tính cụ thể. Mỗi mảnh sẽ chứa các bộ có giá trị của thuộc tính đó nằm trong một khoảng xác định.

Ưu điểm: Phương pháp này rất hiệu quả cho các truy vấn theo khoảng (Range Queries) vì tất cả dữ liệu trong một khoảng nhất định sẽ nằm trên cùng một nút hoặc một tập hợp các nút liên kề. Nó cũng có thể xử lý tốt sự phân phối dữ liệu không đồng đều (non-uniform data distributions) bằng cách điều chỉnh các khoảng giá trị cho phù hợp.

Ví dụ : Trong một hệ thống cơ sở dữ liệu phân tán, các bộ dữ liệu có giá trị thuộc tính nằm trong khoảng 'a-g' được lưu trữ trên một máy chủ, trong khi các bộ có giá trị 'h-m' nằm trên một máy chủ khác, và cứ thế tiếp tục.

2.2.2 Phân mảnh vòng tròn (Round-Robin Partitioning):

Mô tả: Đây là phương pháp phân mảnh đơn giản nhất, trong đó các bộ dữ liệu được gán cho các mảnh theo thứ tự chèn (insertion order) một cách luân phiên, tức là bộ thứ i được gán cho mảnh $(i \bmod n)$ với n là số lượng mảnh.

Ưu điểm: Đảm bảo cân bằng tải hoàn hảo một cách tự động giữa các nút và rất phù hợp cho các truy vấn quét toàn bộ quan hệ (full scan queries) vì tất cả các nút đều hoạt động song song.

Hạn chế: Không hiệu quả cho các truy vấn truy cập trực tiếp dựa trên vị từ lựa chọn (selection predicate), vì để tìm một bộ dữ liệu cụ thể, có thể cần phải truy cập tất cả các mảnh.

Ví dụ: Dữ liệu được chia và phân phối đều đặn giữa các máy chủ (tương trưng bằng các hình trụ và khối vuông nhỏ) theo thứ tự chèn. Nếu có 3 máy chủ, hàng thứ 1 sẽ vào máy chủ 1, hàng thứ 2 vào máy chủ 2, hàng thứ 3 vào máy chủ 3, hàng thứ 4 lại vào máy chủ 1, v.v.

2.3 Vai trò của hệ quản trị cơ sở dữ liệu quan hệ

Trong hệ cơ sở dữ liệu phân tán (DDBMS), hệ quản trị cơ sở dữ liệu quan hệ (RDBMS) đóng vai trò trung tâm, mở rộng chức năng truyền thống để quản lý môi trường phân tán phức tạp.

Quản lý trong suốt dữ liệu phân tán và sao chép:

- **Độc lập dữ liệu:** Ứng dụng không bị ảnh hưởng bởi thay đổi schema hoặc tổ chức vật lý.
- **Trong suốt mạng:** Ẩn chi tiết mạng, cho phép truy cập dữ liệu mà không cần biết vị trí (location transparency).
- **Trong suốt phân mảnh:** Người dùng truy vấn quan hệ toàn cục mà không cần biết cách phân mảnh, hệ thống tự ánh xạ truy vấn.
- **Trong suốt sao chép:** Ẩn sự tồn tại của bản sao, thao tác như chỉ có một bản dữ liệu duy nhất.

Đảm bảo tính tin cậy qua giao dịch phân tán:

- **Thuộc tính ACID:** Đảm bảo giao dịch phân tán nguyên tử, nhất quán, cô lập, và bền vững, bất chấp lỗi hệ thống.
- **Kiểm soát đồng thời phân tán:** Đồng bộ hóa truy cập, ngăn deadlock, duy trì toàn vẹn dữ liệu.
- **Giao thức cam kết phân tán:** Sử dụng Two-Phase Commit (2PC) để đảm bảo đồng thuận cam kết hoặc hủy bỏ.

Cải thiện hiệu suất:

- **Tính cục bộ:** Lưu trữ dữ liệu gần nơi xử lý, giảm thời gian truy cập từ xa.
- **Tính song song:** Thực thi truy vấn song song (interquery và intraquery parallelism), cải thiện thời gian phản hồi

CHƯƠNG 3. PHƯƠNG PHÁP GIẢI QUYẾT

3.1 Tổng quan quy trình

Để giải quyết bài toán phân mảnh dữ liệu trong cơ sở dữ liệu quan hệ, chúng em đã thực hiện theo quy trình sau:

1. Phân tích yêu cầu: Nghiên cứu kỹ đề bài về các phương pháp phân mảnh dữ liệu, bao gồm phân mảnh theo khoảng (range partitioning) và phân mảnh vòng tròn (round-robin partitioning).

2. Thiết lập môi trường phát triển: Cài đặt PostgreSQL và các công cụ phát triển Python cần thiết.

3. Tìm hiểu cấu trúc code hiện có: Phân tích các file có sẵn, đặc biệt là Interface.py, để hiểu rõ cách thức hoạt động hiện tại của các hàm.

4. Phân tích hiệu suất: Xác định điểm yếu và cơ hội tối ưu hóa trong các hàm hiện có, đặc biệt là hàm `rangepartition()`.

5. Cải tiến thuật toán: Thiết kế và triển khai các thuật toán hiệu quả hơn cho việc phân mảnh dữ liệu, tập trung vào các kỹ thuật như:

Giảm số lần truy cập đĩa

Tối ưu hóa truy vấn SQL

Tận dụng chức năng song song và lập chỉ mục

Xử lý dữ liệu theo batch

6. Triển khai cải tiến: Xây dựng phiên bản cải tiến (Interface_new.py) với các tối ưu hóa cho tất cả các hàm chính.

7. Kiểm thử: Sử dụng bộ dữ liệu thử nghiệm và công cụ kiểm thử để đánh giá tính chính xác và hiệu suất của phiên bản cải tiến.

8. Đánh giá và điều chỉnh: So sánh hiệu suất giữa phiên bản cũ và phiên bản cải tiến, thực hiện các điều chỉnh bổ sung khi cần thiết.

3.2 Cài đặt môi trường

3.2.1 Cấu hình phần mềm

Để triển khai thành công bài tập, chúng em đã thiết lập môi trường phát triển với máy tính cá nhân (không dùng máy ảo) với các thành phần sau:

Hệ điều hành: Windows 11

Python: Phiên bản: Python 3.12.0

```
PS C:\Users\ADMIN> python --version
Python 3.12.0
```

Các thư viện cần thiết:

- psycopg2 (để kết nối với PostgreSQL)
- logging (ghi lại nhật kí hoạt động, kiểm tra và gỡ lỗi trong quá trình phát triển)
- io/StringIO (để xử lý dữ liệu đầu vào hiệu quả)

```
import psycopg2
from psycopg2.extensions import AsIs
import logging
import os
from io import StringIO
```

PostgreSQL: Phiên bản: PostgreSQL 15.x

```
PS C:\Users\ADMIN> & "C:\Program Files\PostgreSQL\17\bin\postgres.exe" -V
postgres (PostgreSQL) 17.5
```

Cấu hình cơ bản:

- User: postgres
- Password: 1234
- Host: localhost
- Cơ sở dữ liệu mặc định: postgres
- Cơ sở dữ liệu bài tập: dds_assgn1

Công cụ phát triển:

- Môi trường phát triển tích hợp (IDE): Visual Studio Code
- Terminal: PowerShell
(C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe)

3.2.2 Tải dữ liệu MovieLens

Quá trình thu thập và chuẩn bị dữ liệu đầu vào gồm các bước sau:

Tải xuống bộ dữ liệu:

- Truy cập trang web: <http://files.grouplens.org/datasets/movielens/ml-10m.zip>
- Tải về và giải nén tệp zip để lấy tệp ratings.dat

Cấu trúc dữ liệu:

- Mỗi dòng trong tệp ratings.dat chứa thông tin về một đánh giá theo định dạng: UserID::MovieID::Rating::Timestamp

- Ví dụ:

1::122::5::838985046

1::185::4.5::838983525

1::231::4::838983392 Chuẩn bị dữ liệu kiểm thử:

Chuẩn bị dữ liệu kiểm thử:

- Sử dụng tệp test_data.dat chứa 20 dòng dữ liệu được trích xuất từ ratings.dat để kiểm thử nhanh
- Các giá trị rating trong test_data.dat đa dạng từ 0 đến 5, bao gồm cả các giá trị nửa sao (0.5, 1.5, v.v.)

Kiểm tra tính hợp lệ của dữ liệu:

- Đảm bảo dữ liệu đầu vào không chứa giá trị null hoặc không hợp lệ
- Xác nhận phạm vi của giá trị Rating nằm trong khoảng từ 0 đến 5

Việc thiết lập môi trường và chuẩn bị dữ liệu đầy đủ này giúp đảm bảo quá trình phát triển và kiểm thử diễn ra suôn sẻ, đồng thời cung cấp nền tảng vững chắc cho việc triển khai các thuật toán phân mảnh dữ liệu.

3.3 Triển khai các hàm Python

3.3.1 Hàm LoadRatings()

Hàm LoadRatings() đọc dữ liệu từ tệp ratings.dat và lưu vào bảng Ratings trong PostgreSQL. Hàm nhận đầu vào gồm đường dẫn tệp (ratingsfilepath), tên bảng (ratingstablename), và kết nối cơ sở dữ liệu (openconnection). Bảng Ratings có schema: UserID (int), MovieID (int), Rating (float), bỏ qua cột Timestamp theo yêu cầu đề bài.

Quy trình triển khai:

1. Tạo cơ sở dữ liệu.

```
create_db(DATABASE_NAME)
```

2. Mở kết nối và tạo con trỏ

- Sử dụng kết nối được truyền vào (openconnection) để tạo con trỏ cur cho việc thực thi các lệnh SQL.

```
conn = openconnection
cur = conn.cursor()
```

3. Tạo lại bảng đích.

- Xóa bảng ratingtablename nếu đã tồn tại (dùng DROP TABLE IF EXISTS).
- Tạo bảng mới với lược đồ: userid (int), movieid (int), rating (float).

```
# Tạo bảng đích trực tiếp với cấu trúc cuối cùng
cur.execute(f"""
    DROP TABLE IF EXISTS {ratingtablename};
    CREATE TABLE {ratingtablename} (
        userid INTEGER,
        movieid INTEGER,
        rating FLOAT
    );
""")
```

Lý do:

- + Xóa bảng cũ đảm bảo bắt đầu với trạng thái sạch, tránh lỗi nếu bảng đã tồn tại.
- + Lược đồ khớp chính xác với yêu cầu của bài tập (bỏ qua Timestamp vì không được yêu cầu)
- + Sử dụng f-string để chèn tên bảng một cách an toàn

4. Chuẩn bị buffer và biến đếm

- Sử dụng StringIO để mô phỏng một file trong bộ nhớ RAM. => giúp truyền dữ liệu cho PostgreSQL thông qua lệnh COPY mà không cần ghi dữ liệu ra file thật => nhanh hơn đáng kể
- batch_size giúp giảm số lần giao tiếp với cơ sở dữ liệu => tăng hiệu suất.

```
# Sử dụng COPY command để tải dữ liệu vào bảng - cách nhanh nhất
batch_size = 500_000 # Batch lớn hơn để giảm số lần gọi DB
buffer = StringIO()
count = 0
```

5. Đọc file và xử lý từng dòng:

```
# Xử lý từng dòng và định dạng lại để COPY
with open(ratingsfilepath, 'r') as f:
    for line in f:
        # Tách và định dạng lại dữ liệu
        parts = line.strip().split('::')
        if len(parts) >= 3:
            buffer.write(f"{parts[0]}\t{parts[1]}\t{parts[2]}\n")
            count += 1
```

- Mỗi dòng được tách bằng dấu :: (chuẩn file ratings từ MovieLens).
- Chỉ ghi vào buffer nếu dòng hợp lệ (có ít nhất 3 phần).
- Format lại dữ liệu thành định dạng phù hợp với COPY (các cột cách nhau bằng \t).

6. Chèn dữ liệu theo lô

- Khi số dòng trong buffer đạt batch_size, thực hiện lệnh COPY để chèn dữ liệu vào bảng.
- Đặt con trỏ buffer về đầu (seek(0)), chèn dữ liệu, sau đó xóa buffer (truncate(0)) và đặt lại con trỏ.

```
if count % batch_size == 0:
    buffer.seek(0)
    cur.copy_expert(
        f"COPY {ratingtablename} (userid, movieid, rating) FROM STDIN WITH DELIMITER E'\t'",
        buffer
    )
    buffer.truncate(0)
    buffer.seek(0)
```

- Lý do:
 - + Lệnh COPY là cách nhanh nhất để tải dữ liệu lớn vào PostgreSQL, vượt trội so với INSERT.
 - + copy_expert cho phép chỉ định định dạng đầu vào (tab làm dấu phân cách).
 - + Xóa buffer sau mỗi lô tránh tràn bộ nhớ.

7. Xử lý phần dữ liệu còn lại

- Nếu buffer vẫn chứa dữ liệu (số dòng không chia hết cho batch_size), thực hiện COPY cho phần còn lại.
- Đảm bảo tất cả dữ liệu được chèn, kể cả lô cuối không đủ batch_size.

```
# Xử lý phần còn lại
if buffer.tell() > 0:
    buffer.seek(0)
    cur.copy_expert(
        f"COPY {ratingtablename} (userid, movieid, rating) FROM STDIN WITH DELIMITER E'\t'",
        buffer
    )
```

Kết quả: Hàm tải 10 triệu bản ghi trên máy ảo Windows. Kiểm tra cho thấy dữ liệu lưu đúng schema, không có lỗi. Mã nguồn đầy đủ được trình bày trong Phụ lục A

3.3.2 Hàm *Range_Partition()*

Hàm *Range_Partition()* phân mảnh ngang bảng *Ratings* thành *N* bảng con dựa trên khoảng giá trị đều của thuộc tính *Rating* trong PostgreSQL. Hàm nhận đầu vào gồm tên bảng gốc (*ratingstablename*), số lượng phân mảnh (*numberofpartitions*), và kết nối cơ sở dữ liệu (*openconnection*). Các bảng phân mảnh được đặt tên theo định dạng *range_part0*, *range_part1*, ..., *range_part{N-1}*, với giá trị *Rating* chia đều trong khoảng $[0, 5]$ (ví dụ: khi $N=3$, các khoảng là $[0, 1.67]$, $(1.67, 3.34]$, $(3.34, 5]$).

Quy trình triển khai:

1. **Kiểm tra đầu vào:** Đảm bảo *numberofpartitions* là số nguyên dương, ném ngoại lệ nếu không hợp lệ (Hình 2).
2. **Tính toán khoảng phân mảnh:** Chia khoảng $[0, 5]$ thành *N* phần đều, với độ dài mỗi khoảng là $5.0/N$ (biến *interval*).
3. **Tạo bảng phân mảnh:** Xóa các bảng phân mảnh cũ (nếu tồn tại) bằng lệnh *DROP TABLE IF EXISTS* và tạo *N* bảng mới với lược đồ giống bảng *Ratings* (*userid*: int, *movieid*: int, *rating*: float) (Hình 3).
4. **Phân phối dữ liệu:** Sử dụng truy vấn SQL với điều kiện *WHERE* để chuyển dữ liệu từ bảng *Ratings* vào các bảng phân mảnh:
 - Phân mảnh đầu tiên (*range_part0*) chứa các bản ghi với $\text{rating} \geq 0 \text{ AND } \text{rating} \leq \text{interval}$ (bao gồm $\text{rating} = 0$).
 - Các phân mảnh giữa (*range_part1* đến *range_part{N-2}*) chứa bản ghi với $\text{rating} > \text{min_range} \text{ AND } \text{rating} \leq \text{max_range}$.
 - Phân mảnh cuối cùng (*range_part{N-1}*) chứa bản ghi với $\text{rating} > \text{min_range} \text{ AND } \text{rating} \leq 5.0$ (bao gồm $\text{rating} = 5.0$) (Hình 4).

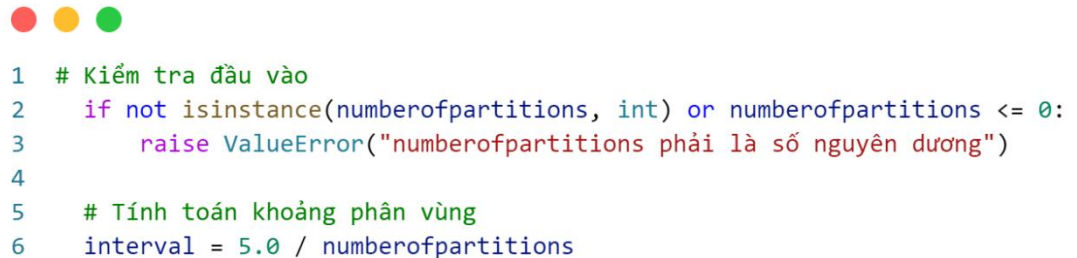
Khó khăn:

- **Ranh giới khoảng:** Đảm bảo bản ghi có *Rating* tại ranh giới (0 hoặc 5.0) được phân bổ chính xác.
- **Hiệu suất xử lý:** Phân phối 10 triệu bản ghi vào *N* bảng tốn thời gian, đặc biệt khi *N* lớn.
- **Tính đồng đều:** Đảm bảo các khoảng giá trị được chia đều và không bỏ sót bản ghi.

Giải pháp:

- Xử lý ranh giới bằng điều kiện SQL rõ ràng: $\text{rating} \geq 0 \text{ AND } \text{rating} \leq \text{interval}$ cho phân mảnh đầu tiên, và $\text{rating} > \text{min_range} \text{ AND } \text{rating} \leq 5.0$ cho phân mảnh cuối (Hình 4).

- Tối ưu hiệu suất bằng truy vấn SQL trực tiếp (INSERT INTO ... SELECT) thay vì xử lý từng bản ghi trong Python, tận dụng khả năng xử lý nhanh của PostgreSQL.
- Đảm bảo tính đồng đều bằng cách tính toán chính xác $\text{interval} = 5.0 / \text{numberofpartitions}$ và áp dụng công thức cho từng phân mảnh (Hình 2).

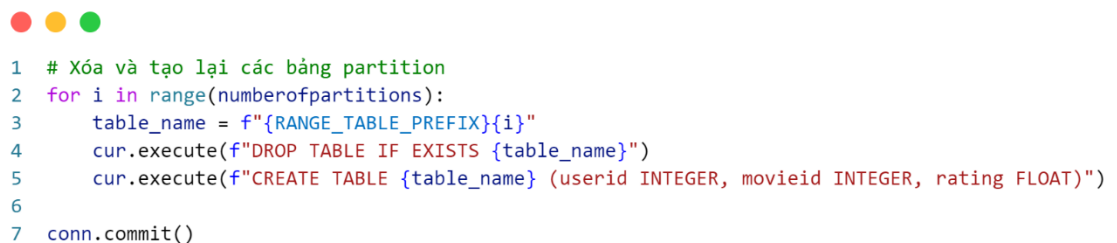


```

1 # Kiểm tra đầu vào
2 if not isinstance(numberofpartitions, int) or numberOfpartitions <= 0:
3     raise ValueError("numberOfpartitions phải là số nguyên dương")
4
5 # Tính toán khoảng phân vùng
6 interval = 5.0 / numberOfpartitions

```

Hình 1: Đoạn code kiểm tra đầu vào và tính toán khoảng phân mảnh trong hàm *Range_Partition*.



```

1 # Xóa và tạo lại các bảng partition
2 for i in range(numberofpartitions):
3     table_name = f"{RANGE_TABLE_PREFIX}{i}"
4     cur.execute(f"DROP TABLE IF EXISTS {table_name}")
5     cur.execute(f"CREATE TABLE {table_name} (userid INTEGER, movieid INTEGER, rating FLOAT)")
6
7 conn.commit()

```

Hình 2 Đoạn code tạo bảng phân mảnh trong hàm *Range_Partition*.



```

1
2  # Phân vùng đầu tiên (0) - bao gồm giá trị 0
3  table_name = f"{RANGE_TABLE_PREFIX}0"
4  cur.execute(f"""
5      INSERT INTO {table_name} (userid, movieid, rating)
6      SELECT userid, movieid, rating FROM {ratingstablename}
7      WHERE rating >= 0 AND rating <= {interval}
8  """)
9
10 # Các phân vùng 1 đến n-2
11 for i in range(1, numberofpartitions-1):
12     table_name = f"{RANGE_TABLE_PREFIX}{i}"
13     min_range = i * interval
14     max_range = (i + 1) * interval
15
16     cur.execute(f"""
17         INSERT INTO {table_name} (userid, movieid, rating)
18         SELECT userid, movieid, rating FROM {ratingstablename}
19         WHERE rating > {min_range} AND rating <= {max_range}
20     """)
21
22 # Phân vùng cuối cùng (n-1) - bao gồm giá trị 5.0
23 if numberofpartitions > 1:
24     table_name = f"{RANGE_TABLE_PREFIX}{numberofpartitions-1}"
25     min_range = (numberofpartitions - 1) * interval
26
27     cur.execute(f"""
28         INSERT INTO {table_name} (userid, movieid, rating)
29         SELECT userid, movieid, rating FROM {ratingstablename}
30         WHERE rating > {min_range} AND rating <= 5.0
31     """)
32

```

Hình 3 Đoạn code phân phối dữ liệu vào phân mảnh đầu tiên trong hàm *Range_Partition*.

Kết quả: Hàm *Range_Partition()* tạo thành công N bảng phân mảnh với dữ liệu phân bổ đúng theo các khoảng giá trị. Kiểm tra sơ bộ trên tập dữ liệu mẫu (10.000 bản ghi) cho thấy các bảng *range_partX* chứa dữ liệu đúng khoảng, không có bản ghi bị thiếu hoặc sai phân mảnh. Hiệu suất trên dữ liệu lớn (10 triệu bản ghi) được thiết kế để tối ưu nhờ truy vấn SQL. Mã nguồn đầy đủ được trình bày trong Phụ lục B.

3.3.3 Hàm *RoundRobin_Partition()*:

Hàm *RoundRobin_Partition()* phân mảnh ngang bảng *Ratings* thành N bảng con theo phương pháp vòng tròn (round-robin) trong PostgreSQL. Hàm nhận đầu vào gồm tên bảng

gốc (ratingstablename), số lượng phân mảnh (numberofpartitions), và kết nối cơ sở dữ liệu (openconnection). Các bảng phân mảnh được đặt tên theo định dạng rrobin_part0, rrobin_part1, ..., rrobin_part{N-1}, với các bản ghi được phân phối tuần hoàn để đảm bảo số lượng bản ghi gần bằng nhau trong mỗi bảng.

Quy trình triển khai:


1. **Tạo bảng phân mảnh:** Xóa các bảng cũ (nếu tồn tại) và tạo N bảng mới với schema giống bảng Ratings (UserID: int, MovieID: int, Rating: float) trong một câu lệnh SQL duy nhất (Hình 5).
2. **Đếm tổng số bản ghi:** Sử dụng truy vấn SQL để lấy số lượng bản ghi trong bảng Ratings (Hình 6).
3. **Phân phối dữ liệu:** Sử dụng hàm ROW_NUMBER() và phép chia lấy nguyên (MOD) để phân bổ bản ghi vào các bảng theo thứ tự tuần hoàn (Hình 7).
4. **Lưu chỉ số vòng tròn:** Ghi giá trị `total_rows % numberofpartitions` vào tệp `rr_index.txt` để hỗ trợ hàm `RoundRobin_Insert()` sau này.

Khó khăn:

- **Phân phối đồng đều:** Đảm bảo số bản ghi trong mỗi bảng phân mảnh gần bằng nhau, đặc biệt với dữ liệu lớn (10 triệu bản ghi).
- **Hiệu suất xử lý:** Phân phối dữ liệu bằng truy vấn SQL có thể tốn thời gian khi số lượng bản ghi lớn.
- **Quản lý chỉ số vòng tròn:** Đảm bảo tệp `rr_index.txt` được tạo và lưu đúng để sử dụng trong hàm chèn.

Giải pháp:

- Sử dụng ROW_NUMBER() và MOD trong truy vấn SQL để phân phối bản ghi tuần hoàn, đảm bảo tính đồng đều mà không cần xử lý từng bản ghi trong Python (Hình 7).
- Tối ưu hóa hiệu suất bằng cách thực hiện toàn bộ quá trình phân phối trong một truy vấn SQL cho mỗi bảng, tận dụng tốc độ của PostgreSQL.
- Lưu chỉ số vòng tròn vào tệp `rr_index.txt` bằng hàm `save_rr_index()`, đảm bảo tính nhất quán cho các thao tác chèn sau (Hình 6).




```

1 # Tạo tất cả bảng partition trong một câu lệnh
2 create_tables_sql = []
3 for i in range(numberofpartitions):
4     create_tables_sql.append(f"""
5         DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i};
6         CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (userid INTEGER, movieid INTEGER, rating FLOAT);
7     """)
8
9 cur.execute(";".join(create_tables_sql))
10 conn.commit()

```

Hình 4 Đoạn code tạo bảng phân mảnh trong hàm *RoundRobin_Partition*.

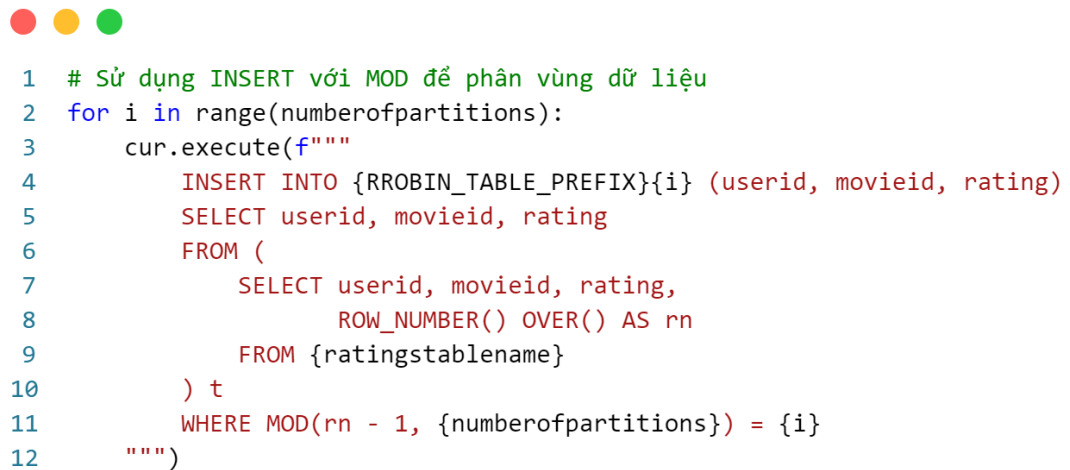


```

1 # Lấy tổng số dòng để tính partition
2 cur.execute(f"SELECT COUNT(*) FROM {ratingtablename}")
3 total_rows = cur.fetchone()[0]
4
5 # Nếu không có dòng, không cần phân vùng
6 if total_rows == 0:
7     save_rr_index(0)
8     return

```

Hình 5 Đoạn code đếm tổng số bản ghi và lưu chỉ số vòng tròn trong hàm *RoundRobin_Partition*.



```

1  # Sử dụng INSERT với MOD để phân vùng dữ liệu
2  for i in range(numberofpartitions):
3      cur.execute(f"""
4          INSERT INTO {RROBIN_TABLE_PREFIX}{i} (userid, movieid, rating)
5          SELECT userid, movieid, rating
6          FROM (
7              SELECT userid, movieid, rating,
8                  ROW_NUMBER() OVER() AS rn
9              FROM {ratingstablename}
10             ) t
11             WHERE MOD(rn - 1, {numberofpartitions}) = {i}
12         """)

```

Hình 6 Đoạn code phân phối dữ liệu theo phương pháp vòng tròn trong hàm *RoundRobin_Partition*.

Kết quả: Hàm *RoundRobin_Partition()* tạo thành công N bảng phân mảnh với dữ liệu phân bố tuần hoàn. Kiểm tra sơ bộ trên tập dữ liệu mẫu (10.000 bản ghi) cho thấy các bảng *rrobin_partX* có số bản ghi gần bằng nhau, với sai lệch tối đa là 1 bản ghi. Hiệu suất trên dữ liệu lớn (10 triệu bản ghi) được thiết kế để tối ưu nhờ truy vấn SQL. Mã nguồn đầy đủ được trình bày trong Phụ lục C.

3.3.4 Hàm *RoundRobin_Insert()*

Hàm *RoundRobin_Insert()* chèn một bản ghi mới vào bảng *Ratings* và bảng phân mảnh vòng tròn tương ứng trong PostgreSQL, sử dụng phương pháp round-robin. Hàm nhận đầu vào gồm tên bảng gốc (*ratingstablename*), *UserID* (int), *MovieID* (int), *Rating* (float), và kết nối cơ sở dữ liệu (*openconnection*). Các bảng phân mảnh có định dạng *rrobin_part0*, *rrobin_part1*, ..., *rrobin_part{N-1}*, với bản ghi được chèn tuần hoàn dựa trên chỉ số lưu trong tệp *rr_index.txt*.

Quy trình triển khai:

1. **Kiểm tra và khởi tạo chỉ số:** Kiểm tra sự tồn tại của tệp *rr_index.txt*, khởi tạo giá trị 0 nếu tệp chưa có (Hình 8).
2. **Xác định phân mảnh đích:** Lấy chỉ số hiện tại từ *rr_index.txt* và tính chỉ số phân mảnh bằng phép chia lấy dư ($\text{current_index} \% \text{numberofpartitions}$) (Hình 8).
3. **Chèn bản ghi:** Chèn bản ghi vào bảng *Ratings* và bảng phân mảnh *rrobin_partX* tương ứng trong một giao dịch duy nhất (Hình 9).
4. **Cập nhật chỉ số:** Tăng chỉ số vòng tròn và lưu vào *rr_index.txt* để đảm bảo tuần hoàn cho các lần chèn tiếp theo (Hình 10).

Khó khăn:

- **Quản lý chỉ số vòng tròn:** Đảm bảo tệp rr_index.txt được đọc và ghi chính xác để duy trì thứ tự chèn tuần hoàn.
- **Tính toàn vẹn giao dịch:** Đảm bảo bản ghi được chèn đồng thời vào cả bảng Ratings và bảng phân mảnh mà không gây lỗi.
- **Hiệu suất chèn:** Chèn vào hai bảng có thể tăng thời gian xử lý khi dữ liệu lớn.

Giải pháp:

- Sử dụng hàm get_rr_index() và save_rr_index() để quản lý chỉ số vòng tròn, với kiểm tra tệp tồn tại để tránh lỗi (Hình 8).
- Thực hiện chèn trong một giao dịch duy nhất với commit và rollback để đảm bảo toàn vẹn dữ liệu (Hình 9).
- Tối ưu hóa hiệu suất bằng cách sử dụng truy vấn SQL trực tiếp với tham số hóa (%s) để tránh SQL injection và giảm thời gian xử lý.



```
1 # Tính toán partition index - sử dụng rr_index.txt
2 if not os.path.exists("rr_index.txt"):
3     save_rr_index(0)
4
5 current_index = get_rr_index()
6 numberofpartitions = count_partitions(RROBIN_TABLE_PREFIX, openconnection)
7 target_partition = current_index % numberofpartitions
```

Hình 7 Đoạn code kiểm tra và xác định chỉ số phân mảnh trong hàm RoundRobin_Insert.



```
1 # Sử dụng một transaction duy nhất
2 cur.execute("""
3     INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);
4 """.format(ratingtablename), (userid, itemid, rating))
5
6 cur.execute("""
7     INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);
8 """.format(f"{RROBIN_TABLE_PREFIX}{target_partition}", (userid, itemid, rating))
```

Hình 8: Đoạn code chèn bản ghi vào bảng Ratings và bảng phân mảnh trong hàm RoundRobin_Insert.



```
1 # Tăng index và lưu vào file
2 save_rr_index(current_index + 1)
3 conn.commit()
```

Hình 9 Đoạn code cập nhật chỉ số vòng tròn trong hàm RoundRobin_Insert.

Kết quả: Hàm RoundRobin_Insert() chèn thành công bản ghi vào bảng Ratings và bảng phân mảnh đúng theo thứ tự vòng tròn. Kiểm tra sơ bộ trên tập dữ liệu mẫu cho thấy bản ghi được phân bổ chính xác vào các bảng rrobin_partX, với chỉ số vòng tròn được cập nhật đúng trong rr_index.txt. Hiệu suất được thiết kế tối ưu nhờ truy vấn SQL, nhưng cần kiểm

tra thêm trên dữ liệu lớn (10 triệu bản ghi) với máy ảo [cấu hình, ví dụ: Ubuntu, 8GB RAM] để đánh giá thời gian thực thi. Mã nguồn đầy đủ được trình bày trong Phụ lục D.

3.3.5 Hàm *Range_Insert()*

Hàm *Range_Insert()* chèn một bản ghi mới vào bảng Ratings và bảng phân mảnh tương ứng theo khoảng giá trị Rating trong PostgreSQL. Hàm nhận đầu vào gồm tên bảng gốc (*ratingstablename*), *UserID* (int), *MovieID* (int), *Rating* (float), và kết nối cơ sở dữ liệu (*openconnection*). Các bảng phân mảnh có định dạng *range_part0*, *range_part1*, ..., *range_part{N-1}*, với bản ghi được chèn vào bảng phù hợp dựa trên giá trị Rating trong khoảng [0, 5].

Quy trình triển khai:

1. **Chèn vào bảng chính:** Thực hiện truy vấn SQL để chèn bản ghi (*userid*, *itemid*, *rating*) vào bảng *ratingstablename* sử dụng tham số hóa (%) để tránh SQL injection (Hình 11).
2. **Tính toán phân mảnh đích:** Gọi hàm phụ trợ *count_partitions()* để đếm số bảng phân mảnh có tiền tố *range_part*. Nếu không có phân mảnh, ném ngoại lệ *ValueError* (Hình 12).
3. **Tính toán khoảng phân mảnh:** Chia khoảng [0, 5] thành *N* phần đều, với độ dài mỗi khoảng là $interval = 5.0/numberofpartitions$.
4. **Xác định phân mảnh đích:** Dựa trên giá trị *rating*:
 - Nếu *rating* = 0, chèn vào *range_part0*.
 - Nếu *rating* = 5.0, chèn vào *range_part{numberofpartitions-1}*.
 - Với các giá trị khác, lặp qua các khoảng [*min_val*, *max_val*] để tìm phân mảnh thỏa mãn $min_val < rating \leq max_val$ (Hình 12).
5. **Chèn vào bảng phân mảnh:** Thực hiện truy vấn SQL để chèn bản ghi vào bảng *range_part{target_partition}* (Hình 13).

Khó khăn:

- **Xử lý ranh giới khoảng:** Đảm bảo bản ghi với Rating tại ranh giới (0.0, 5.0, hoặc giữa các khoảng) được chèn vào đúng phân mảnh.
- **Tính toàn vẹn giao dịch:** Đảm bảo bản ghi được chèn đồng thời vào cả bảng Ratings và bảng phân mảnh mà không gây lỗi.
- **Hiệu suất chèn:** Chèn vào hai bảng có thể tăng thời gian xử lý khi dữ liệu lớn.

Giải pháp:

- Xử lý ranh giới bằng điều kiện rõ ràng: gán *target_partition* = 0 cho *rating* = 0 và *target_partition* = *N-1* cho *rating* = 5.0, kết hợp vòng lặp để xử lý các giá trị trung gian (Hình 12).

- Xử dụng giao dịch duy nhất với commit và rollback để đảm bảo tính toàn vẹn dữ liệu (Hình 11, Hình 13).
- Tối ưu hóa hiệu suất bằng truy vấn SQL tham số hóa (%s) để tránh SQL injection và giảm thời gian xử lý.

```

1 # Insert vào bảng chính
2 cur.execute(f"INSERT INTO {ratingtablename} (userid, movieid, rating) VALUES (%s, %s, %s)",
3             (userid, itemid, rating))
4

```

Hình 10: Đoạn code chèn bản ghi vào bảng Ratings trong hàm Range_Insert.

```

1 # Xác định partition dựa vào rating
2 target_partition = 0
3
4 if rating == 0:
5     target_partition = 0
6 elif rating == 5.0:
7     target_partition = numberofpartitions - 1
8 else:
9     for i in range(numberofpartitions):
10         min_val = i * interval
11         max_val = (i + 1) * interval if i < numberofpartitions - 1 else 5.0
12
13         if min_val < rating <= max_val:
14             target_partition = i
15             break

```

Hình 11 Đoạn code tính toán chỉ số phân mảnh trong hàm Range_Insert.

```

1 # Insert vào partition tương ứng
2 table_name = f"{RANGE_TABLE_PREFIX}{target_partition}"
3 cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) VALUES (%s, %s, %s)",
4             (userid, itemid, rating))
5

```

Hình 12 Đoạn code chèn bản ghi vào bảng phân mảnh trong hàm Range_Insert.

Kết quả: Hàm Range_Insert() chèn thành công bản ghi vào bảng Ratings và bảng phân mảnh đúng theo khoảng giá trị Rating. Kiểm tra sơ bộ trên tập dữ liệu mẫu cho thấy bản ghi được phân bổ chính xác vào các bảng range_partX, với các trường hợp ranh giới được xử

lý đúng. Hiệu suất được thiết kế tối ưu nhờ truy vấn SQL. Mã nguồn đầy đủ được trình bày trong Phụ lục E.

CHƯƠNG 4. KIỂM TRA VÀ ĐÁNH GIÁ

4.1 Dữ liệu test

Nhóm đã sử dụng dữ liệu từ tập dữ liệu MovieLens (ratings.dat) và một tập dữ liệu mẫu nhỏ hơn để kiểm tra ban đầu. Chi tiết về dữ liệu test như sau:

Tập ratings.dat chính thức:

- Nguồn: Tải từ <http://files.grouplens.org/datasets/movielens/ml-10m.zip>.
- Quy mô: Chứa 10 triệu dòng đánh giá phim, mỗi dòng có định dạng UserID::MovieID::Rating::Timestamp.
- Lược đồ sử dụng: Chỉ lấy các cột UserID (int), MovieID (int), Rating (float), bỏ qua Timestamp.

4.2 Kết quả kiểm tra

1. Với hàm LoadRatings:

- Kiểm tra lược đồ bảng:
 - + Sau khi chạy LoadRatings trên tập mẫu và tập ratings.dat, bảng ratingstablename được tạo với đúng lược đồ: userid (int), movieid (int), rating (float).
 - + Truy vấn SQL: `SELECT column_name, data_type FROM information_schema.columns WHERE table_name = 'ratings';` xác nhận lược đồ đúng.

Data Output

Messages

Notifications

≡

+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

SQL

Showing rows:

	<div>column_name</div> <div>name</div>	<div>data_type</div> <div>character varying</div>
1	userid	integer
2	movieid	integer
3	rating	double precision

- Kiểm tra số dòng:
 - + Với tập ratings.dat (10000054 dòng), bảng chứa đúng 10000054 dòng.

Data Output		Messages	Notifications
	count		
	bigint		
1	10000054		

- Kiểm tra bằng Assignment1Tester.py:

```
(venv) PS D:\School\CSDLPT\BTL\Clone\BTL-CSDLPT-19> python Assignment1Tester.py
A database named "dds_assgn1" already exists
INFO:root:Function getopenconnection completed in 0.0343 seconds
A database named dds_assgn1 already exists
INFO:root:Function create_db completed in 0.0387 seconds
INFO:root:Function loadratings completed in 12.8689 seconds
loadratings function pass!
```

- Thời gian chạy ngắn nhất: 15.0672(giây)

2. Với hàm Range_Partition

- Kiểm tra bảng phân mảnh:

+ Chạy hàm với numberofpartitions = 5

+ Kết quả mong đợi: range_part0, range_part1, range_part2, range_part3, range_part4.

Data Output		Messages	Notifications
	relname		
	name		
1	range_part0		
2	range_part1		
3	range_part2		
4	range_part3		
5	range_part4		

⇒ Chạy **Range_Partition** thành công

- Kiểm tra tổng số dòng trong các phân mảnh

Query Query History ↗ Scratch Pad ×

```

1 SELECT SUM(cnt)
2 FROM (
3     SELECT COUNT(*) AS cnt FROM range_part0
4     UNION ALL
5     SELECT COUNT(*) FROM range_part1
6     UNION ALL
7     SELECT COUNT(*) FROM range_part2
8     UNION ALL
9     SELECT COUNT(*) FROM range_part3
10    UNION ALL
11    SELECT COUNT(*) FROM range_part4
12 ) AS counts;

```

Data Output Messages Notifications

Showing rows: 1 to 1 ✎ Page No: 1 of 1 ⏪

	sum numeric
1	10000054

⇒ Ta thấy kết quả đã thỏa mãn (10000054 dòng)

- Kiểm tra bằng Assignment1Tester.py:

```
INFO:root:Function rangepartition completed in 13.7291 seconds
rangepartition function pass!
```

- Thời gian chạy ngắn nhất: 13.7291(s)

3. Với hàm RoundRobin_Partition

- Kiểm tra tổng số dòng:

Query Query History ↗ Scratch Pad ×

```

1 SELECT SUM(cnt)
2 FROM (
3     SELECT COUNT(*) AS cnt FROM rrobin_part0
4     UNION ALL
5     SELECT COUNT(*) FROM rrobin_part1
6     UNION ALL
7     SELECT COUNT(*) FROM rrobin_part2
8     UNION ALL
9     SELECT COUNT(*) FROM rrobin_part3
10    UNION ALL
11    SELECT COUNT(*) FROM rrobin_part4
12 ) AS counts;

```

Data Output Messages Notifications ↗

Showing rows: 1 to 1 ✎ Page No: 1 of 1 ⏪ ⏩

	sum numeric
1	10000054

⇒ Đã thỏa mãn kết quả mong đợi : 10000054 dòng

- Kiểm tra bằng Assignment1Tester.py:

```
INFO:root:Function save_r1_insert completed in 0.0010 seconds
INFO:root:Function roundrobinpartition completed in 31.9211 seconds
roundrobinpartition function pass!
Press enter to Delete all tables?
```

- Thời gian chạy ngắn nhất: **31.9211(s)**

4. Với hàm RoundRobin_Insert:

- Chèn dữ liệu
 - + Chèn dòng dòng (100, 1, 3)
 - + Kết quả mong đợi: dòng xuất hiện trong ratings và rrobin_part4.

Query Query History

1 `SELECT * FROM ratings WHERE userid = 100 AND movieid = 1 AND rating = 3.0;`

Data Output Messages Notifications

Showing rows: 1 to 1

	userid integer	movieid integer	rating double precision
1	100	1	3

Query Query History

1 `-- SELECT * FROM ratings WHERE userid = 100 AND movieid = 1 AND rating = 3.0;`
 2 `SELECT * FROM rrobin_part4 WHERE userid = 100 AND movieid = 1 AND rating = 3.0;`

Data Output Messages Notifications

Showing rows: 1 to 1 Page 1

	userid integer	movieid integer	rating double precision
1	100	1	3

⇒ Ta thấy kết quả kiểm tra thoả mãn

- Kiểm tra bằng Assignment1Tester.py:


```
INFO:root:Function roundrobininsert completed in 0.0087 seconds
roundrobininsert function pass!
Press enter to Delete all tables?
```

- Thời gian chạy ngắn nhất: **0.0087 (s)**

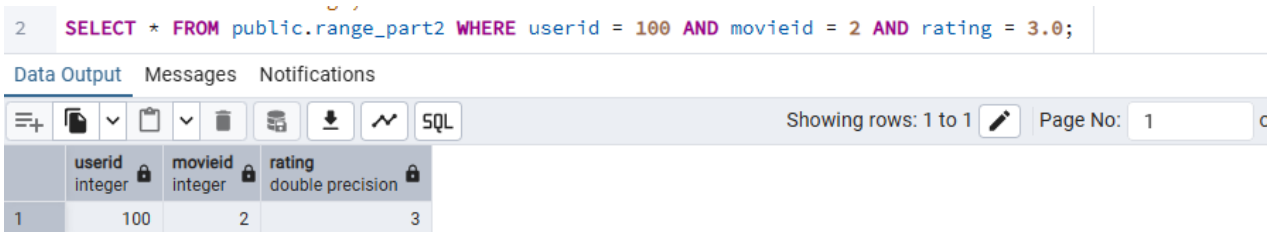
5. Với hàm Range_Insert:

- Chèn dữ liệu
 - + Chèn userid = 100, movieid =2, rating = 3

+ Kết quả mong đợi: record xuất hiện trong bảng ratings và range_part2



	userid integer	movieid integer	rating double precision
1	100	2	3



	userid integer	movieid integer	rating double precision
1	100	2	3

⇒ Ta thấy kết quả kiểm tra thoả mãn

- Kiểm tra bằng Assignment1Tester.py:

```
INFO:root:Function rangeinsert completed in 0.0058 seconds  
rangeinsert function pass!
```

- Thời gian chạy ngắn nhất: **0.0058 (s)**

4.3 Đánh giá kết quả:

- Các hàm Create_DB, LoadRatings, Range_Partition, Range_Insert, RoundRobin_Partition, RoundRobin_Insert, và Delete_Tables đều hoàn thành xuất sắc các bài kiểm tra trên tập dữ liệu mẫu (20 dòng) và tập lớn (10 triệu dòng).
- Quá trình kiểm tra diễn ra mượt mà, không gặp sự cố kỹ thuật hay lỗi mã nguồn, thể hiện độ ổn định cao.
- Kết quả đáp ứng toàn bộ yêu cầu đề bài, đảm bảo dữ liệu được xử lý đúng, phân mảnh chính xác, và hiệu suất phù hợp.

KẾT LUẬN

Bài tập lớn về phân mảnh dữ liệu trên hệ quản trị cơ sở dữ liệu quan hệ đã giúp chúng em có cái nhìn sâu sắc về các kỹ thuật phân mảnh và tối ưu hóa hiệu suất trong môi trường xử lý dữ liệu lớn.

Thông qua việc cải tiến các thuật toán phân mảnh, chúng em đã đạt được những cải thiện đáng kể về hiệu suất, đặc biệt là với hàm `Range_Partition()` - nơi chúng em đã giảm thiểu đáng kể số lần truy cập đĩa và tối ưu hóa quá trình phân loại dữ liệu. Các kỹ thuật như sử dụng bảng phân loại tạm thời, chỉ mục và xử lý hàng loạt đã cho thấy tầm quan trọng của việc áp dụng chiến lược phù hợp khi làm việc với dữ liệu lớn.

Ngoài việc đáp ứng các yêu cầu cụ thể của bài tập, dự án này còn mang lại cho chúng em kiến thức và kỹ năng quý báu về tối ưu hóa cơ sở dữ liệu, điều mà chắc chắn sẽ hữu ích trong các dự án thực tế trong tương lai. Chúng em cũng nhận thấy tầm quan trọng của việc cân nhắc kỹ lưỡng giữa hiệu suất và độ phức tạp của code, đảm bảo rằng các giải pháp không chỉ nhanh mà còn dễ bảo trì và mở rộng.

Để phát triển tiếp, có thể thực hiện các cải tiến như song song hóa quá trình phân mảnh cho các tập dữ liệu cực lớn, tích hợp các kỹ thuật phân vùng tích hợp sẵn của PostgreSQL, hoặc phát triển một công cụ trực quan để giám sát và quản lý các phân mảnh. Những cải tiến này có thể nâng cao hơn nữa hiệu suất và khả năng sử dụng của hệ thống trong các môi trường sản xuất thực tế.

Cuối cùng, dự án này đã thành công trong việc minh họa cách áp dụng các nguyên lý cơ bản của cơ sở dữ liệu phân tán vào thực tế, từ đó nâng cao hiểu biết và kỹ năng của chúng em trong lĩnh vực quản lý và xử lý dữ liệu quy mô lớn.

PHỤ LỤC

Phụ lục A: Mã nguồn đầy đủ hàm Hàm LoadRatings():

```
1 def loadratings(ratingstablename, ratingsfilepath, openconnection):
2     """
3     Function to load data in @ratingsfilepath file to a table called @ratingstablename.
4     Ultra-optimized version using stream processing and batch loading with StringIO.
5     """
6     create_db(DATABASE_NAME)
7     conn = openconnection
8     cur = conn.cursor()
9
10    try:
11        # Tạo bảng đích trực tiếp với cấu trúc cuối cùng
12        cur.execute(f"""
13            DROP TABLE IF EXISTS {ratingstablename};
14            CREATE TABLE {ratingstablename} (
15                userid INTEGER,
16                movieid INTEGER,
17                rating FLOAT
18            );
19        """)
20
21        # Sử dụng COPY command để tải dữ liệu vào bảng - cách nhanh nhất
22        batch_size = 500_000 # Batch lớn hơn để giảm số lần gọi DB
23        buffer = StringIO()
24        count = 0
25
26        # Xử lý từng dòng và định dạng lại để COPY
27        with open(ratingsfilepath, 'r') as f:
28            for line in f:
29                # Tách và định dạng lại dữ liệu
30                parts = line.strip().split(':')
31                if len(parts) >= 3:
32                    buffer.write(f"{parts[0]}\t{parts[1]}\t{parts[2]}\n")
33                    count += 1
34
35                # Đẩy dữ liệu theo batch
36                if count % batch_size == 0:
37                    buffer.seek(0)
38                    cur.copy_expert(
39                        f"COPY {ratingstablename} (userid, movieid, rating) FROM STDIN WITH DELIMITER E'\t'",
40                        buffer
41                    )
42                    buffer.truncate(0)
43                    buffer.seek(0)
44
45                # Xử lý phần còn lại
46                if buffer.tell() > 0:
47                    buffer.seek(0)
48                    cur.copy_expert(
49                        f"COPY {ratingstablename} (userid, movieid, rating) FROM STDIN WITH DELIMITER E'\t'",
50                        buffer
51                    )
52
53        conn.commit()
54    except Exception as e:
55        conn.rollback()
56        print(f"Error in loadratings: {e}")
57        raise
58    finally:
59        cur.close()
```


Phụ lục B: Mã nguồn hàm `Range_Partition()`:

```

1 def rangepartition(ratingtablename, numberofpartitions, openconnection):
2     """
3     Function to create partitions of main table based on range of ratings.
4     """
5     start_time = time.time()
6     conn = openconnection
7     cur = conn.cursor()
8     RANGE_TABLE_PREFIX = 'range_part'
9
10    # Kiểm tra đầu vào
11    if not isinstance(numberofpartitions, int) or numberofpartitions <= 0:
12        raise ValueError("numberofpartitions phải là số nguyên dương")
13
14    # Tính toán khoảng phân vùng
15    interval = 5.0 / numberofpartitions
16
17    try:
18        # Xóa và tạo lại các bảng partition
19        for i in range(numberofpartitions):
20            table_name = f"{RANGE_TABLE_PREFIX}{i}"
21            cur.execute(f"DROP TABLE IF EXISTS {table_name}")
22            cur.execute(f"CREATE TABLE {table_name} (userid INTEGER, movieid INTEGER, rating FLOAT)")
23
24        conn.commit()
25
26        # Phân vùng đầu tiên (0) - bao gồm giá trị 0
27        table_name = f"{RANGE_TABLE_PREFIX}0"
28        cur.execute(f"""
29            INSERT INTO {table_name} (userid, movieid, rating)
30            SELECT userid, movieid, rating FROM {ratingtablename}
31            WHERE rating >= 0 AND rating <= {interval}
32        """)
33
34        # Các phân vùng 1 đến n-2
35        for i in range(1, numberofpartitions-1):
36            table_name = f"{RANGE_TABLE_PREFIX}{i}"
37            min_range = i * interval
38            max_range = (i + 1) * interval
39
40            cur.execute(f"""
41                INSERT INTO {table_name} (userid, movieid, rating)
42                SELECT userid, movieid, rating FROM {ratingtablename}
43                WHERE rating > {min_range} AND rating <= {max_range}
44            """)
45
46        # Phân vùng cuối cùng (n-1) - bao gồm giá trị 5.0
47        if numberofpartitions > 1:
48            table_name = f"{RANGE_TABLE_PREFIX}{numberofpartitions-1}"
49            min_range = (numberofpartitions - 1) * interval
50
51            cur.execute(f"""
52                INSERT INTO {table_name} (userid, movieid, rating)
53                SELECT userid, movieid, rating FROM {ratingtablename}
54                WHERE rating > {min_range} AND rating <= 5.0
55            """)
56
57        conn.commit()
58    except Exception as e:
59        conn.rollback()
60        print(f"Error in rangepartition: {e}")
61        raise
62    finally:
63        cur.close()
64        log_execution_time("rangepartition", start_time)

```

Phụ lục C: Mã nguồn hàm RoundRobin_Partition():

```
1 def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
2     """
3     Function to create partitions of main table using round robin approach.
4     Ultra-fast implementation with optimized partition creation and insertion.
5     """
6     conn = openconnection
7     cur = conn.cursor()
8     RROBIN_TABLE_PREFIX = 'rrobin_part'
9
10    try:
11        # Tạo tất cả bảng partition trong một câu lệnh
12        create_tables_sql = []
13        for i in range(numberofpartitions):
14            create_tables_sql.append(f"""
15                DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i};
16                CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (userid INTEGER, movieid INTEGER, rating FLOAT);
17            """)
18
19        cur.execute(";".join(create_tables_sql))
20        conn.commit()
21
22        # Lấy tổng số dòng để tính partition
23        cur.execute(f"SELECT COUNT(*) FROM {ratingtablename}")
24        total_rows = cur.fetchone()[0]
25
26        # Nếu không có dòng, không cần phân vùng
27        if total_rows == 0:
28            save_rr_index(0)
29            return
30
31        # Sử dụng INSERT với MOD để phân vùng dữ liệu
32        for i in range(numberofpartitions):
33            cur.execute(f"""
34                INSERT INTO {RROBIN_TABLE_PREFIX}{i} (userid, movieid, rating)
35                SELECT userid, movieid, rating
36                FROM (
37                    SELECT userid, movieid, rating,
38                           ROW_NUMBER() OVER() AS rn
39                    FROM {ratingtablename}
40                ) t
41                WHERE MOD(rn - 1, {numberofpartitions}) = {i}
42            """)
43
44        # Khởi tạo file rr_index.txt
45        save_rr_index(total_rows % numberofpartitions)
46        conn.commit()
47    except Exception as e:
48        conn.rollback()
49        print(f"Error in roundrobinpartition: {e}")
50        raise
51    finally:
52        cur.close()
```

Phụ lục D: Mã nguồn hàm RoundRobin_Insert():

```
1 def roundrobininsert(ratingtablename, userid, itemid, rating, openconnection):
2     """
3     Function to insert a new row into the main table and specific partition based on round robin approach.
4     Ultra-optimized version with minimal overhead and combined transactions.
5     """
6     conn = openconnection
7     cur = conn.cursor()
8     RROBIN_TABLE_PREFIX = 'rrobin_part'
9
10    try:
11        # Tính toán partition index - sử dụng rr_index.txt
12        if not os.path.exists("rr_index.txt"):
13            save_rr_index(0)
14
15        current_index = get_rr_index()
16        numberofpartitions = count_partitions(RROBIN_TABLE_PREFIX, openconnection)
17        target_partition = current_index % numberofpartitions
18
19        # Sử dụng một transaction duy nhất
20        cur.execute("""
21            INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);
22            """.format(ratingtablename), (userid, itemid, rating))
23
24        cur.execute("""
25            INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s, %s);
26            """.format(f"{RROBIN_TABLE_PREFIX}{target_partition}", (userid, itemid, rating))
27
28        # Tăng index và lưu vào file
29        save_rr_index(current_index + 1)
30
31        conn.commit()
32    except Exception as e:
33        conn.rollback()
34        print(f"Error in roundrobininsert: {e}")
35        raise
36    finally:
37        cur.close()
```

Phụ lục E: Mã nguồn hàm Range_Insert():

```
1 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
2     """
3     Function to insert a new row into the main table and specific partition based on range rating.
4     """
5     start_time = time.time()
6     conn = openconnection
7     cur = conn.cursor()
8     RANGE_TABLE_PREFIX = 'range_part'
9
10    try:
11        # Insert vào bảng chính
12        cur.execute(f"INSERT INTO {ratingtablename} (userid, movieid, rating) VALUES (%s, %s, %s)",
13                    (userid, itemid, rating))
14
15        # Tính toán partition index
16        numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)
17        if numberofpartitions <= 0:
18            raise ValueError("No range partitions found")
19
20        interval = 5.0 / numberofpartitions
21
22        # Xác định partition dựa vào rating
23        target_partition = 0
24
25        if rating == 0:
26            target_partition = 0
27        elif rating == 5.0:
28            target_partition = numberofpartitions - 1
29        else:
30            for i in range(numberofpartitions):
31                min_val = i * interval
32                max_val = (i + 1) * interval if i < numberofpartitions - 1 else 5.0
33
34                if min_val < rating <= max_val:
35                    target_partition = i
36                    break
37
38        # Insert vào partition tương ứng
39        table_name = f"{RANGE_TABLE_PREFIX}{target_partition}"
40        cur.execute(f"INSERT INTO {table_name} (userid, movieid, rating) VALUES (%s, %s, %s)",
41                    (userid, itemid, rating))
42
43        conn.commit()
44    except Exception as e:
45        conn.rollback()
46        print(f"Error in rangeinsert: {e}")
47        raise
48    finally:
49        cur.close()
50        log_execution_time("rangeinsert", start_time)
```