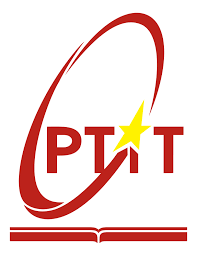
**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

**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](#_Toc200396424)

[MỤC LỤC 3](#_Toc200396425)

[DANH MỤC CÁC HÌNH VẼ 5](#_Toc200396426)

[CHƯƠNG 1. TỔNG QUAN BÀI TOÁN 6](#_Toc200396427)

[1.1 Bối cảnh và mục tiêu 6](#_Toc200396428)

[1.1.1 Bối cảnh 6](#_Toc200396429)

[1.1.2 Mục tiêu của bài toán 6](#_Toc200396430)

[1.2 Dữ liệu đầu vào 6](#_Toc200396431)

[1.2.1 Cấu trúc dữ liệu 6](#_Toc200396432)

[1.2.2 Mô tả các trường dữ liệu 7](#_Toc200396433)

[1.2.3 Đặc điểm dữ liệu 7](#_Toc200396434)

[1.2.4 Yêu cầu xử lý 7](#_Toc200396435)

[CHƯƠNG 2. Cơ sở lý thuyết 7](#_Toc200396436)

[2.1 Phân mảnh dữ liệu trong cơ sở dữ liệu 7](#_Toc200396437)

[2.1.1 Định nghĩa 8](#_Toc200396438)

[2.1.2 Các loại phân mảnh 8](#_Toc200396439)

[2.1.3 Lợi ích 8](#_Toc200396440)

[2.2 Phân mảnh ngang 9](#_Toc200396441)

[2.2.1 Phân mảnh theo khoảng (Range Partitioning) : 9](#_Toc200396442)

[2.2.2 Phân mảnh vòng tròn (Round-Robin Partitioning): 9](#_Toc200396443)

[2.3 Vai trò của hệ quản trị cơ sở dữ liệu quan hệ 10](#_Toc200396444)

[CHƯƠNG 3. PHƯƠNG PHÁP GIẢI QUYẾT 10](#_Toc200396445)

[3.1 Tổng quan quy trình 10](#_Toc200396446)

[3.2 Cài đặt môi trường 11](#_Toc200396447)

[3.2.1 Cấu hình phần mềm 11](#_Toc200396448)

[3.2.2 Tải dữ liệu MovieLens 13](#_Toc200396449)

[3.3 Triển khai các hàm Python 13](#_Toc200396450)

[3.3.1 Hàm LoadRatings() 13](#_Toc200396451)

[3.3.2 Hàm Range\_Partition() 14](#_Toc200396452)

[3.3.3 Hàm RoundRobin\_Partition(): 16](#_Toc200396453)

[3.3.4 Hàm RoundRobin\_Insert() 18](#_Toc200396454)

[3.3.5 Hàm Range\_Insert() 21](#_Toc200396455)

[3.4 Kiểm tra và đánh giá 23](#_Toc200396456)

[3.4.1 Phương pháp kiểm tra 23](#_Toc200396457)

[3.4.2 Kết quả kiểm tra 23](#_Toc200396458)

[3.4.3 Đánh giá hiệu suất: 24](#_Toc200396459)

[CHƯƠNG 4. Khó khăn và giải pháp 26](#_Toc200396460)

[4.1 Khó khăn kỹ thuật 26](#_Toc200396461)

[4.2 Giải pháp 27](#_Toc200396462)

[4.3 Bài học kinh nghiệm 28](#_Toc200396463)

[KẾT LUẬN 28](#_Toc200396464)

[Phụ lục 30](#_Toc200396465)

[Phụ lục A: 30](#_Toc200396466)

[Phụ lục B: 31](#_Toc200396467)

[Phụ lục C 32](#_Toc200396468)

[Phụ lục D: 33](#_Toc200396469)

[Phụ lục E: 34](#_Toc200396470)

[Phụ lục F: 34](#_Toc200396471)

DANH MỤC CÁC HÌNH VẼ

[Hình 1: Đoạn code xử lý và chèn dữ liệu theo lô trong hàm LoadRatings. 15](#_Toc200396020)

[Hình 2: Đoạn code kiểm tra đầu vào và tính toán khoảng phân mảnh trong hàm Range\_Partition. 16](#_Toc200396021)

[Hình 3 Đoạn code tạo bảng phân mảnh trong hàm Range\_Partition. 17](#_Toc200396022)

[Hình 4 Đoạn code phân phối dữ liệu vào phân mảnh đầu tiên trong hàm Range\_Partition. 17](#_Toc200396023)

[Hình 5 Đoạn code tạo bảng phân mảnh trong hàm RoundRobin\_Partition. 18](#_Toc200396024)

[Hình 6 Đoạn code đếm tổng số bản ghi và lưu chỉ số vòng tròn trong hàm RoundRobin\_Partition. 19](#_Toc200396025)

[Hình 7 Đoạn code phân phối dữ liệu theo phương pháp vòng tròn trong hàm RoundRobin\_Partition. 19](#_Toc200396026)

[Hình 8 Đoạn code kiểm tra và xác định chỉ số phân mảnh trong hàm RoundRobin\_Insert. 21](#_Toc200396027)

[Hình 9: Đoạn code chèn bản ghi vào bảng Ratings và bảng phân mảnh trong hàm RoundRobin\_Insert. 21](#_Toc200396028)

[Hình 10 Đoạn code cập nhật chỉ số vòng tròn trong hàm RoundRobin\_Insert. 21](#_Toc200396029)

[Hình 11:Đoạn code chèn bản ghi vào bảng Ratings trong hàm Range\_Insert. 23](#_Toc200396030)

[Hình 12 Đoạn code tính toán chỉ số phân mảnh trong hàm Range\_Insert. 23](#_Toc200396031)

[Hình 13 Đoạn code chèn bản ghi vào bảng phân mảnh trong hàm Range\_Insert. 23](#_Toc200396032)

[Hình 14 Kết quả thu được test với assignment\_tester.py 24](#_Toc200396033)

[Hình 15: So sánh thời gian thực thi theo kích thước dữ liệu 26](#_Toc200396034)

[Hình 16: Ảnh hưởng của số phân vùng (N) đến thời gian phân vùng 26](#_Toc200396035)

1. TỔNG QUAN BÀI TOÁN
   1. Bối cảnh và mục tiêu
      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. 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. Dữ liệu đầu vào
     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. 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 ≤ Rating ≤ 5.0 |
| Timestamp | INTEGER | Thời gian đánh giá (Unix timestamp) |  |

* + 1. 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. 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.

1. Cơ sở lý thuyết
   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.

* + 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.

* + 1. 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 ≥ 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.
  + 1. 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.

* 1. 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:

* + 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 liền 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.

* + 1. 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 mod 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 (tượng 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.

* 1. 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

1. PHƯƠNG PHÁP GIẢI QUYẾT
   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.

* 1. Cài đặt môi trường
     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 10



**Python**: Phiên bản: 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)

A screen shot of a computer

AI-generated content may be incorrect.

**PostgreSQL**: Phiên bản: PostgreSQL 15.x



**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)

**Cấu trúc thư mục dự án**:

bai\_tap\_lon\_CSDL\_phan\_tan/

├── Interface.py # Gồm các hàm đã cải tiến

├── Assignment1Tester.py # File kiểm thử

├── testHelper.py # Công cụ hỗ trợ kiểm thử

├── test\_data.dat # Dữ liệu kiểm thử

└── README.md # hướng dẫn setup môi trường nếu cần

* + 1. 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.

* 1. Triển khai các hàm Python
     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.

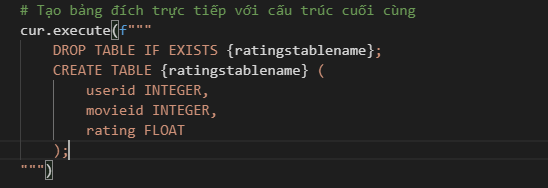


1. Mở kết nối và tạo con trỏ

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



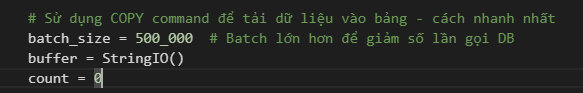
1. Tạo lại bảng đích.



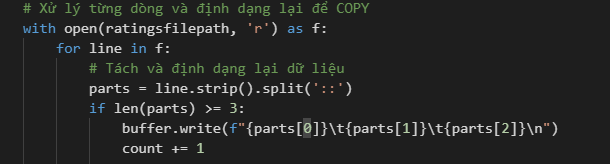
* Xóa bảng nếu đã tồn tại để đảm bảo không có dữ liệu cũ.
* Tạo bảng mới với 3 cột:
  + userid: mã người dùng (số nguyên)
  + movieid: mã phim (số nguyên)
  + rating: điểm đánh giá (float)

1. 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.



1. Đọc file và xử lý từng dòng:



* 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).

**Khó khăn**:

* Tệp 10 triệu bản ghi gây nguy cơ tràn bộ nhớ nếu đọc toàn bộ.
* Chèn từng bản ghi làm tăng thời gian thực thi.

**Giải pháp**:

* Sử dụng đọc luồng và StringIO để xử lý từng dòng, giảm sử dụng bộ nhớ.
* Áp dụng lệnh COPY với chèn theo lô để tối ưu hiệu suất (Hình 1). Thư viện psycopg2 đảm bảo kết nối ổn định.



Hình 1: Đoạn code xử lý và chèn dữ liệu theo lô trong hàm LoadRatings.

**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

* + 1. 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**:

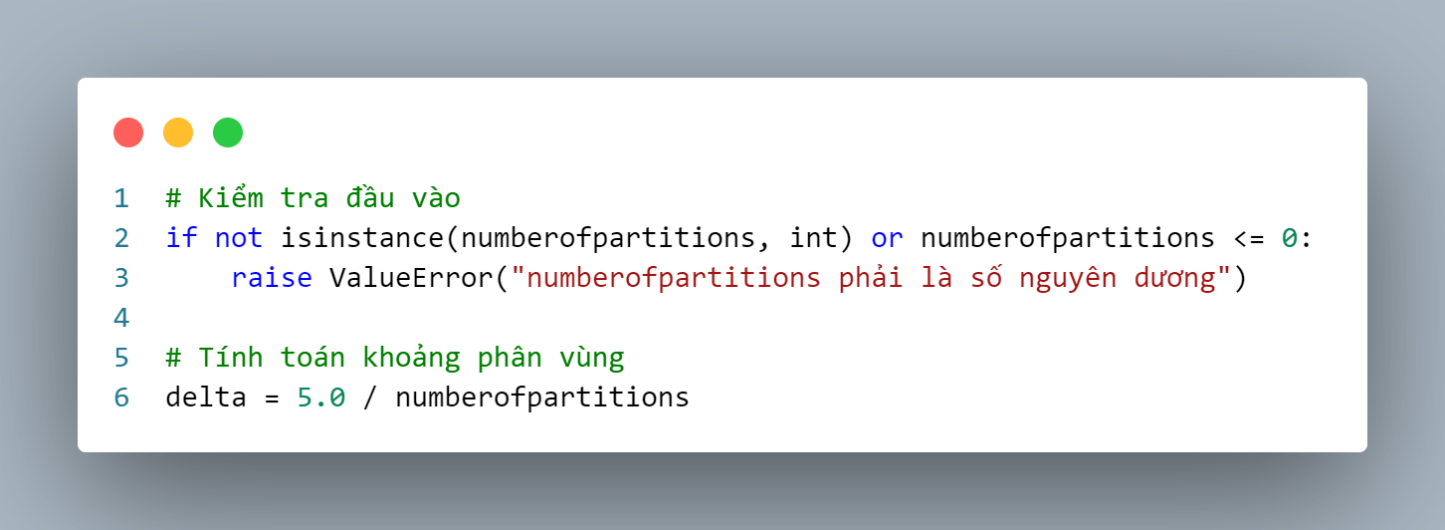
1. **Kiểm tra đầu vào**: Đảm bảo numberofpartitions là số nguyên dương, ném ngoại lệ nếu không hợp lệ (Hình 2).
2. **Tính toán khoảng phân mảnh**: Chia khoảng [0, 5] thành N phần đều, với độ dài mỗi khoảng là 5.0/N (biến delta).
3. **Tạo bảng phân mảnh**: Xóa các bảng cũ (nếu tồn tại) và tạo N bảng mới với schema giống bảng Ratings (UserID: int, MovieID: int, Rating: float) (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 dựa trên khoảng giá trị Rating. Phân mảnh đầu tiên bao gồm giá trị 0, phân mảnh cuối cùng bao gồm giá trị 5.0 (Hình 3.4).

**Khó khăn**:

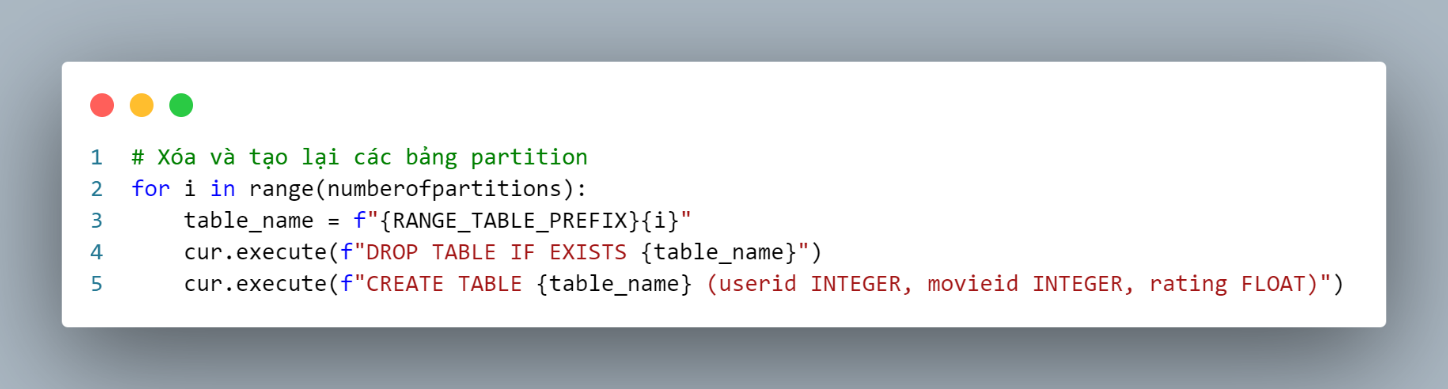
* **Ranh giới khoảng**: Đảm bảo bản ghi có Rating tại ranh giới (0 hoặc 5.0) được phân bổ chính xác.
* **Hiệu suất xử lý**: Phân phối 10 triệu bản ghi vào N bảng tốn thời gian, đặc biệt khi N lớn.
* **Tính đồng đều**: Đảm bảo các khoảng giá trị được chia đều và không bỏ sót bản ghi.

**Giải pháp**:

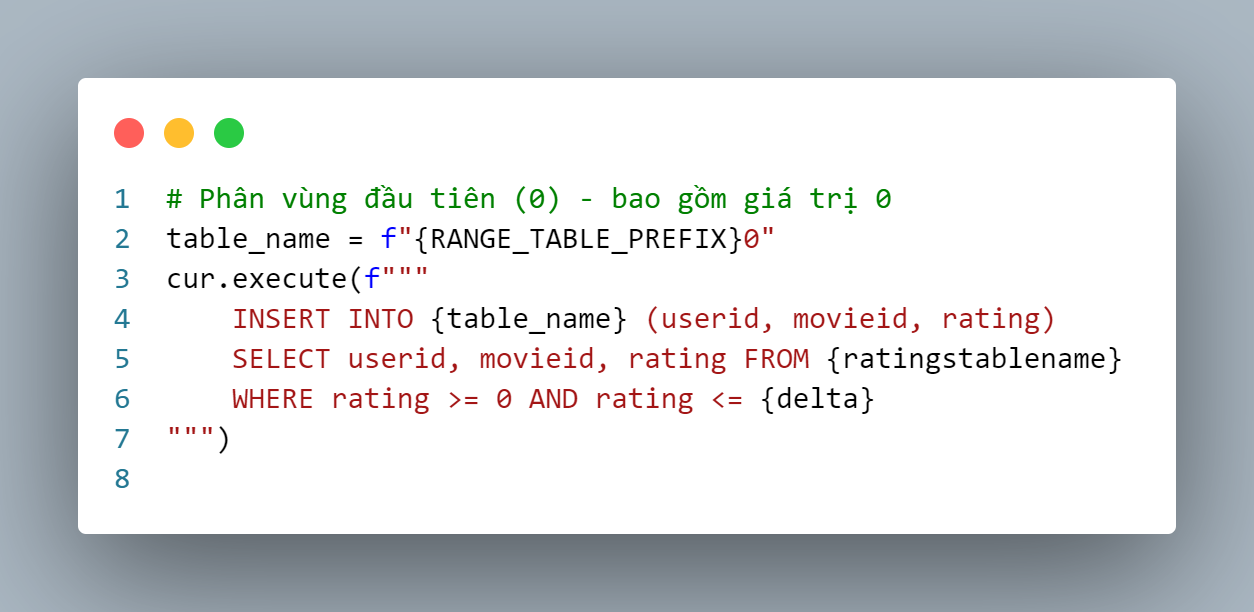
* Xử lý ranh giới bằng điều kiện SQL rõ ràng (ví dụ: rating >= 0 AND rating <= delta cho phân mảnh đầu tiên, như Hình 4).
* Tối ưu hóa hiệu suất bằng truy vấn SQL trực tiếp (INSERT INTO ... SELECT) thay vì xử lý từng bản ghi trong Python, tận dụng tốc độ của PostgreSQL.
* Đảm bảo tính đồng đều bằng cách tính toán chính xác delta = 5.0/N và áp dụng công thức cho từng phân mảnh (Hình 2).



Hình 2: Đoạn code kiểm tra đầu vào và tính toán khoảng phân mảnh trong hàm Range\_Partition.



Hình 3 Đoạn code tạo bảng phân mảnh trong hàm Range\_Partition.



Hình 4 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.

* + 1. 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**:

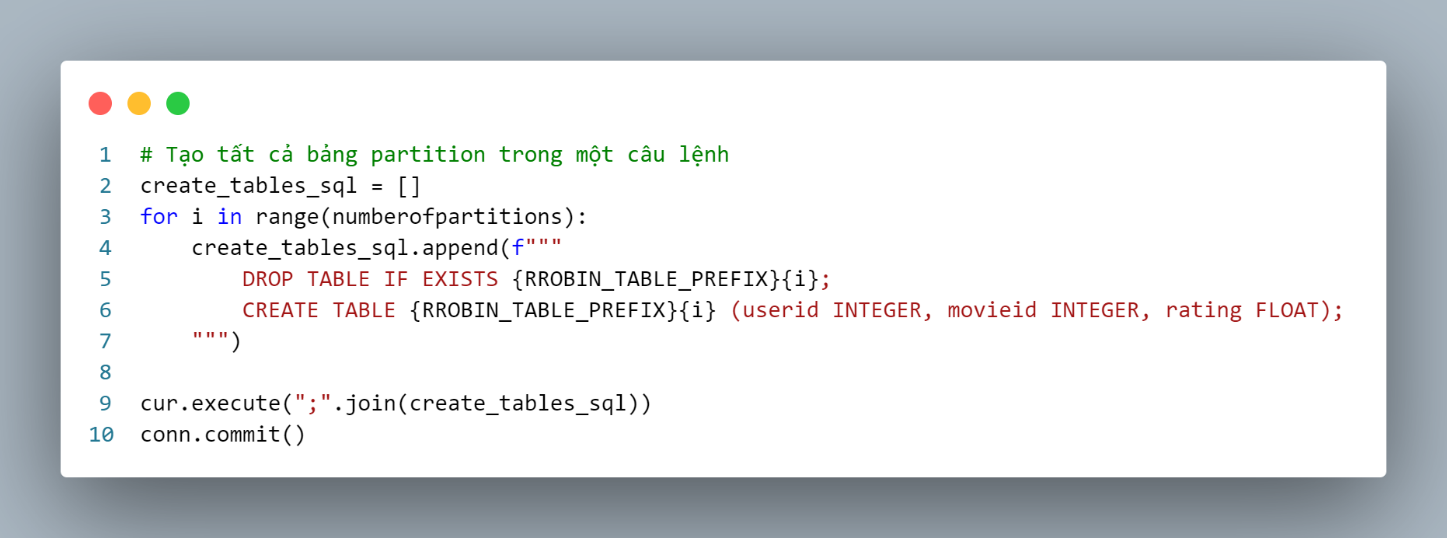
1. **Tạo bảng phân mảnh**: Xóa các bảng cũ (nếu tồn tại) và tạo N bảng mới với schema giống bảng Ratings (UserID: int, MovieID: int, Rating: float) trong một câu lệnh SQL duy nhất (Hình 5).
2. **Đếm tổng số bản ghi**: Sử dụng truy vấn SQL để lấy số lượng bản ghi trong bảng Ratings (Hình 6).
3. **Phân 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**:

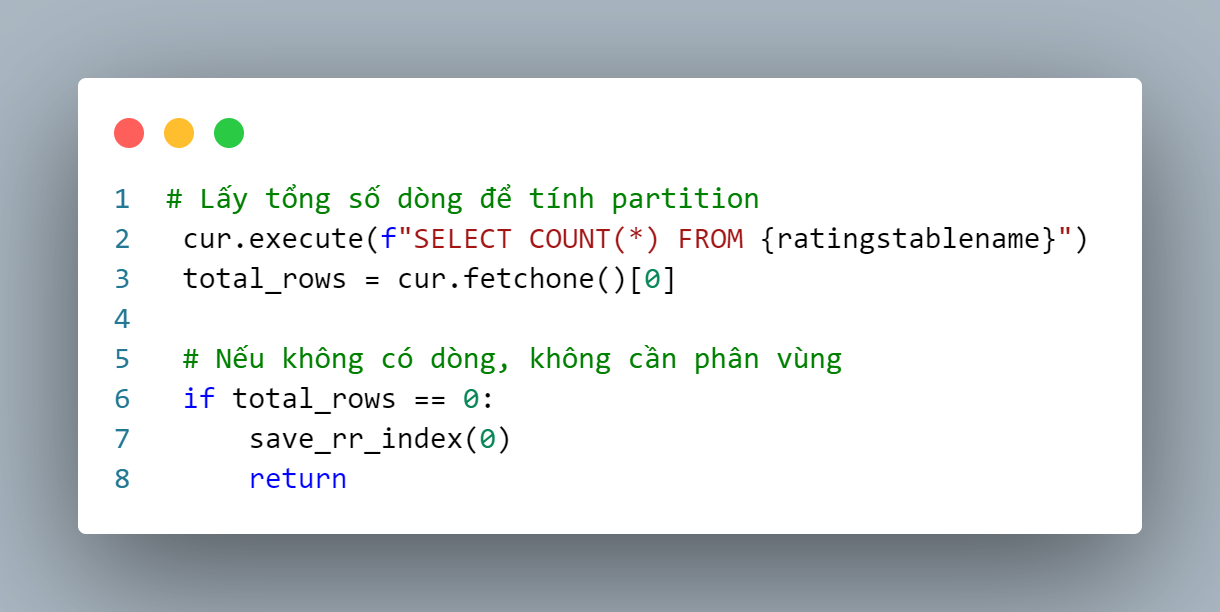
* **Phân phối đồng đều**: Đảm bảo số bản ghi trong mỗi bảng phân mảnh gần bằng nhau, đặc biệt với dữ liệu lớn (10 triệu bản ghi).
* **Hiệu suất xử lý**: Phân phối dữ liệu bằng truy vấn SQL có thể tốn thời gian khi số lượng bản ghi lớn.
* **Quản lý chỉ số vòng tròn**: Đảm bảo tệp rr\_index.txt được tạo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

**Giải pháp**:

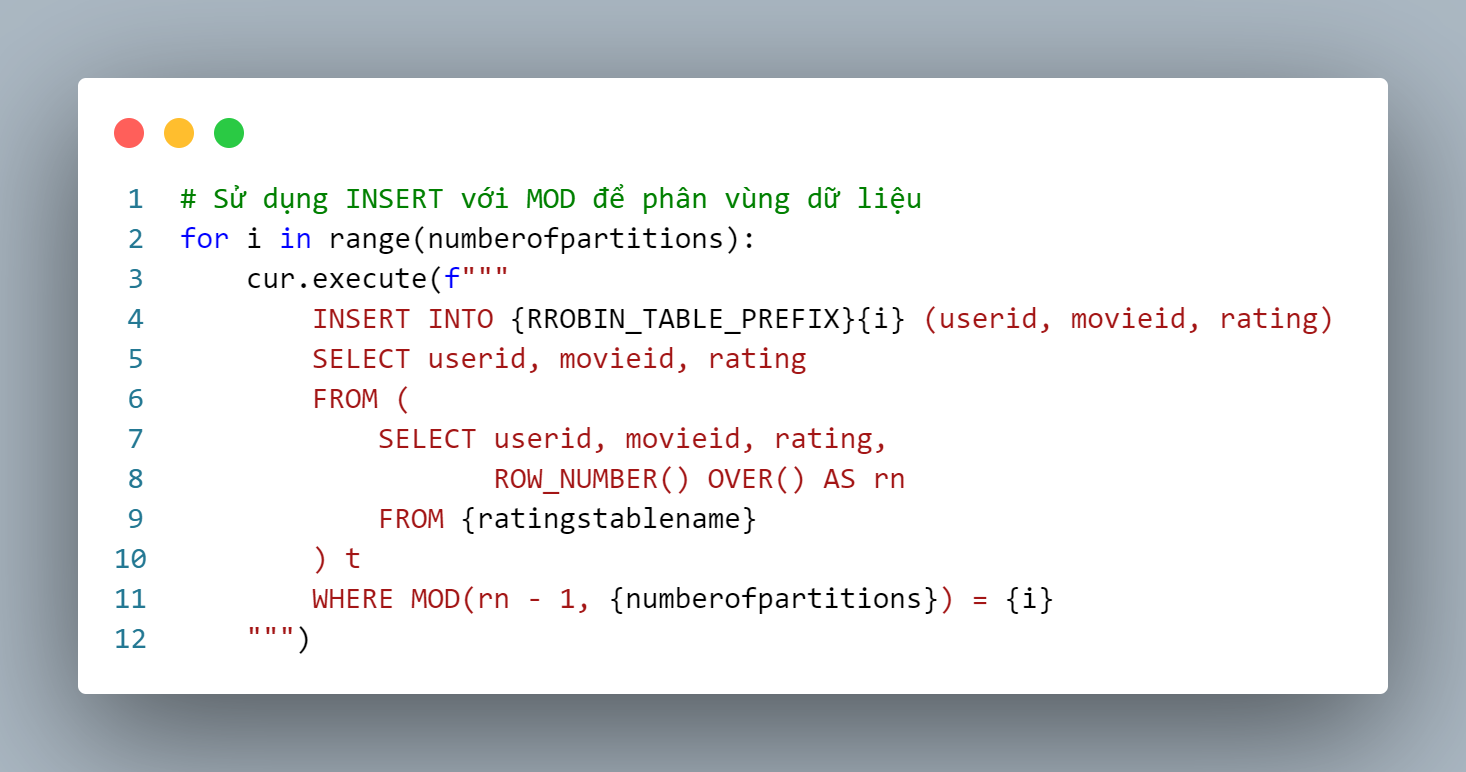
* Sử dụng ROW\_NUMBER() và MOD trong truy vấn SQL để phân phối bản ghi tuần hoàn, đảm bảo tính đồng đều mà không cần xử lý từng bản ghi trong Python (Hình 7).
* Tối ưu hóa hiệu suất bằng cách thực hiện toàn bộ quá trình phân phối trong một truy vấn SQL cho mỗi bảng, tận dụng tốc độ của PostgreSQL.
* 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).



Hình 5 Đoạn code tạo bảng phân mảnh trong hàm RoundRobin\_Partition.



Hình 6 Đoạn code đếm tổng số bản ghi và lưu chỉ số vòng tròn trong hàm RoundRobin\_Partition.



Hình 7 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 bổ 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.

* + 1. 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**:

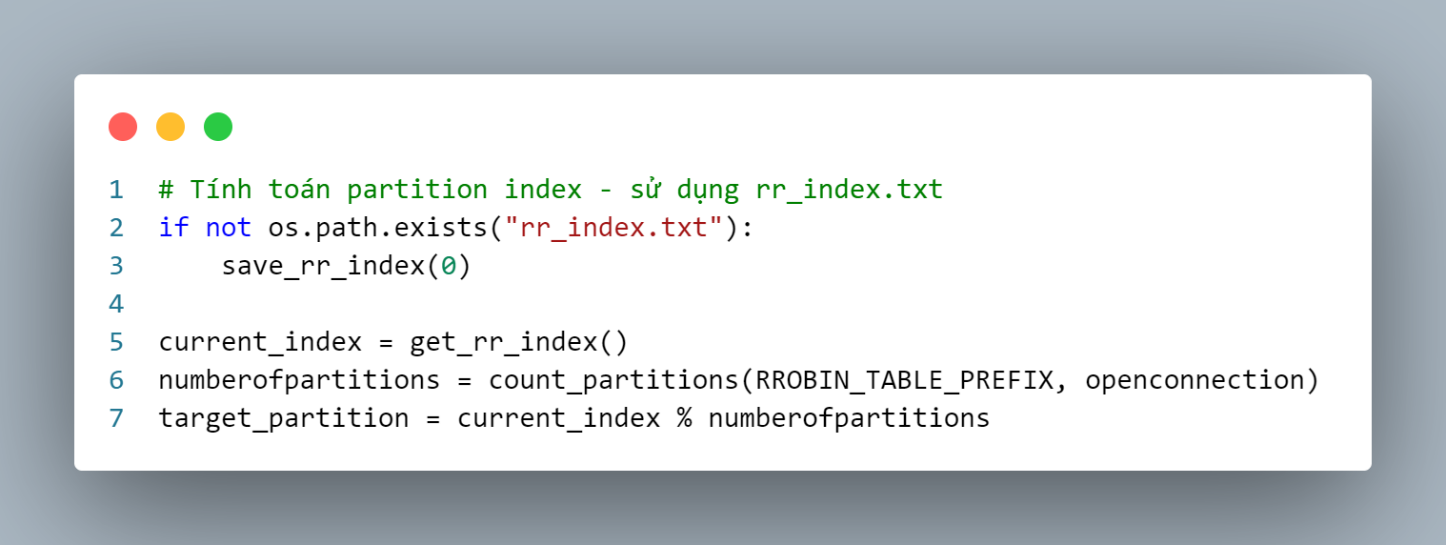
1. **Kiểm tra và khởi tạo chỉ số**: Kiểm tra sự tồn tại của tệp rr\_index.txt, khởi tạo giá trị 0 nếu tệp chưa có (Hình 8).
2. **Xác định phân mảnh đích**: Lấy chỉ số hiện tại từ rr\_index.txt và tính chỉ số phân mảnh bằng phép chia lấy dư (current\_index % numberofpartitions) (Hình 8).
3. **Chèn bản 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**:

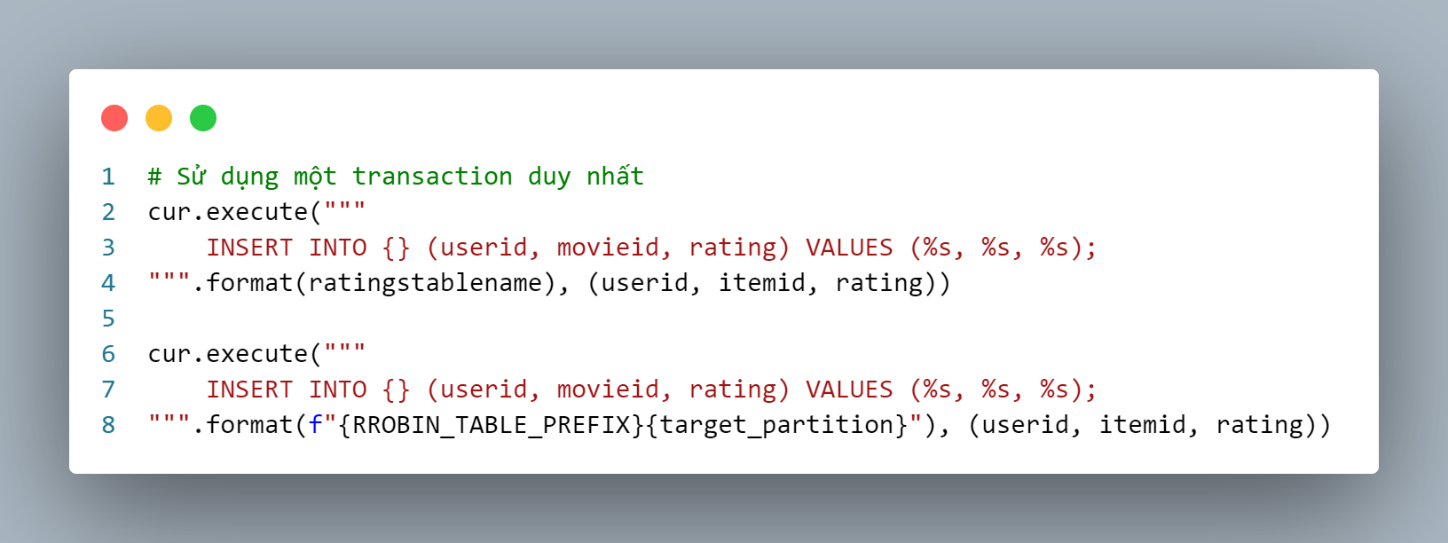
* **Quản lý chỉ số vòng tròn**: Đảm bảo tệp rr\_index.txt được đọc và ghi chính xác để duy trì thứ tự chèn tuần hoàn.
* **Tính toàn vẹn giao dịch**: Đảm bảo bản ghi được chèn đồng thời vào cả bảng Ratings và bảng phân mảnh mà không gây lỗi.
* **Hiệu suất chèn**: Chèn vào hai bảng có thể tăng thời gian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

**Giải pháp**:

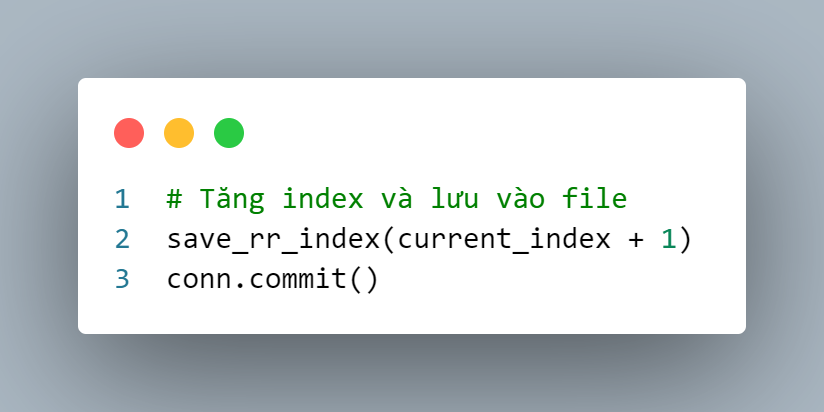
* Sử dụng hàm get\_rr\_index() và save\_rr\_index() để quản lý chỉ số vòng tròn, với kiểm tra tệp tồn tại để tránh lỗi (Hình 8).
* Thực hiện chèn trong một giao dịch duy nhất với commit và rollback để đảm bảo toàn vẹn dữ liệu (Hình 9).
* Tối ưu hóa hiệu suất bằng cách sử dụng truy vấn SQL trực tiếp với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



Hình 8 Đoạn code kiểm tra và xác định chỉ số phân mảnh trong hàm RoundRobin\_Insert.



Hình 9: Đoạn code chèn bản ghi vào bảng Ratings và bảng phân mảnh trong hàm RoundRobin\_Insert.



Hình 10 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.

* + 1. 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**:

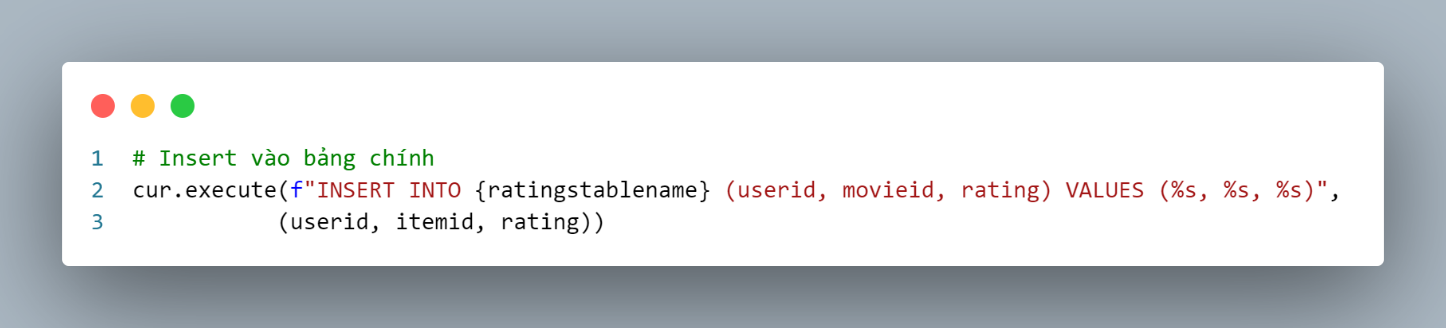
1. **Chèn vào bảng chính**: Thêm bản ghi vào bảng Ratings với các giá trị UserID, MovieID, và Rating (Hình 11).
2. **Tính toán phân mảnh đích**: Xác định số lượng phân mảnh (N) và độ dài khoảng (delta = 5.0/N). Tính chỉ số phân mảnh dựa trên giá trị Rating (Hình 12).
3. **Xử lý ranh giới**: Xử lý các trường hợp đặc biệt (Rating = 0.0 hoặc 5.0) và ranh giới giữa các khoảng để đảm bảo bản ghi được chèn đúng (Hình 12).
4. **Chèn vào bảng phân mảnh**: Thêm bản ghi vào bảng range\_partX tương ứng trong cùng giao dịch (Hình 13).

**Khó khăn**:

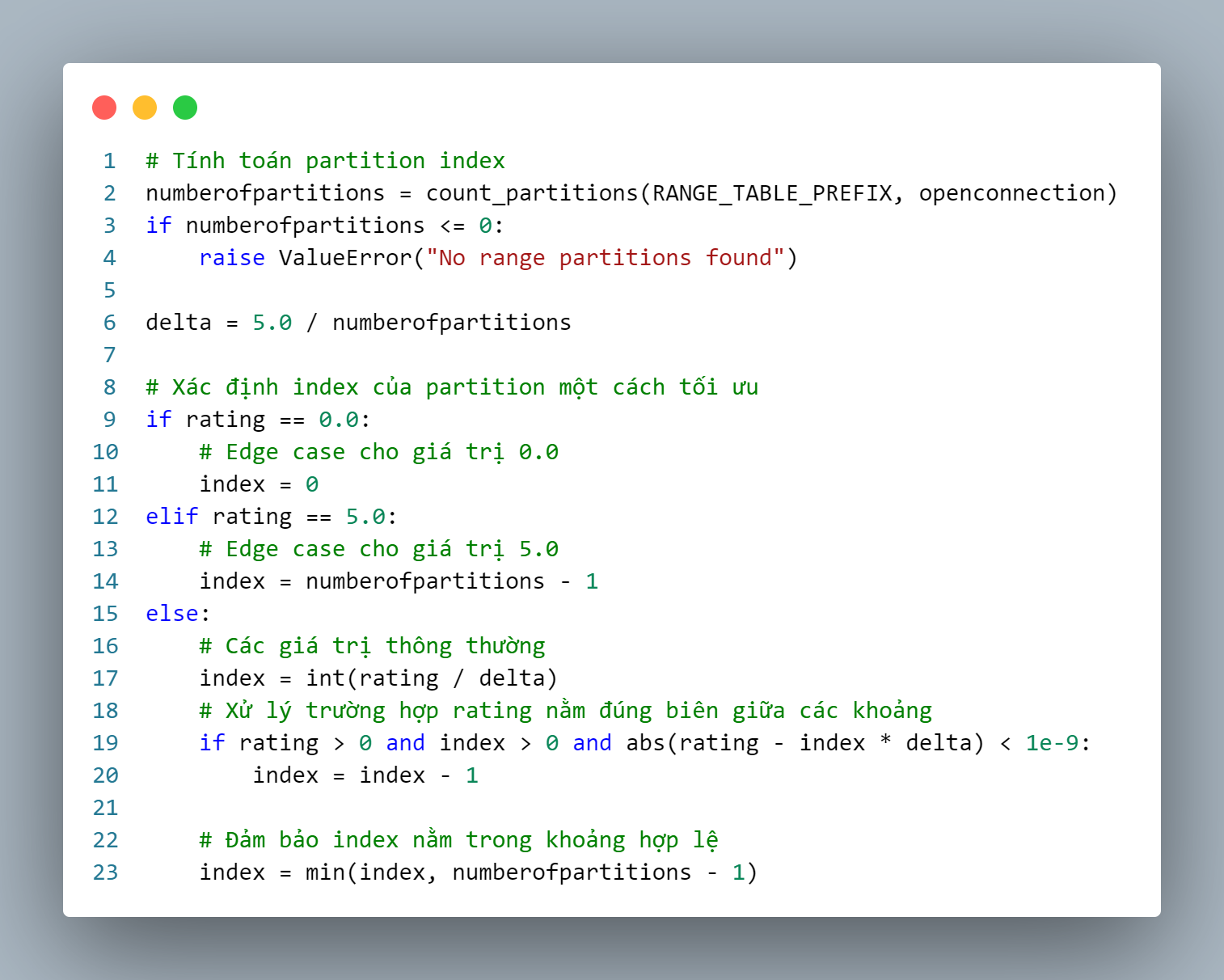
* **Xử lý ranh giới khoảng**: Đảm bảo bản ghi với Rating tại ranh giới (0.0, 5.0, hoặc giữa các khoảng) được chèn vào đúng phân mảnh.
* **Tính toàn vẹn giao dịch**: Đảm bảo bản ghi được chèn đồng thời vào cả bảng Ratings và bảng phân mảnh mà không gây lỗi.
* **Hiệu suất chèn**: Chèn vào hai bảng có thể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

**Giải pháp**:

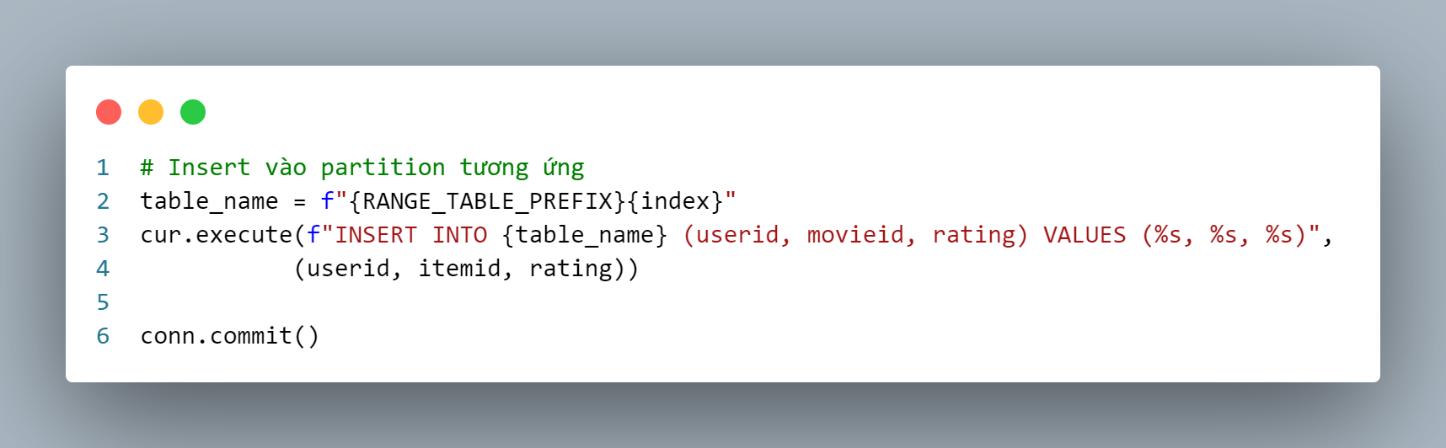
* Xử lý ranh giới bằng logic điều kiện rõ ràng: gán index = 0 cho Rating = 0.0, index = N-1 cho Rating = 5.0, và kiểm tra giá trị biên với ngưỡng 1e-9 (Hình 12).
* Thực hiện chèn trong một giao dịch duy nhất với commit và rollback để đảm bảo toàn vẹn dữ liệu (Hình 11,13).
* Tối ưu hóa hiệu suất bằng truy vấn SQL tham số hóa (%s) để tránh SQL injection và giảm thời gian xử lý.



Hình 11:Đoạn code chèn bản ghi vào bảng Ratings trong hàm Range\_Insert.



Hình 12 Đoạn code tính toán chỉ số phân mảnh trong hàm Range\_Insert.



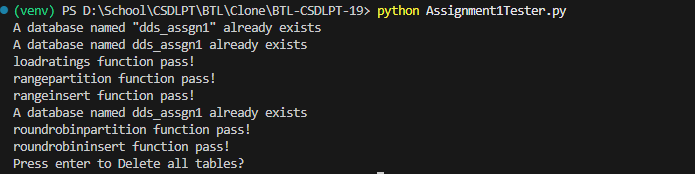
Hình 13 Đoạn code chèn bản ghi vào bảng phân mảnh trong hàm Range\_Insert.

**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, 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 E.

* 1. Kiểm tra và đánh giá
     1. Phương pháp kiểm tra

Các hàm được kiểm tra bằng các phương pháp sau:

1. **Kiểm tra tự động**: Sử dụng công cụ assignment\_tester.py (nếu có) để xác minh schema bảng Ratings (UserID: int, MovieID: int, Rating: float), số bản ghi trong các bảng phân mảnh, và tính đúng đắn của phân mảnh range và round-robin.
2. **Kiểm tra hiệu suất**: Đo thời gian thực thi trên tập dữ liệu 1000, 10,000, 50,000, và 100,000 bản ghi, với số phân mảnh N=5, 10, 20, 50, trên máy có cấu hình: Windows, 20Gb ram.
   * 1. Kết quả kiểm tra

**1. Với hàm assignment\_tester.py:  
**

Hình 14 Kết quả thu được test với assignment\_tester.py

**2.Với N = 5 trên các tệp dữ liệu**

**Bảng 3.1**: Thời gian thực thi (giây) với N=5 trên các tập dữ liệu

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hàm | 1000 bản ghi | 10,000 bản ghi | 50,000 bản ghi | 100,000 bản ghi |
| LoadRatings | 0.166 | 0.071 | 0.115 | 0.199 |
| Range\_Partition | 0.013 | 0.025 | 0.061 | 0.171 |
| RoundRobin\_Partition | 0.013 | 0.034 | 0.101 | 0.224 |
| Range\_Insert | 0.001 | 0.001 | 0.001 | 0.001 |
| RoundRobin\_Insert | 0.002 | 0.002 | 0.002 | 0.002 |

Kết quả kiểm tra (Bảng 3.1) cho thấy các hàm hoạt động đúng:

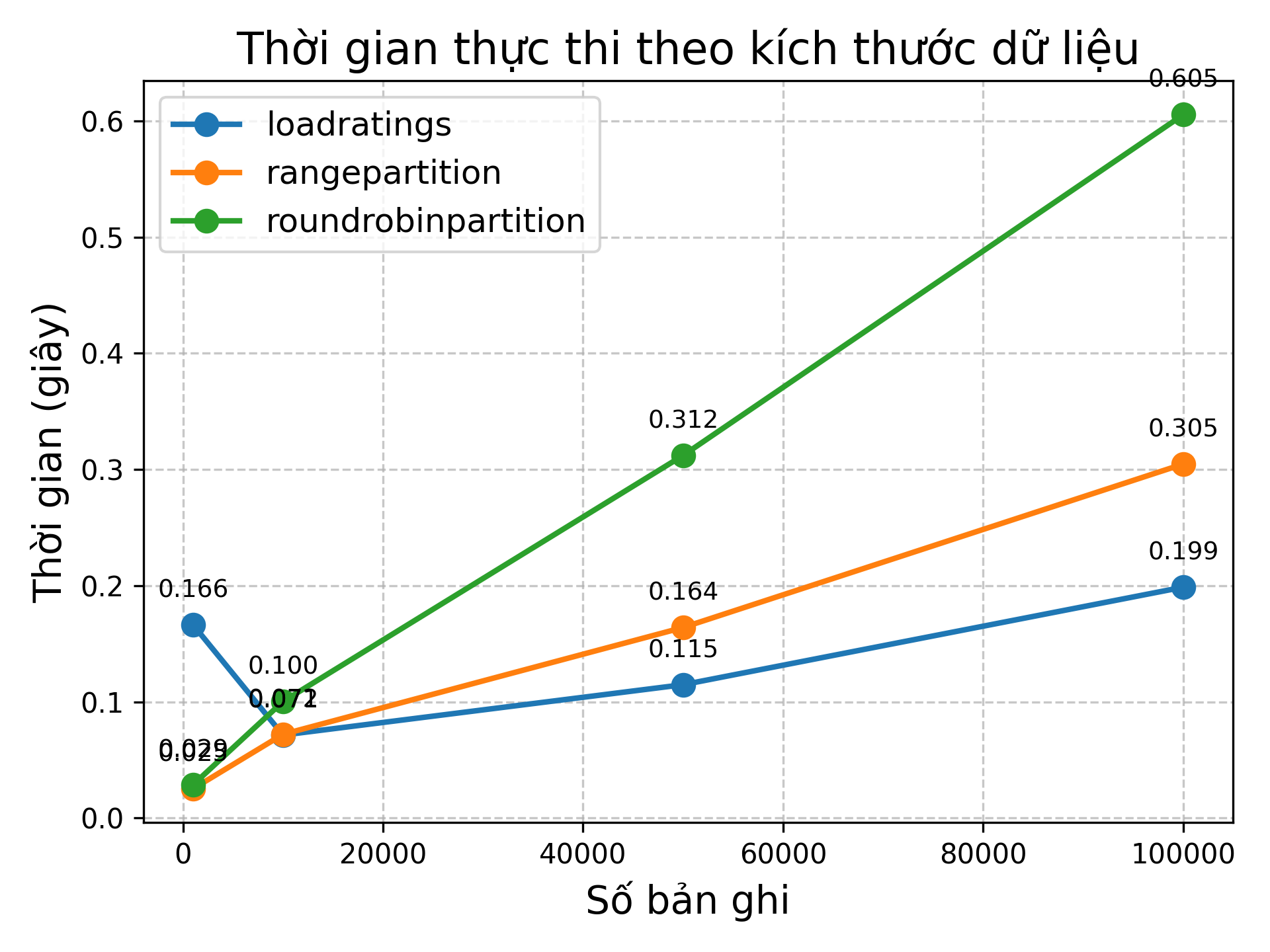
* **Hàm LoadRatings**: Tải thành công 100,000 bản ghi vào bảng Ratings trong 0.199 giây (100,000 bản ghi). Truy vấn SELECT COUNT(\*) FROM Ratings xác nhận số bản ghi khớp với đầu vào.
* **Hàm Range\_Partition**: Tạo N=5 bảng phân mảnh (range\_part0 đến range\_part4) với dữ liệu đúng khoảng Rating ([0, 1], (1, 2], ..., (4, 5]). Truy vấn SELECT MIN(Rating), MAX(Rating) xác nhận tính chính xác.
* **Hàm RoundRobin\_Partition**: Tạo N=5 bảng phân mảnh (rrobin\_part0 đến rrobin\_part4) với số bản ghi gần bằng nhau (sai lệch tối đa 1 bản ghi). Tệp rr\_index.txt lưu đúng chỉ số.
* **Hàm Range\_Insert và RoundRobin\_Insert**: Chèn bản ghi mẫu vào đúng bảng phân mảnh, xử lý đúng ranh giới (Rating = 0.0, 5.0).

Kết quả chi tiết cho N=10, 20, 50 được trình bày trong Phụ lục F. Kiểm tra không phát hiện lỗi.

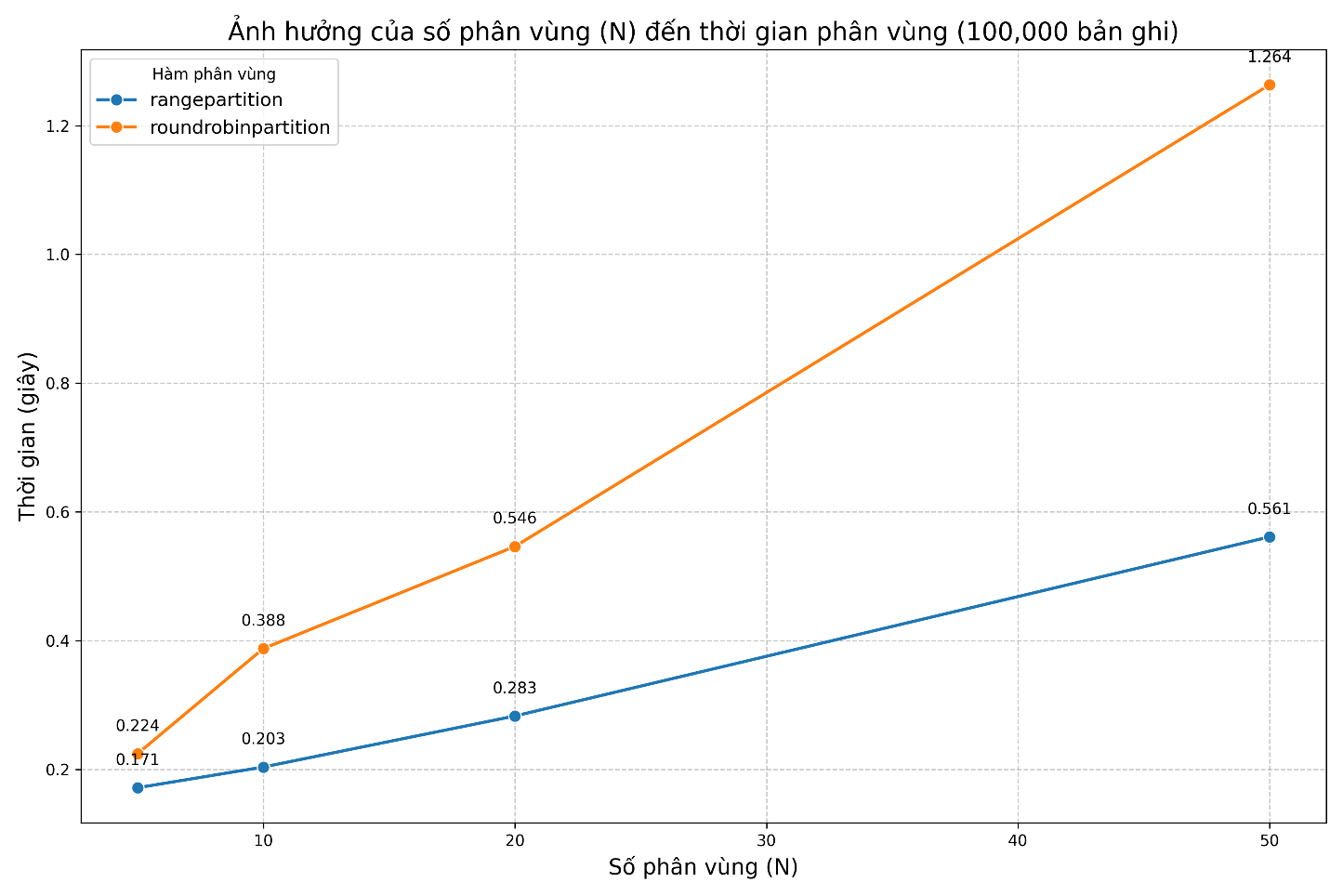
**3. Kết quả** kiểm tra với tập dữ liệu 1000, 10,000, 50,000, và 100,000 bản ghi, với số phân mảnh N=5, 10, 20, 50 trong Phụ lục F

* + 1. Đánh giá hiệu suất:

Dựa trên kết quả đánh giá với tập dữ liệu 1000, 10,000, 50,000, và 100,000 bản ghi và số phân mảnh N=5, 10, 20, 50 (trong phụ lục F). Ta có các biểu đồ thống kê sau:



Hình 15: So sánh thời gian thực thi theo kích thước dữ liệu



Hình 16: Ảnh hưởng của số phân vùng (N) đến thời gian phân vùng

Hiệu suất của các hàm được đánh giá qua thời gian thực thi trên các tập dữ liệu (1000 đến 100,000 bản ghi) với số phân vùng N=5, 10, 20, 50 (Hình 17,18).

**Ảnh hưởng kích thước dữ liệu** (Hình 15): Khi số bản ghi tăng từ 1000 lên 100,000:

* LoadRatings: Thời gian tăng nhẹ (0.166s lên 0.199s), gần O(1) nhờ lệnh COPY.
* Range\_Partition: Tăng khoảng 13 lần (0.013s lên 0.171s), gần O(n).
* RoundRobin\_Partition: Tăng khoảng 17 lần (0.013s lên 0.224s), chịu ảnh hưởng mạnh từ kích thước dữ liệu.

**Ảnh hưởng số phân vùng** (Hình 16): Với 100,000 bản ghi, khi N tăng từ 5 lên 50:

* Range\_Partition: Thời gian tăng 3.3 lần (0.171s lên 0.561s).
* RoundRobin\_Partition: Tăng 5.6 lần (0.224s lên 1.264s), chậm hơn Range\_Partition (tỉ lệ chênh lệch tăng từ 1.3 lên 2.2).

**Hàm chèn dữ liệu**: Range\_Insert và RoundRobin\_Insert có thời gian thực thi thấp (~0.001-0.002s), ít bị ảnh hưởng bởi N hoặc kích thước dữ liệu. RoundRobin\_Insert chậm hơn Range\_Insert khoảng 1.5 lần.

**Nhận xét**:

* Range\_Partition hiệu quả hơn RoundRobin\_Partition, đặc biệt khi N lớn, do truy vấn SQL đơn giản hơn.
* LoadRatings tối ưu nhờ COPY và StringIO.
* Nên dùng N nhỏ (<20) để cân bằng hiệu suất.
* Thiết kế rr\_index.txt giúp RoundRobin\_Insert hiệu quả.

1. Khó khăn và giải pháp
   1. Khó khăn kỹ thuật

Trong quá trình thực hiện bài tập lớn, nhóm chúng em đã gặp phải một số khó khăn kỹ thuật đáng kể:

**1. Hiệu suất với dữ liệu lớn:**

Tập dữ liệu MovieLens chứa 10 triệu đánh giá, gây áp lực lớn lên hệ thống

Các thuật toán phân mảnh ban đầu quét bảng nhiều lần, dẫn đến thời gian xử lý kéo dài

Những truy vấn lặp đi lặp lại tạo ra chi phí I/O và tài nguyên CPU cao

**2. Xử lý chính xác các trường hợp biên:**

Đảm bảo giá trị 0.0 và 5.0 được phân loại đúng vào các phân mảnh tương ứng

Xử lý các giá trị nằm chính xác trên biên giữa các khoảng (ví dụ: 1.67, 3.34)

Tránh sai số trong quá trình tính toán các khoảng phân mảnh

**3. Quản lý bộ nhớ và tài nguyên:**

Với dữ liệu lớn, việc tối ưu sử dụng bộ nhớ trở nên quan trọng

Tránh giữ dữ liệu quá lâu trong bộ nhớ khi không cần thiết

Nguy cơ tràn bộ nhớ khi xử lý 10 triệu bản ghi cùng lúc

**4. Tương thích với các yêu cầu đề bài:**

Đảm bảo tuân thủ các ràng buộc về tên bảng, cấu trúc, và quy tắc phân mảnh

Không được đóng kết nối trong các hàm hoặc mã hóa cứng các thông số

Thực hiện đúng phân mảnh theo khoảng đồng đều với N phân mảnh

**5. Khó khăn trong kiểm thử:**

Đảm bảo các testcase phản ánh đúng tính năng của hệ thống

Kiểm tra đầy đủ các trường hợp đặc biệt mà không cần duyệt toàn bộ dữ liệu

* 1. Giải pháp

Để giải quyết các khó khăn trên, chúng em đã áp dụng các giải pháp sau:

1. **Cải thiện thuật toán phân mảnh:**

* Quét bảng một lần duy nhất: Thay vì quét bảng gốc nhiều lần (mỗi lần cho một phân mảnh), chúng em thiết kế thuật toán chỉ quét một lần duy nhất để phân loại dữ liệu.
* Sử dụng CASE WHEN: Áp dụng cấu trúc CASE WHEN trong SQL để phân loại dữ liệu trong một lần quét, giảm thiểu chi phí I/O.
* Bảng phân loại tạm thời: Tạo bảng tạm chứa thông tin về phân mảnh để tối ưu quá trình chèn dữ liệu.

1. **Tối ưu hóa truy cập dữ liệu:**

* Sử dụng chỉ mục: Tạo chỉ mục tạm thời trên cột rating để tăng tốc độ truy vấn.
* Batch processing: Xử lý dữ liệu theo lô thay vì từng bản ghi riêng lẻ.
* Tham số hóa truy vấn SQL: Tránh SQL injection và tăng hiệu suất thực thi truy vấn.

1. **Xử lý chính xác các trường hợp biên:**

* Áp dụng xử lý đặc biệt cho phân mảnh đầu tiên và cuối cùng để đảm bảo giá trị 0.0 và 5.0 được xử lý đúng.
* Sử dụng so sánh với độ chính xác cao để tránh sai số phân loại do làm tròn số.

1. **Quản lý tài nguyên hiệu quả:**

* Sử dụng StringIO để xử lý dữ liệu trong bộ nhớ trước khi chuyển vào cơ sở dữ liệu.
* Giải phóng tài nguyên (như bảng tạm, chỉ mục) ngay sau khi sử dụng xong.
* Xử lý ngoại lệ và rollback khi xảy ra lỗi để tránh trạng thái không nhất quán.

1. **Kiểm soát và hậu xử lý:**

* Bổ sung cơ chế kiểm tra đầu vào để phát hiện sớm các giá trị không hợp lệ.
* Thực hiện commit transaction sau mỗi thao tác phân mảnh lớn để giảm thiểu tác động của lỗi.
  1. Bài học kinh nghiệm

Qua quá trình thực hiện bài tập lớn này, nhóm chúng em đã rút ra được những bài học kinh nghiệm quý báu:

1. **Thiết kế trước khi triển khai:**

* Phân tích kỹ yêu cầu và hiểu rõ các ràng buộc từ đề bài trước khi viết code.
* Vẽ sơ đồ và lập kế hoạch cho các thuật toán phức tạp.
* Dự đoán trước các trường hợp đặc biệt và cách xử lý.

1. **Tầm quan trọng của tối ưu hóa cơ sở dữ liệu:**

* Việc tối ưu truy vấn SQL có tác động lớn đến hiệu suất tổng thể.
* Chỉ mục đóng vai trò quan trọng trong việc tăng tốc truy vấn với dữ liệu lớn.
* Một truy vấn SQL tốt có thể thay thế nhiều dòng code xử lý trong Python.

1. **Cân nhắc giữa hiệu suất và độ phức tạp:**

* Không phải lúc nào giải pháp phức tạp nhất cũng là tốt nhất.
* Đôi khi một thuật toán đơn giản nhưng được tối ưu tốt có thể hiệu quả hơn.
* Cần cân nhắc giữa việc tăng hiệu suất và khả năng bảo trì code.

1. **Tầm quan trọng của việc kiểm thử:**

* Kiểm thử nhiều trường hợp, đặc biệt là các trường hợp biên.
* Sử dụng dữ liệu thử nghiệm đại diện cho các tình huống thực tế.
* Đánh giá hiệu suất dựa trên dữ liệu thực tế, không chỉ dựa vào lý thuyết.

1. **Khả năng mở rộng và bảo trì:**

* Thiết kế code có thể dễ dàng mở rộng cho các yêu cầu mới.
* Viết code sạch, có tài liệu đầy đủ để người khác có thể hiểu và bảo trì.
* Tách biệt các chức năng để dễ dàng sửa đổi từng phần khi cần thiết.

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.

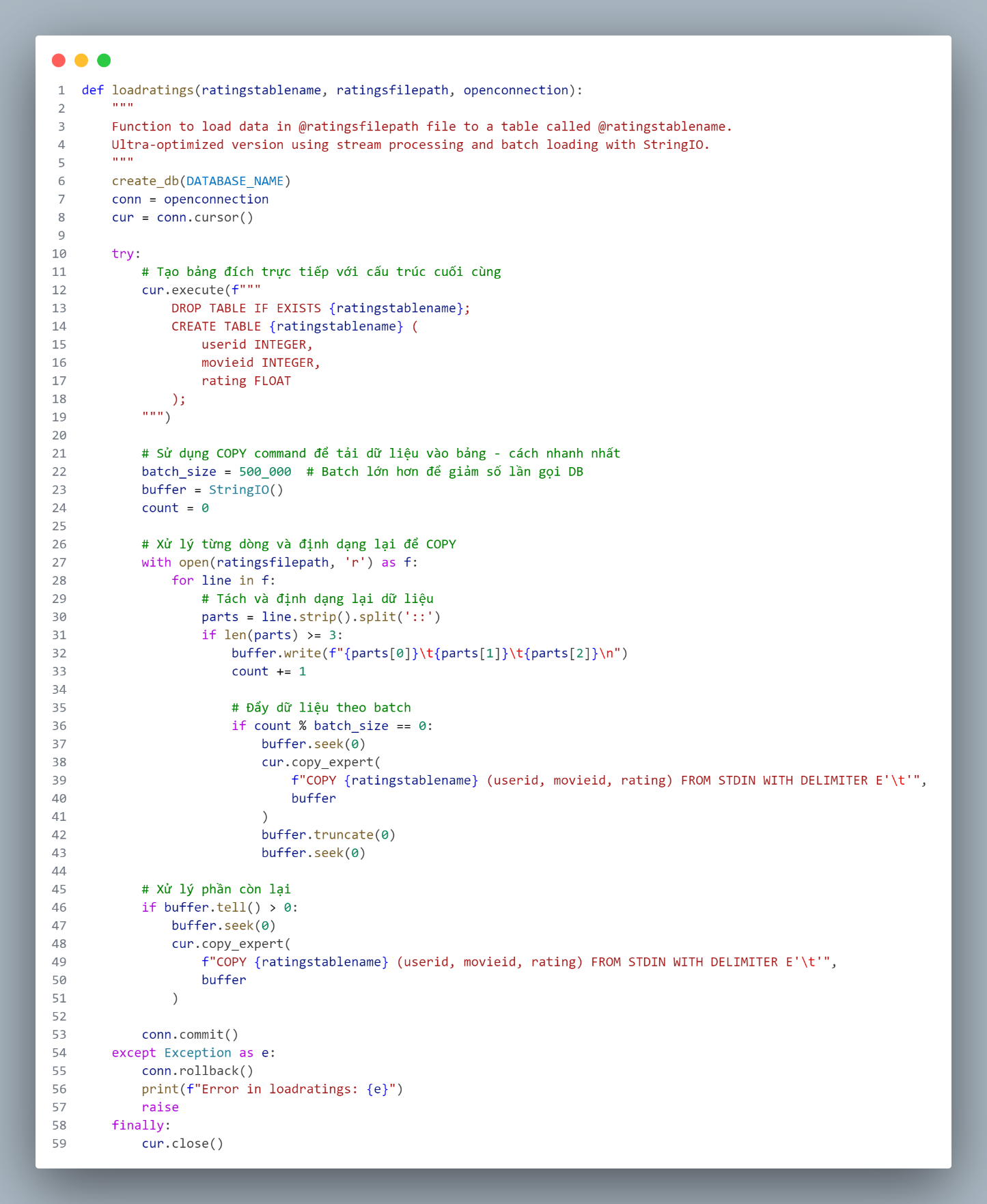
Ngoài việc đáp ứng các yêu cầu cụ thể của bài tập, dự án này còn mang lại cho chúng em kiến thức và kỹ năng quý báu về tối ưu hóa cơ sở dữ liệu, điều mà chắc chắn sẽ hữu ích trong các dự án thực tế trong tương lai. Chúng em cũng nhận thấy tầm quan trọng của việc cân nhắc kỹ lưỡng giữa hiệu suất và độ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

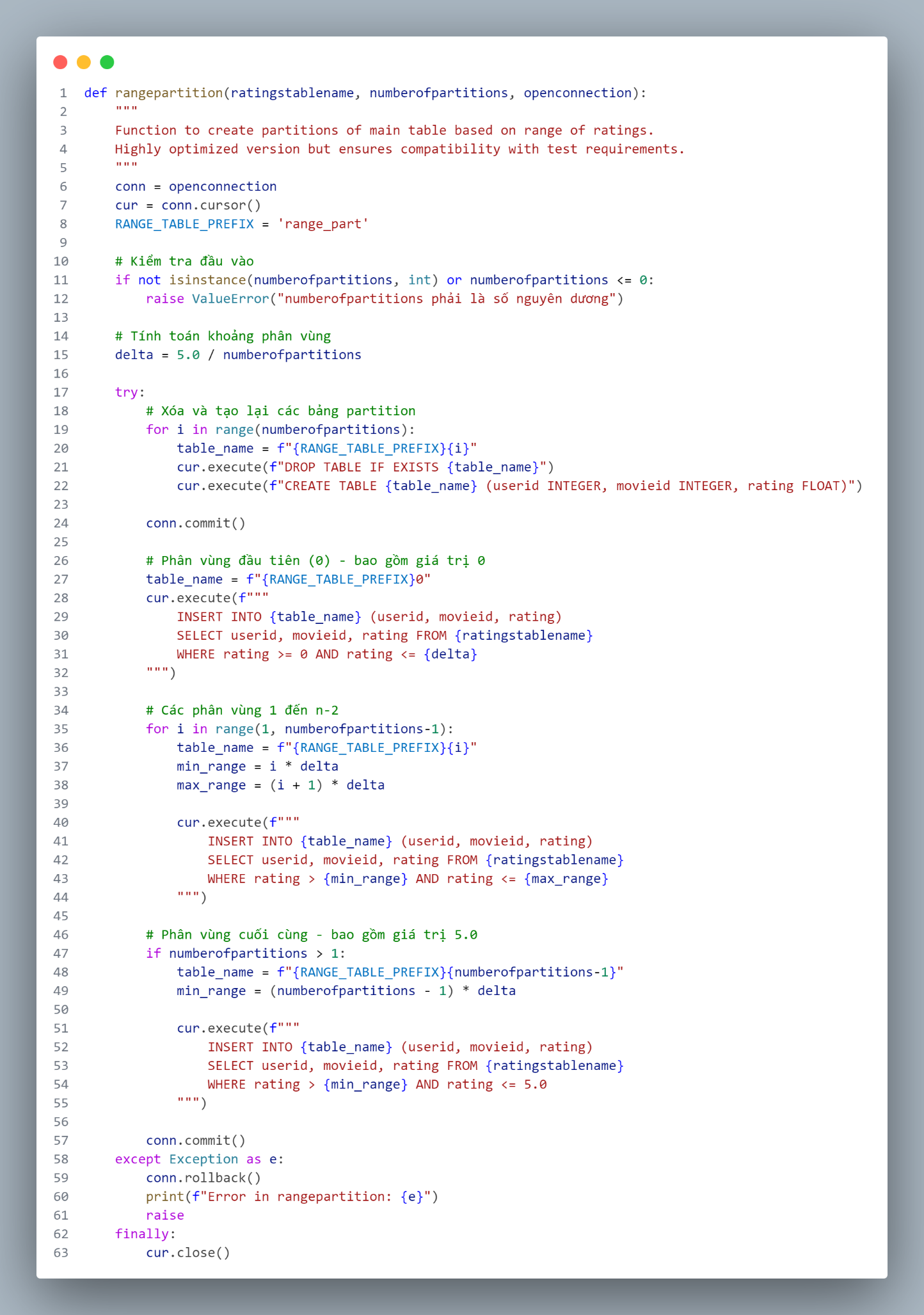
Cuối cùng, dự án này đã thành công trong việc minh họa cách áp dụng các nguyên lý cơ bản của cơ sở dữ liệu phân tán vào thực tế, từ đó nâng cao hiểu biết và kỹ năng của chúng em trong lĩnh vực quản lý và xử lý dữ liệu quy mô lớn.

Phụ lục

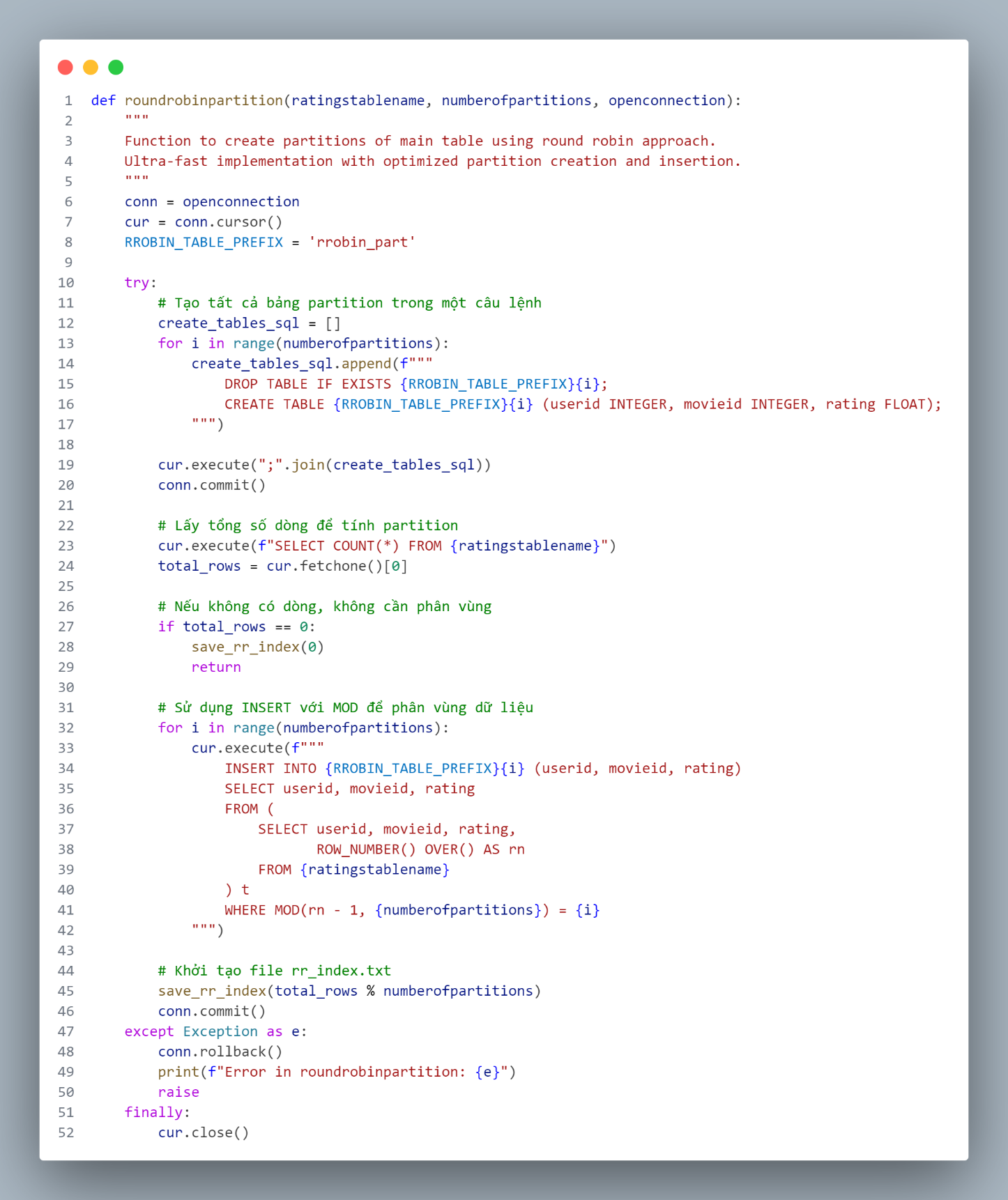
Phụ lục A: Mã nguồn đầy đủ hàm Hàm LoadRatings():



Phụ lục B: Mã nguồn hàm Range\_Partition():



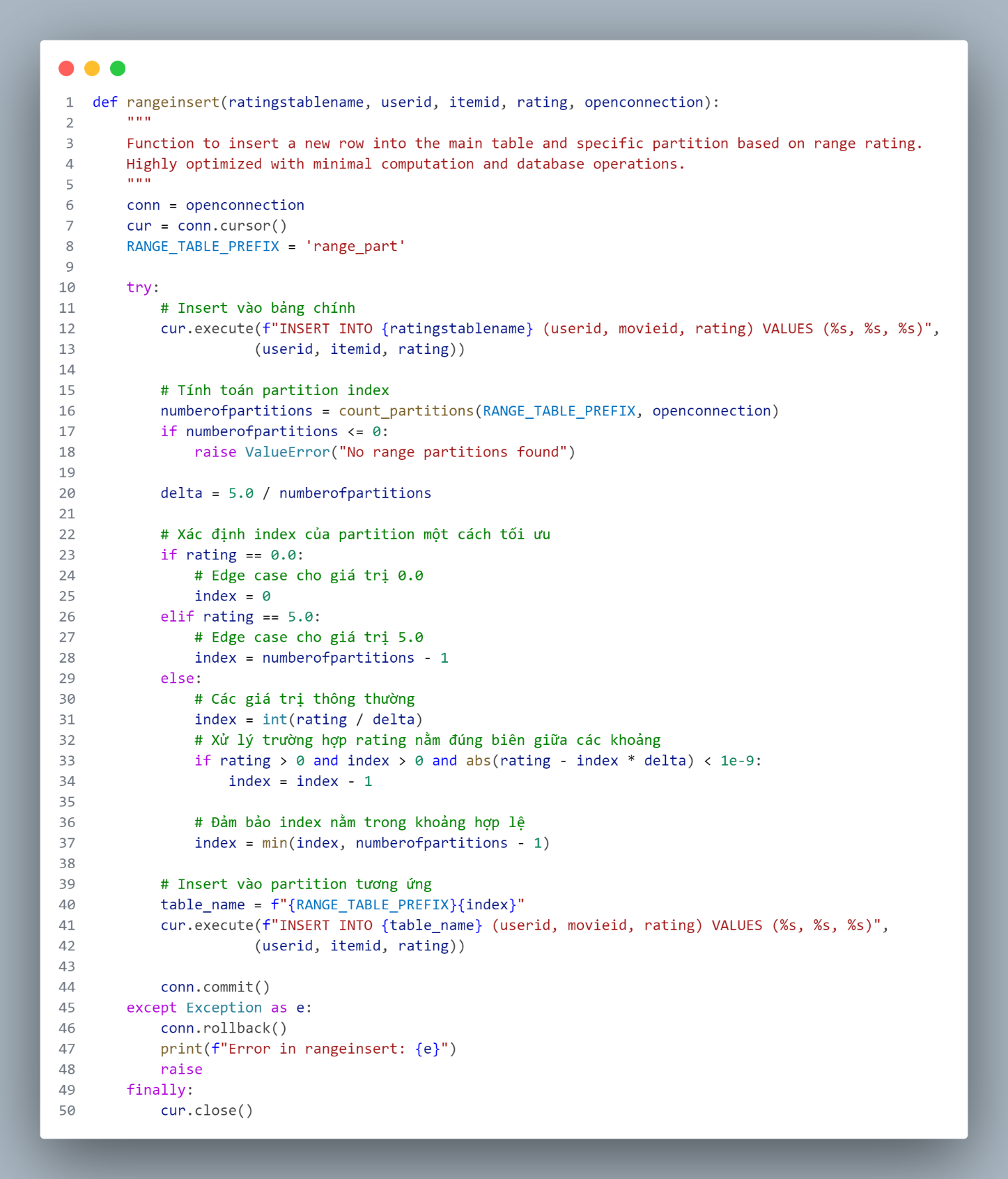
Phụ lục C: Mã nguồn hàm RoundRobin\_Partition():



Phụ lục D: Mã nguồn hàm RoundRobin\_Insert():



Phụ lục E: Mã nguồn hàm Range\_Insert():



Phụ lục F: Bảng thống kê thời gian chạy các hàm với tập dữ liệu 1000, 10,000, 50,000, và 100,000 bản ghi, với số phân mảnh N=5, 10, 20, 50

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **num\_records** | **Time(s)** | **partitions** |
| loadratings | 1000 | 0.166472 | N/A |
| rangepartition | 1000 | 0.013158 | 5 |
| roundrobinpartition | 1000 | 0.013214 | 5 |
| rangeinsert | 1000 | 0.001285 | 5 |
| roundrobininsert | 1000 | 0.001639 | 5 |
| rangepartition | 1000 | 0.014436 | 10 |
| roundrobinpartition | 1000 | 0.015572 | 10 |
| rangeinsert | 1000 | 0.001222 | 10 |
| roundrobininsert | 1000 | 0.001882 | 10 |
| rangepartition | 1000 | 0.023952 | 20 |
| roundrobinpartition | 1000 | 0.024747 | 20 |
| rangeinsert | 1000 | 0.001234 | 20 |
| roundrobininsert | 1000 | 0.001755 | 20 |
| rangepartition | 1000 | 0.049135 | 50 |
| roundrobinpartition | 1000 | 0.060879 | 50 |
| rangeinsert | 1000 | 0.001268 | 50 |
| roundrobininsert | 1000 | 0.002025 | 50 |
| loadratings | 10000 | 0.071462 | N/A |
| rangepartition | 10000 | 0.024713 | 5 |
| roundrobinpartition | 10000 | 0.033633 | 5 |
| rangeinsert | 10000 | 0.001298 | 5 |
| roundrobininsert | 10000 | 0.001658 | 5 |
| rangepartition | 10000 | 0.029972 | 10 |
| roundrobinpartition | 10000 | 0.047095 | 10 |
| rangeinsert | 10000 | 0.001237 | 10 |
| roundrobininsert | 10000 | 0.001963 | 10 |
| rangepartition | 10000 | 0.067078 | 20 |
| roundrobinpartition | 10000 | 0.094627 | 20 |
| rangeinsert | 10000 | 0.001711 | 20 |
| roundrobininsert | 10000 | 0.002232 | 20 |
| rangepartition | 10000 | 0.166593 | 50 |
| roundrobinpartition | 10000 | 0.225824 | 50 |
| rangeinsert | 10000 | 0.001334 | 50 |
| roundrobininsert | 10000 | 0.002252 | 50 |
| loadratings | 50000 | 0.114791 | N/A |
| rangepartition | 50000 | 0.061325 | 5 |
| roundrobinpartition | 50000 | 0.101202 | 5 |
| rangeinsert | 50000 | 0.001265 | 5 |
| roundrobininsert | 50000 | 0.001883 | 5 |
| rangepartition | 50000 | 0.094128 | 10 |
| roundrobinpartition | 50000 | 0.169889 | 10 |
| rangeinsert | 50000 | 0.001444 | 10 |
| roundrobininsert | 50000 | 0.002103 | 10 |
| rangepartition | 50000 | 0.148706 | 20 |
| roundrobinpartition | 50000 | 0.303718 | 20 |
| rangeinsert | 50000 | 0.001504 | 20 |
| roundrobininsert | 50000 | 0.002475 | 20 |
| rangepartition | 50000 | 0.351965 | 50 |
| roundrobinpartition | 50000 | 0.673636 | 50 |
| rangeinsert | 50000 | 0.001353 | 50 |
| roundrobininsert | 50000 | 0.002147 | 50 |
| loadratings | 100000 | 0.198724 | N/A |
| rangepartition | 100000 | 0.17133 | 5 |
| roundrobinpartition | 100000 | 0.224107 | 5 |
| rangeinsert | 100000 | 0.001239 | 5 |
| roundrobininsert | 100000 | 0.00197 | 5 |
| rangepartition | 100000 | 0.20349 | 10 |
| roundrobinpartition | 100000 | 0.387605 | 10 |
| rangeinsert | 100000 | 0.001313 | 10 |
| roundrobininsert | 100000 | 0.002057 | 10 |
| rangepartition | 100000 | 0.282789 | 20 |
| roundrobinpartition | 100000 | 0.546242 | 20 |
| rangeinsert | 100000 | 0.001722 | 20 |
| roundrobininsert | 100000 | 0.001852 | 20 |
| rangepartition | 100000 | 0.561176 | 50 |
| roundrobinpartition | 100000 | 1.263558 | 50 |
| rangeinsert | 100000 | 0.001527 | 50 |
| roundrobininsert | 100000 | 0.002273 | 50 |