

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



BÀI TẬP LỚN
CƠ SỞ DỮ LIỆU PHÂN TÁN

Đề tài: Mô phỏng các phương pháp phân mảnh dữ liệu trên một hệ quản trị cơ sở dữ liệu quan hệ mã nguồn mở PostgreSQL

Nhóm: 14

Sinh viên thực hiện

| Tên thành viên | Mã sinh viên |
|--------------------------|---------------------|
| Lý Tuấn Anh | B22DCCN022 |
| Nguyễn Thị Cẩm Ly | B22DCCN541 |
| Trần Đức Trung | B22DCCN875 |

Giảng viên hướng dẫn: Kim Ngọc Bách

Hà Nội, 2025

MỤC LỤC

| | |
|-------------------------------------------------------------|----|
| MỤC LỤC..... | 2 |
| PHÂN CHIA CÔNG VIỆC GIỮA CÁC THÀNH VIÊN..... | 4 |
| CHƯƠNG 1. GIỚI THIỆU..... | 5 |
| 1.1 Mục tiêu bài tập..... | 5 |
| 1.2 Phạm vi thực hiện..... | 5 |
| 1.3 Yêu cầu chức năng..... | 5 |
| CHƯƠNG 2. CƠ SỞ LÝ THUYẾT..... | 6 |
| 2.1 Khái niệm phân mảnh dữ liệu..... | 6 |
| 2.2 Phân mảnh ngang (Horizontal Partitioning)..... | 7 |
| 2.3 Range Partitioning..... | 7 |
| 2.3.1 Khái niệm và cách thức hoạt động..... | 7 |
| 2.3.2 Thuật toán phân chia theo khoảng giá trị..... | 8 |
| 2.3.3 Ưu điểm: Truy vấn theo khoảng hiệu quả..... | 8 |
| 2.3.4 Nhược điểm: Có thể gây mất cân bằng dữ liệu..... | 9 |
| 2.4 Round Robin Partitioning..... | 10 |
| 2.4.1 Khái niệm và cách thức hoạt động..... | 10 |
| 2.4.2 Thuật toán phân chia tuần tự..... | 11 |
| 2.4.3 Ưu điểm: Phân bố dữ liệu đồng đều..... | 13 |
| 2.4.4 Nhược điểm: Khó tối ưu cho truy vấn có điều kiện..... | 15 |
| CHƯƠNG 3. PHÂN TÍCH VÀ THIẾT KẾ..... | 18 |
| 3.1 Phân tích dữ liệu đầu vào..... | 18 |
| 3.1.1 Cấu trúc file ratings.dat..... | 18 |
| 3.1.2 Schema bảng Ratings trong CSDL..... | 18 |
| 3.2 Thiết kế kiến trúc hệ thống..... | 20 |
| 3.2.1 Sơ đồ tổng quan hệ thống..... | 20 |
| 3.2.2 Mô hình kết nối CSDL..... | 21 |
| 3.2.3 Cấu trúc thư mục và module Python..... | 21 |
| 3.3 Thiết kế các hàm chính..... | 21 |
| 3.3.1 Hàm LoadRatings()..... | 21 |
| 3.3.2 Hàm Range_Partition()..... | 21 |
| 3.3.3 Hàm RoundRobin_Partition()..... | 22 |
| 3.3.4 Hàm Range_Insert() và RoundRobin_Insert()..... | 22 |
| CHƯƠNG 4. TRIỂN KHAI VÀ CÀI ĐẶT..... | 23 |
| 4.1 Môi trường phát triển..... | 23 |
| 4.2 Cài đặt các hàm chức năng..... | 23 |

| | |
|----------------------------------------------------------|----|
| 4.2.1. Xử lý kết nối database..... | 23 |
| 4.2.2 Hàm tải xuống dữ liệu đầu vào..... | 24 |
| 4.2.3 Hàm LoadRatings()..... | 26 |
| Phương pháp 1: COPY Command (file > 50MB)..... | 28 |
| Phương pháp 2: Parallel Batch Insert..... | 28 |
| Phương pháp 3: Optimized Batch Insert (file nhỏ)..... | 28 |
| 4.2.4 Hàm Range_Partition()..... | 31 |
| 4.2.5 Hàm RoundRobin_Partition()..... | 33 |
| 4.2.6 Hàm Range_Insert()..... | 36 |
| 4.2.7 Hàm RoundRobin_Insert()..... | 40 |
| CHƯƠNG 5. KIỂM THỬ VÀ ĐÁNH GIÁ..... | 42 |
| 5.1 Kết quả kiểm thử..... | 42 |
| 5.1.1 Test với dữ liệu mẫu nhỏ (file test_data.dat)..... | 42 |
| 5.1.2 Test với dữ liệu đầy đủ 10 triệu đánh giá..... | 45 |
| 5.2 Đánh giá hiệu suất..... | 50 |

PHÂN CHIA CÔNG VIỆC GIỮA CÁC THÀNH VIÊN

| STT | Sinh viên thực hiện | Công việc/ Nhiệm vụ |
|-----|---------------------|-------------------------------------------------------------------------------------------|
| 1 | Trần Đức Trung | Thiết kế database, triển khai hàm LoadRatings() Viết báo cáo: Chương 1, 2, 3, 4 |
| 2 | Lý Tuấn Anh | Triển khai Range_Partition() và Range_Insert() Viết báo cáo: Chương 5 |
| 3 | Nguyễn Thị Cẩm Ly | Triển khai RoundRobin_Partition() RoundRobin_Insert() Viết báo cáo: Chương 5 |

CHƯƠNG 1. GIỚI THIỆU

1.1 Mục tiêu bài tập

- Mô phỏng các phương pháp phân mảnh dữ liệu trên CSDL quan hệ mã nguồn mở (PostgreSQL)
- Triển khai các kỹ thuật phân mảnh ngang (horizontal partitioning)
- Thực hành với dữ liệu thực tế từ MovieLens: <http://movielens.org/>

1.2 Phạm vi thực hiện

- Sử dụng PostgreSQL làm hệ quản trị CSDL
- Môi trường: máy ảo Ubuntu 16.04
- Ngôn ngữ lập trình: Python 3.12.3
- Dữ liệu: Tập đánh giá phim MovieLens (khoảng 10 triệu bản ghi)

1.3 Yêu cầu chức năng

- Tải dữ liệu từ file vào CSDL
- Phân mảnh theo phương pháp Range Partition
- Phân mảnh theo phương pháp Round Robin
- Chèn dữ liệu mới vào đúng phân mảnh

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1 Khái niệm phân mảnh dữ liệu

- Phân phối quan hệ (Distributing Relations): Trong một số trường hợp, việc chỉ phân phối toàn bộ các quan hệ (relations) mà không phân mảnh có thể không hiệu quả. Điều này có thể dẫn đến việc dữ liệu không được phân bổ một cách tối ưu, gây ra tình trạng nghẽn cổ chai và giảm hiệu suất.
- Đơn vị phân phối hợp lý (Reasonable Unit of Distribution)
 - Quan hệ (Relation): Phân phối toàn bộ quan hệ có thể đơn giản nhưng không phải lúc nào cũng hiệu quả.
 - Các mảnh của quan hệ (Fragments of Relations): Phân phối các mảnh nhỏ hơn của quan hệ (sub-relations) có thể cải thiện hiệu suất và tận dụng tốt hơn tài nguyên hệ thống.
- Lợi ích của việc phân mảnh
 - Tính cục bộ (Locality): Các mảnh dữ liệu có thể được đặt gần hơn với các ứng dụng hoặc người dùng thường xuyên truy cập chúng, giảm độ trễ và tăng tốc độ truy cập.
 - Thực thi đồng thời (Concurrent Execution): Các giao dịch có thể thực thi đồng thời trên các phần khác nhau của một quan hệ, tăng khả năng xử lý song song và hiệu suất tổng thể.
- Thách thức của việc phân mảnh
 - Giao tiếp thêm (Extra Communication): Việc phân mảnh có thể yêu cầu thêm giao tiếp giữa các nút để truy cập dữ liệu từ các mảnh khác nhau.
 - Xử lý thêm (Extra Processing): Các view không thể định nghĩa trên một mảnh duy nhất sẽ yêu cầu xử lý thêm để kết hợp dữ liệu từ nhiều mảnh.
 - Kiểm soát dữ liệu ngữ nghĩa (Semantic Data Control): Đảm bảo tính toàn vẹn dữ liệu (integrity enforcement) trở nên phức tạp hơn khi dữ liệu được phân mảnh và phân phối.
- Phân loại các phương pháp phân mảnh
 - **Phân mảnh ngang** (Horizontal Fragmentation): Chia quan hệ thành các tập hợp hàng dựa trên điều kiện nhất định.
 - **Phân mảnh dọc** (Vertical Fragmentation): Chia quan hệ thành các tập hợp cột dựa trên các thuộc tính cụ thể.

- **Phân mảnh hỗn hợp (Hybrid Fragmentation):** Kết hợp cả phân mảnh ngang và dọc để tối ưu hóa hiệu suất và quản lý dữ liệu.

2.2 Phân mảnh ngang (Horizontal Partitioning)

- Phân mảnh ngang là một kỹ thuật phân chia cơ sở dữ liệu, trong đó một quan hệ R được chia thành nhiều phần nhỏ (fragments) chứa một tập hợp các bộ dữ liệu (tuples) thỏa mãn một điều kiện nhất định.
- Mỗi phân mảnh fragment R_j là một tập con của R được tạo bằng cách áp dụng một mệnh đề chọn lọc (selection formula F_j).
- Các tiêu chí phân mảnh:
 - **Tính đầy đủ (Completeness):** Đảm bảo rằng không có dữ liệu nào bị mất trong quá trình phân mảnh. Mọi dữ liệu trong quan hệ gốc R phải xuất hiện trong ít nhất một mảnh R_i .
 - **Tính tái tạo (Reconstruction):** Đảm bảo rằng quan hệ gốc R có thể được tái tạo lại từ các mảnh R_i bằng cách sử dụng một toán tử quan hệ phù hợp (ví dụ: phép hợp trong trường hợp phân mảnh ngang).
 - **Tính rời rạc (Disjointness):** Đảm bảo rằng các mảnh là rời rạc, không có sự chồng chéo dữ liệu giữa các mảnh. Điều này giúp tránh lãng phí tài nguyên và đảm bảo tính nhất quán dữ liệu.

2.3 Range Partitioning

2.3.1 Khái niệm và cách thức hoạt động

Range Partitioning (phân mảnh theo khoảng) là một phương pháp phân mảnh ngang trong đó dữ liệu được phân chia dựa trên các khoảng giá trị liên tục của một thuộc tính được chọn làm khóa phân mảnh. Trong bài tập, thuộc tính Rating với miền giá trị $[0, 5]$ được sử dụng làm khóa phân mảnh.

- Cách thức hoạt động:
 - **Xác định miền giá trị:** Thuộc tính Rating có miền giá trị từ 0 đến 5 với các mức: $[0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]$
 - **Phân chia khoảng đồng đều:** Miền giá trị được chia thành N khoảng con có độ rộng bằng nhau, trong đó N là số phân mảnh yêu cầu
 - **Gán dữ liệu vào phân mảnh:** Mỗi bản ghi được đặt vào phân mảnh tương ứng với khoảng giá trị chứa giá trị Rating của nó

- **Đặt tên phân mảnh:** Các phân mảnh được đặt tên theo quy ước range_part0, range_part1, ..., range_part(N-1)

2.3.2 Thuật toán phân chia theo khoảng giá trị

Thuật toán phân chia Range Partitioning được thực hiện theo các bước sau:

Bước 1: Tính độ rộng khoảng

$$\text{range_width} = (\text{max_value} - \text{min_value}) / N$$

Với Rating: $\text{range_width} = (5 - 0) / N = 5/N$

Bước 2: Xác định ranh giới các khoảng

Partition i có khoảng giá trị:

$$[\text{min_value} + i * \text{range_width}, \text{min_value} + (i+1) * \text{range_width}]$$

Riêng partition 0 bao gồm cả giá trị min_value:

$$[\text{min_value}, \text{min_value} + \text{range_width}]$$

Bước 3: Áp dụng với các trường hợp cụ thể

- **N = 1:** Một phân mảnh chứa toàn bộ dữ liệu [0, 5]
- **N = 2:**
 - Partition 0: [0, 2.5]
 - Partition 1: (2.5, 5]
- **N = 3:**
 - Partition 0: [0, 1.67]
 - Partition 1: (1.67, 3.34]
 - Partition 2: (3.34, 5]
- **N = 5:**
 - Partition 0: [0, 1]
 - Partition 1: (1, 2]
 - Partition 2: (2, 3]
 - Partition 3: (3, 4]
 - Partition 4: (4, 5]

2.3.3 Ưu điểm: Truy vấn theo khoảng hiệu quả

Range Partitioning mang lại nhiều ưu điểm quan trọng, đặc biệt trong việc tối ưu hóa truy vấn:

1. Hiệu suất truy vấn theo khoảng cao

- Khi thực hiện truy vấn với điều kiện khoảng (WHERE rating BETWEEN 2.0 AND 3.5), hệ thống chỉ cần truy cập vào các phân mảnh có chứa dữ liệu trong khoảng đó
- Giảm đáng kể số lượng phân mảnh cần quét, từ đó cải thiện hiệu suất truy vấn

2. Tối ưu cho các phép so sánh

- Các truy vấn với điều kiện $>$, $<$, \geq , \leq trên thuộc tính Rating có thể được tối ưu hóa bằng cách loại bỏ các phân mảnh không liên quan
- Ví dụ: Truy vấn "WHERE rating \geq 4.0" chỉ cần truy cập partition cuối cùng

3. Hỗ trợ song song hóa hiệu quả

- Các truy vấn có thể được thực thi song song trên nhiều phân mảnh một cách hiệu quả
- Dễ dàng phân phối tải công việc dựa trên khoảng giá trị

4. Thuận tiện cho việc bảo trì

- Có thể thực hiện bảo trì, sao lưu hoặc khôi phục từng phân mảnh độc lập
- Dễ dàng thêm hoặc xóa dữ liệu theo khoảng giá trị cụ thể

5. Tối ưu cho các phép JOIN

- Khi thực hiện JOIN với các bảng khác có cùng

2.3.4 Nhược điểm: Có thể gây mất cân bằng dữ liệu

Mặc dù có nhiều ưu điểm, Range Partitioning cũng tồn tại một số nhược điểm đáng kể:

1. Phân bố dữ liệu không đồng đều

- Nếu dữ liệu không được phân bố đều trong miền giá trị, một số phân mảnh có thể chứa nhiều dữ liệu hơn các phân mảnh khác
- Ví dụ: Nếu đa số người dùng cho điểm rating từ 3-5, các partition chứa khoảng này sẽ có nhiều dữ liệu hơn

2. Hiện tượng "Hot Spot"

- Các phân mảnh chứa dữ liệu được truy cập thường xuyên sẽ trở thành điểm nghẽn (hot spot)
- Dẫn đến tình trạng một số server xử lý quá tải trong khi các server khác nhàn rỗi

3. Khó khăn trong việc mở rộng

- Khi cần thêm phân mảnh mới, phải tái phân bố lại toàn bộ dữ liệu
- Quá trình này tốn thời gian và tài nguyên, đặc biệt với dữ liệu lớn

4. Phụ thuộc vào sự hiểu biết về dữ liệu

- Cần phải hiểu rõ về phân bố dữ liệu để chọn khoảng phân mảnh hợp lý
- Nếu phân bố dữ liệu thay đổi theo thời gian, hiệu quả phân mảnh có thể giảm

5. Không hiệu quả cho truy vấn không theo khoảng

- Các truy vấn không sử dụng thuộc tính phân mảnh làm điều kiện sẽ phải quét toàn bộ các phân mảnh
- Ví dụ: Truy vấn theo UserID sẽ phải tìm kiếm trên tất cả các phân mảnh

Ví dụ minh họa mất cân bằng dữ liệu:

Giả sử với $N = 3$ phân mảnh và phân bố rating thực tế như sau:

- Partition 0 [0, 1.67]: 100,000 bản ghi (10%)
- Partition 1 (1.67, 3.34]: 300,000 bản ghi (30%)
- Partition 2 (3.34, 5]: 600,000 bản ghi (60%)

Partition 2 sẽ chứa gấp 6 lần dữ liệu so với Partition 0, gây ra tình trạng mất cân bằng tải nghiêm trọng.

2.4 Round Robin Partitioning

2.4.1 Khái niệm và cách thức hoạt động

Round Robin Partitioning (phân mảnh vòng tròn) là một phương pháp phân mảnh ngang trong đó dữ liệu được phân phối tuần tự và đồng đều vào các phân mảnh theo một chu kỳ vòng tròn. Khác với Range Partitioning dựa trên giá trị của thuộc tính, Round Robin Partitioning phân chia dữ liệu dựa trên thứ tự xuất hiện của các bản ghi.

Cách thức hoạt động của Round Robin Partitioning:

1. **Nguyên lý vòng tròn:** Dữ liệu được phân phối theo chu kỳ từ phân mảnh 0 đến phân mảnh N-1, sau đó quay lại phân mảnh 0 và lặp lại
2. **Phân phối tuần tự:** Bản ghi đầu tiên được đặt vào partition 0, bản ghi thứ hai vào partition 1, ..., bản ghi thứ (N+1) lại được đặt vào partition 0
3. **Không phụ thuộc vào giá trị:** Việc phân mảnh không dựa trên bất kỳ thuộc tính nào của dữ liệu, chỉ dựa trên vị trí của bản ghi trong tập dữ liệu
4. **Đặt tên phân mảnh:** Các phân mảnh được đặt tên theo quy ước `rrobin_part0`, `rrobin_part1`, ..., `rrobin_part(N-1)`

Ví dụ minh họa với N = 3:

Bản ghi 1 → `rrobin_part0`
Bản ghi 2 → `rrobin_part1`
Bản ghi 3 → `rrobin_part2`
Bản ghi 4 → `rrobin_part0`
Bản ghi 5 → `rrobin_part1`
Bản ghi 6 → `rrobin_part2`

...

2.4.2 Thuật toán phân chia tuần tự

Thuật toán Round Robin Partitioning được thực hiện thông qua các bước sau:

Bước 1: Khởi tạo phân mảnh: Tạo N bảng phân mảnh

Bước 2: Thuật toán phân phối cơ bản

- Duyệt qua từng bản ghi trong tập dữ liệu và phân phối chúng tuần tự vào các phân mảnh.
- Bắt đầu với phân mảnh đầu tiên (`index = 0`), sau khi chèn một bản ghi vào phân mảnh hiện tại, chuyển sang phân mảnh tiếp theo.
- Khi đã chèn vào phân mảnh cuối cùng, quay lại phân mảnh đầu tiên để tiếp tục chu kỳ.

Bước 3: Thuật toán sử dụng counter toàn cục

Sử dụng một bảng metadata để lưu trữ bộ đếm toàn cục nhằm theo dõi vị trí phân mảnh tiếp theo sẽ được chèn dữ liệu.

```

current_count = get_current_counter()
partition_index = current_count % N
increment_counter()

```

Bước 4: Thuật toán dựa trên số lượng bản ghi

Thay vì sử dụng bộ đếm riêng biệt, phương pháp này đếm tổng số bản ghi hiện có trong tất cả các phân mảnh. Chỉ số phân mảnh cho bản ghi mới được xác định bằng cách lấy tổng số bản ghi hiện tại chia lấy dư cho N.

```

total_records = count_all_records_in_partitions()

return total_records % N

```

Các phương pháp theo dõi thứ tự:

1. **Metadata Table:** Tạo bảng riêng để lưu trữ counter hiện tại
2. **Counting Existing Records:** Đếm tổng số bản ghi hiện có trong tất cả phân mảnh
3. **Sequential ID Assignment:** Sử dụng một trường ID tuần tự để xác định phân mảnh

Ví dụ chi tiết với dữ liệu MovieLens:

Giả sử có 12 bản ghi rating đầu tiên và $N = 3$:

| STT | UserID | MovieID | Rating | Phân mảnh đích |
|-----|--------|---------|--------|----------------|
| 1 | 1 | 122 | 5.0 | rrobin_part0 |
| 2 | 1 | 185 | 5.0 | rrobin_part1 |
| 3 | 1 | 231 | 5.0 | rrobin_part2 |
| 4 | 1 | 292 | 4.0 | rrobin_part0 |

| | | | | |
|----|---|-----|-----|--------------|
| 5 | 1 | 316 | 4.5 | rrobin_part1 |
| 6 | 1 | 329 | 4.0 | rrobin_part2 |
| 7 | 1 | 355 | 3.5 | rrobin_part0 |
| 8 | 1 | 356 | 3.0 | rrobin_part1 |
| 9 | 1 | 362 | 4.0 | rrobin_part2 |
| 10 | 1 | 364 | 2.5 | rrobin_part0 |
| 11 | 1 | 370 | 4.5 | rrobin_part1 |
| 12 | 1 | 377 | 3.0 | rrobin_part2 |

Kết quả: Mỗi phân mảnh chứa đúng 4 bản ghi, phân bố hoàn toàn đồng đều.

2.4.3 Ưu điểm: Phân bố dữ liệu đồng đều

Round Robin Partitioning mang lại những ưu điểm vượt trội về phân bố dữ liệu:

1. Phân bố hoàn toàn đồng đều

- Đảm bảo số lượng bản ghi trong các phân mảnh chênh lệch không quá 1
- Với M bản ghi và N phân mảnh: mỗi phân mảnh chứa $\lfloor M/N \rfloor$ hoặc $\lceil M/N \rceil$ bản ghi
- Ví dụ: 10 triệu bản ghi với 3 phân mảnh \rightarrow mỗi phân mảnh ~ 3.33 triệu bản ghi

2. Tối ưu sử dụng tài nguyên

- Không có phân mảnh nào bị quá tải hoặc nhàn rỗi
- Tận dụng tối đa dung lượng lưu trữ của tất cả các server
- Cân bằng tải xử lý một cách tự nhiên

3. Đơn giản trong triển khai

- Thuật toán đơn giản, dễ hiểu và dễ cài đặt
- Không cần phân tích phân bố dữ liệu trước khi phân mảnh
- Không phụ thuộc vào đặc tính của dữ liệu

4. Tính dự đoán cao

- Luôn biết chính xác phân mảnh nào sẽ nhận bản ghi tiếp theo
- Dễ dàng tính toán dung lượng cần thiết cho mỗi phân mảnh

5. Mở rộng linh hoạt

- Khi thêm phân mảnh mới, chỉ cần điều chỉnh thuật toán modulo
- Không cần tái phân bố toàn bộ dữ liệu cũ (trong một số trường hợp)

6. Hiệu suất ổn định

- Không có hiện tượng hot spot
- Mọi truy vấn quét toàn bộ đều có hiệu suất tương đương nhau

Ví dụ minh họa phân bố đồng đều:

Với 1 triệu bản ghi và các kịch bản phân mảnh khác nhau:

| Số phân mảnh | Phân bố dữ liệu | Độ lệch tối đa |
|--------------|-----------------------------|----------------|
| N = 2 | 500,000 - 500,000 | 0 |
| N = 3 | 333,334 - 333,333 - 333,333 | 1 |

| | | |
|-------|---------------------------------------|---|
| N = 4 | 250,000 - 250,000 - 250,000 - 250,000 | 0 |
| N = 5 | 200,000 x 5 | 0 |
| N = 7 | 142,858 - 142,857 x 6 | 1 |

2.4.4 Nhược điểm: Khó tối ưu cho truy vấn có điều kiện

Mặc dù có ưu điểm về phân bố đều, Round Robin Partitioning tồn tại những hạn chế đáng kể:

1. Không tối ưu cho truy vấn có điều kiện lọc

- Mọi truy vấn có điều kiện WHERE đều phải quét toàn bộ các phân mảnh
- Không thể loại bỏ phân mảnh không liên quan (partition elimination)
- Ví dụ: Truy vấn "WHERE rating >= 4.0" vẫn phải tìm kiếm trên cả 3 phân mảnh

2. Hiệu suất truy vấn kém

- Chi phí I/O cao do phải truy cập nhiều phân mảnh
- Thời gian phản hồi lâu hơn so với Range Partitioning
- Tăng tải mạng khi các phân mảnh nằm trên các server khác nhau

3. Khó khăn trong tối ưu JOIN

- Không thể thực hiện JOIN cục bộ hiệu quả
- Phải thực hiện cross-partition JOIN với chi phí cao
- Dữ liệu liên quan có thể nằm rải rác trên các phân mảnh khác nhau

4. Không hỗ trợ locality of reference

- Dữ liệu có mối quan hệ logic không được nhóm lại cùng nhau

- Ví dụ: Các rating của cùng một user có thể nằm trên các phân mảnh khác nhau
- Giảm hiệu quả cache và buffer pool

5. Phức tạp trong bảo trì

- Khó thực hiện backup/restore theo logic nghiệp vụ
- Không thể xóa dữ liệu theo nhóm logic một cách hiệu quả
- Khó phân tích dữ liệu theo các tiêu chí cụ thể

6. Không phù hợp cho OLAP

- Các truy vấn phân tích phức tạp phải xử lý dữ liệu từ nhiều phân mảnh
- Aggregation operations trở nên phức tạp và chậm
- Không tận dụng được các đặc tính của dữ liệu để tối ưu

Ví dụ so sánh hiệu suất truy vấn:

Truy vấn: "Tìm tất cả rating ≥ 4.0 "

Với Range Partitioning (N=5):

- Chỉ cần truy cập partition 3 và 4 (40% phân mảnh)
- Quét ~2 triệu bản ghi (giả sử dữ liệu phân bố đều)

Với Round Robin Partitioning (N=5):

- Phải truy cập tất cả 5 partition (100% phân mảnh)
- Quét ~10 triệu bản ghi (toàn bộ dữ liệu)
- Chi phí I/O cao gấp 2.5 lần

Kịch bản phù hợp và không phù hợp:

Phù hợp:

- Ứng dụng chủ yếu thực hiện full table scan
- Workload cần cân bằng tải đều
- Dữ liệu được truy cập ngẫu nhiên
- Yêu cầu phân bố storage đồng đều

Không phù hợp:

- Truy vấn thường xuyên có điều kiện lọc
- Cần tối ưu hiệu suất cho specific queries
- Ứng dụng OLAP với nhiều aggregation
- Yêu cầu data locality cao

CHƯƠNG 3. PHÂN TÍCH VÀ THIẾT KẾ

3.1 Phân tích dữ liệu đầu vào

3.1.1 Cấu trúc file ratings.dat

File ratings.dat chứa dữ liệu đánh giá phim từ trang web MovieLens với cấu trúc được chuẩn hóa theo định dạng cố định. Mỗi dòng trong file đại diện cho một đánh giá của một người dùng đối với một bộ phim.

Định dạng dữ liệu:

UserID::MovieID::Rating::Timestamp

Đặc điểm của file:

- **Separator:** Sử dụng "::" (hai dấu hai chấm) làm ký tự phân cách
- **Encoding:** UTF-8 hoặc ASCII
- **Line ending:** Mỗi bản ghi trên một dòng riêng biệt
- **File size:** ~256MB (cho dataset khoảng 10 triệu bản ghi)
- **No header:** File không có dòng tiêu đề

Ví dụ dữ liệu thực tế:

1::122::5::838985046

1::185::4.5::838983525

1::231::4::838983392

1::292::3.5::838983421

3.1.2 Schema bảng Ratings trong CSDL

Dữ liệu từ file ratings.dat được chuyển đổi và lưu trữ trong bảng Ratings với schema được thiết kế tối ưu cho việc phân mảnh:

```
CREATE TABLE Ratings (  
    UserID INT NOT NULL,  
    MovieID INT NOT NULL,  
    Rating FLOAT NOT NULL,  
    PRIMARY KEY (UserID , MovieID)  
);
```

Index được đề xuất:

-- Index cho truy vấn theo Rating (thuộc tính phân mảnh)

```
CREATE INDEX CONCURRENTLY idx_Ratings_rating ON  
Ratings(Rating);
```

-- Index cho truy vấn theo UserID

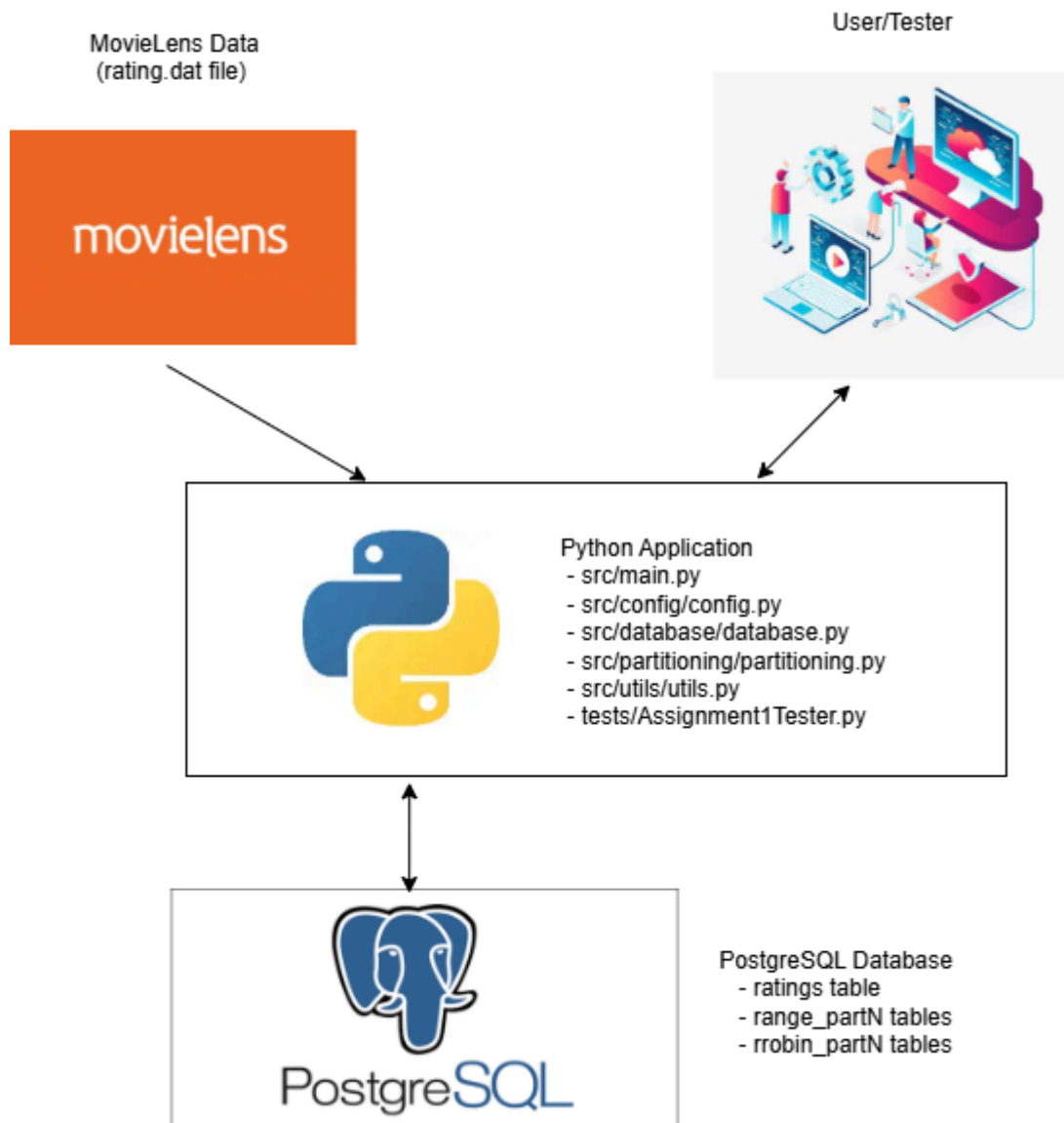
```
CREATE INDEX CONCURRENTLY idx_Ratings_userid ON  
Ratings(UserID);
```

-- Index cho truy vấn theo MovieID

```
CREATE INDEX CONCURRENTLY idx_Ratings_movieid ON  
Ratings(MovieID);
```

3.2 Thiết kế kiến trúc hệ thống

3.2.1 Sơ đồ tổng quan hệ thống

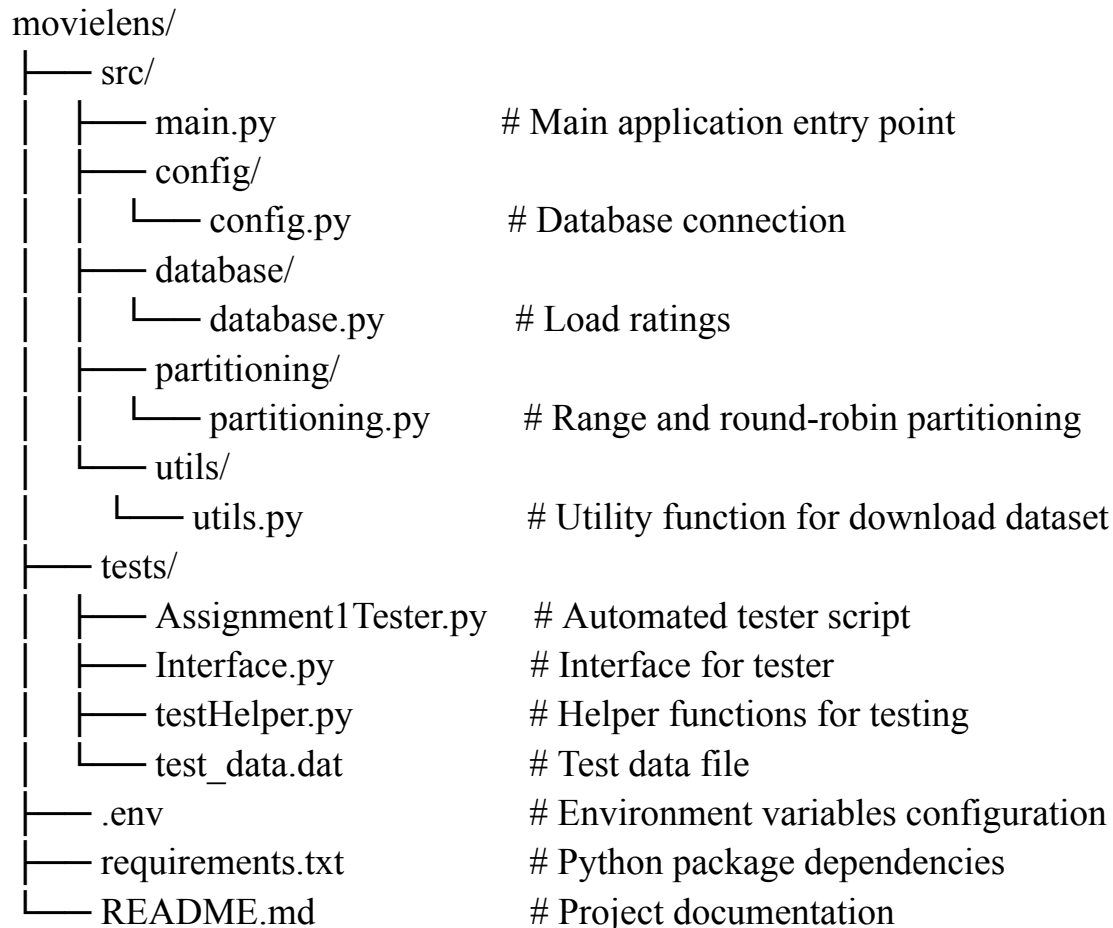


- **MovieLens Data:** Dữ liệu gốc (ratings) được nạp vào hệ thống.
- **Python Application:** Xử lý logic partitioning, insert, và giao tiếp với PostgreSQL.
- **PostgreSQL Database:** Lưu trữ dữ liệu gốc và các bảng partition (range, round-robin).
- **User/Tester:** Tương tác với ứng dụng để kiểm thử và sử dụng các chức năng partition/insert.

3.2.2 Mô hình kết nối CSDL

Ứng dụng đọc cấu hình từ .env → tạo kết nối qua psycopg2 → truyền kết nối này cho các hàm thao tác dữ liệu → đóng kết nối khi xong.

3.2.3 Cấu trúc thư mục và module Python



3.3 Thiết kế các hàm chính

3.3.1 Hàm LoadRatings()

- Input: Đường dẫn file ratings.dat
- Output: Tải dữ liệu vào bảng Ratings
- Xử lý format dữ liệu, tạo các index

3.3.2 Hàm Range_Partition()

- Input: Bảng Ratings, số phân mảnh N
- Output: N bảng phân mảnh theo khoảng Rating

- Thuật toán chia khoảng đồng đều

3.3.3 Hàm RoundRobin_Partition()

- Input: Bảng Ratings, số phân mảnh N
- Output: N bảng phân mảnh theo round robin
- Thuật toán phân chia tuần tự

3.3.4 Hàm Range_Insert() và RoundRobin_Insert()

- Input: Thông tin bản ghi mới
- Output: Chèn vào đúng phân mảnh
- Logic xác định phân mảnh đích

CHƯƠNG 4. TRIỂN KHAI VÀ CÀI ĐẶT

4.1 Môi trường phát triển

- Cấu hình máy ảo Ubuntu 16.04
- Cài đặt PostgreSQL 17.5
- Cài đặt Python 3.12.3 và các thư viện cần thiết:
 - psycopg2
 - python-dotenv
 - requests

4.2 Cài đặt các hàm chức năng

4.2.1. Xử lý kết nối database

Được xử lý trong file config.py (thư mục src/config)

```
class DatabaseConfig:
    @classmethod
    def get_connection_params(cls):
        params = {
            'host': os.getenv('DB_HOST'),
            'database': os.getenv('DB_NAME'),
            'user': os.getenv('DB_USER'),
            'password': os.getenv('DB_PASSWORD'),
            'port': os.getenv('DB_PORT')
        }
        print("[DatabaseConfig] Loaded params:", params)
        return params

    @classmethod
    def get_instance(cls):
        """Singleton pattern to get DatabaseConfig instance"""
        if not hasattr(cls, '_instance'):
            cls._instance = cls()
        return cls._instance
```

1. Định nghĩa class DatabaseConfig

- Class DatabaseConfig chịu trách nhiệm quản lý thông tin cấu hình kết nối đến database.
- Sử dụng biến môi trường để lấy các thông tin cần thiết như host, database, user, password, port.

2. Lấy thông tin kết nối

- Phương thức get_connection_params (class method):

- Đọc các biến môi trường: DB_HOST, DB_NAME, DB_USER, DB_PASSWORD, DB_PORT.
- Trả về một dictionary chứa các tham số này.
- Có in ra thông tin các tham số đã load để kiểm tra.

3. Singleton pattern cho cấu hình

- Phương thức get_instance đảm bảo chỉ tạo một instance duy nhất của DatabaseConfig trong suốt vòng đời chương trình (singleton pattern).

4. Quy trình kết nối database tổng thể (liên quan đến các file khác)

- Ở file main.py, hàm get_connection() (import từ database/database.py) sẽ sử dụng các tham số từ DatabaseConfig.get_connection_params() để tạo kết nối đến database (thường là qua thư viện psycopg2 cho PostgreSQL).
- Kết nối này được truyền vào các hàm thao tác dữ liệu như loadratings, rangepartition, roundrobinpartition, rangeinsert, roundrobininsert.
- Sau khi hoàn thành các thao tác, kết nối sẽ được đóng lại trong khối finally để đảm bảo không bị rò rỉ tài nguyên.

5. Ví dụ sử dụng

```
from config.config import DatabaseConfig

def get_connection():
    """Create connection to PostgreSQL database"""
    try:
        conn = psycopg2.connect(**DatabaseConfig.get_connection_params())
        print("Database connection established.")
        return conn
    except psycopg2.Error as e:
        print(f"Database connection error: {e}")
        raise
```

4.2.2 Hàm tải xuống dữ liệu đầu vào

Hàm download_movielens_dataset() nằm trong file utils.py (ở thư mục utils), có nhiệm vụ tải về và giải nén bộ dữ liệu MovieLens (khoảng 10 triệu dòng), đồng thời trả về đường dẫn đến file ratings.dat thực hiện theo các bước sau:

1. Định nghĩa các đường dẫn và URL:


```
dataset_url = "http://files.grouplens.org/datasets/movielens/ml-10m.zip"
zip_path = os.path.join("data", "ml-10m.zip")
extract_path = os.path.join("data", "ml-10M100K")
```

- dataset_url: URL nơi chứa file zip của bộ dữ liệu MovieLens.
- zip_path: Đường dẫn lưu file zip sau khi tải về (trong thư mục data).
- extract_path: Đường dẫn thư mục sau khi giải nén (data/ml-10M100K).

2. Tạo thư mục data nếu chưa tồn tại:

```
# Create data directory if it doesn't exist
os.makedirs("data", exist_ok=True)
```

- Sử dụng os.makedirs("data", exist_ok=True) để đảm bảo thư mục data luôn tồn tại.

3. Tải file zip nếu chưa có:

```
# Download the dataset if it doesn't exist
if not os.path.exists(zip_path):
    print("Downloading MovieLens dataset...")
    response = requests.get(dataset_url, stream=True)
    response.raise_for_status()

    with open(zip_path, 'wb') as f:
        for chunk in response.iter_content(chunk_size=8192):
            f.write(chunk)
    print("Download completed.")
```

- Kiểm tra nếu zip_path chưa tồn tại thì thực hiện tải về.
- Sử dụng requests.get với stream=True để tải file lớn theo từng phần nhỏ (chunk).
- Ghi từng chunk vào file zip_path.
- In ra thông báo khi bắt đầu và hoàn thành tải.

4. Giải nén file zip nếu chưa giải nén:

```
# Extract the dataset if it hasn't been extracted
if not os.path.exists(extract_path):
    print("Extracting dataset...")
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        # Extract only the ml-10M100K directory
        for file in zip_ref.namelist():
            if file.startswith('ml-10M100K/'):
                zip_ref.extract(file, "data")
    print("Extraction completed.")
```

- Kiểm tra nếu extract_path chưa tồn tại thì thực hiện giải nén.
- Mở file zip bằng zipfile.ZipFile.
- Duyệt qua các file trong zip, chỉ giải nén các file nằm trong thư mục ml-10M100K/.
- In ra thông báo khi bắt đầu và hoàn thành giải nén.

5. Trả về đường dẫn đến file ratings.dat:

```
# Return path to ratings.dat file
return os.path.join(extract_path, "ratings.dat")
```

- Kết thúc hàm, trả về đường dẫn đến file ratings.dat nằm trong thư mục đã giải nén.

Kết quả:

```
shadowz@ubuntu:~/csdlpt/src$ python3.12 main.py
[DatabaseConfig] Loaded params: {'host': 'localhost', 'database': 'dds_assgn1',
'user': 'postgres', 'password': '1234', 'port': '5432'}
Database connection established.
Downloading MovieLens dataset...
Download completed.
Extracting dataset...
Extraction completed.
```

4.2.3 Hàm LoadRatings()

Hàm loadratings trong file database.py (thư mục database) có nhiệm vụ tải dữ liệu ratings (đánh giá phim) từ một file ratings.dat vào bảng trong cơ sở dữ liệu PostgreSQL một cách tối ưu

1. Khởi tạo và tối ưu hóa PostgreSQL

```

# Optimize PostgreSQL settings for bulk insert - only runtime changeable parameters
optimization_settings = [
    "SET maintenance_work_mem = '512MB'",
    "SET work_mem = '128MB'",
    "SET synchronous_commit = OFF",
    "SET commit_delay = 0",
    "SET commit_siblings = 5",
    "SET temp_buffers = '32MB'",
    "SET effective_cache_size = '512MB'"
]

```

Mục đích: Tối ưu hóa cấu hình PostgreSQL cho việc chèn dữ liệu hàng loạt:

- **maintenance_work_mem:** Tăng bộ nhớ cho các thao tác bảo trì (tạo index)
- **work_mem:** Tăng bộ nhớ cho các thao tác sắp xếp/hash
- **synchronous_commit = OFF:** Tắt đồng bộ hóa để tăng tốc (trade-off với độ an toàn)
- **effective_cache_size:** Thông báo cho PostgreSQL về bộ nhớ cache có sẵn

2. Tạo bảng tối ưu

```

cursor.execute(f"""
    CREATE UNLOGGED TABLE {ratingtablename} (
        userid INT NOT NULL,
        movieid INT NOT NULL,
        rating FLOAT NOT NULL,
        PRIMARY KEY (userid, movieid)
    ) WITH (fillfactor = 90)
""")

```

Đặc điểm tối ưu:

- **UNLOGGED:** Không ghi WAL log, tăng tốc độ chèn đáng kể
- **fillfactor = 90:** Để lại 10% không gian trống trong mỗi page cho updates tương lai
- Composite primary key để tránh duplicate

3. Lựa chọn phương pháp tải dữ liệu

Hàm sử dụng 3 phương pháp tùy theo kích thước file:

Phương pháp 1: COPY Command (file > 50MB)

```
# Method 1: Use COPY command (fastest for large datasets)
if file_size > 50 * 1024 * 1024: # Files larger than 50MB
    if use_copy_method(ratingtablename, ratingsfilepath, openconnection):
```

Ưu điểm: Nhanh nhất cho dữ liệu lớn, sử dụng lệnh COPY native của PostgreSQL.

Phương pháp 2: Parallel Batch Insert

```
else:
    # Method 2: Fallback to parallel batch insert
    load_with_parallel_insert(ratingtablename, ratingsfilepath, openconnection)
    end_time = time.time()
    print(f"Data loaded successfully using parallel batch insert in {end_time - start_time:.2f} seconds")
```

Sử dụng khi COPY method thất bại:

- Chia dữ liệu thành chunks
- Xử lý song song với multiple threads
- Mỗi thread có connection riêng

Phương pháp 3: Optimized Batch Insert (file nhỏ)

```
else:
    # Method 3: Optimized batch insert for smaller files
    load_with_batch_insert(ratingtablename, ratingsfilepath, openconnection)
    end_time = time.time()
    print(f"Data loaded successfully using batch insert in {end_time - start_time:.2f} seconds")
```

Sử dụng `execute_values` với batch size được tối ưu.

Chi tiết các phương pháp

COPY Method (use_copy_method)

```
def data_generator():
    with open(ratingsfilepath, 'r', encoding='utf-8', buffering=8192*8) as infile:
        line_count = 0
        for line in infile:
            line_count += 1
            if line_count % 1000000 == 0:
                print(f"Processed {line_count:,} lines...")

            try:
                parts = line.strip().split(':')
                if len(parts) >= 3:
                    userid, movieid, rating = parts[0], parts[1], parts[2]
                    yield f"{userid},{movieid},{rating}\n"
            except (ValueError, IndexError):
                continue # Skip malformed lines
```

Đặc điểm:

- Streaming data processing (không load toàn bộ file vào memory)
- Sử dụng generator để tiết kiệm memory
- Buffer size được tối ưu (8192*8 bytes)

Parallel Insert Method

```
# Process chunks in parallel
def process_chunk(chunk_data):
    try:
        # Create a new connection for this thread
        conn_params = openconnection.get_dsn_parameters()
        thread_conn = psycopg2.connect(**conn_params)
        thread_cursor = thread_conn.cursor()
```

Đặc điểm:

- Tối đa 4 threads để không quá tải database
- Mỗi thread có connection riêng
- Batch size = 100,000 records
- Sử dụng **ON CONFLICT** để handle duplicates

4. Tối ưu hóa sau khi tải dữ liệu

Chuyển đổi table type

```
# Convert UNLOGGED table to LOGGED after data loading
cursor.execute(f"ALTER TABLE {ratingstablename} SET LOGGED")
```

Chuyển từ UNLOGGED sang LOGGED để đảm bảo durability.

Tạo indexes

```
cursor = openconnection.cursor()
indexes = [
    f"CREATE INDEX CONCURRENTLY idx_{ratingstablename}_userid ON {ratingstablename}(userid)",
    f"CREATE INDEX CONCURRENTLY idx_{ratingstablename}_movieid ON {ratingstablename}(movieid)",
    f"CREATE INDEX CONCURRENTLY idx_{ratingstablename}_rating ON {ratingstablename}(rating)"
]
```

Đặc điểm:

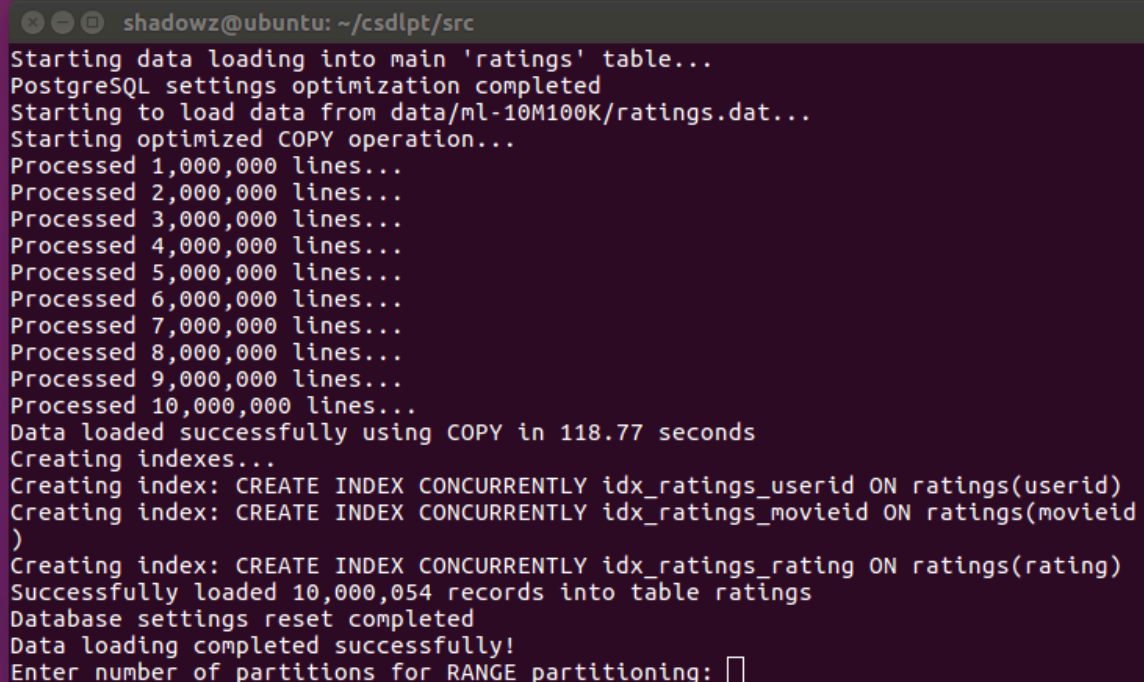
- Sử dụng **CONCURRENTLY** để không block các operations khác
- Có fallback mechanism nếu concurrent creation thất bại
- Tạo indexes sau khi load data (hiệu quả hơn)

Analyze table

```
cursor.execute(f"ANALYZE {ratingstablename}")
```

Cập nhật statistics để query planner hoạt động tốt hơn.

Kết quả



```
shadowz@ubuntu: ~/csdplt/src
Starting data loading into main 'ratings' table...
PostgreSQL settings optimization completed
Starting to load data from data/ml-10M100K/ratings.dat...
Starting optimized COPY operation...
Processed 1,000,000 lines...
Processed 2,000,000 lines...
Processed 3,000,000 lines...
Processed 4,000,000 lines...
Processed 5,000,000 lines...
Processed 6,000,000 lines...
Processed 7,000,000 lines...
Processed 8,000,000 lines...
Processed 9,000,000 lines...
Processed 10,000,000 lines...
Data loaded successfully using COPY in 118.77 seconds
Creating indexes...
Creating index: CREATE INDEX CONCURRENTLY idx_ratings_userid ON ratings(userid)
Creating index: CREATE INDEX CONCURRENTLY idx_ratings_movieid ON ratings(movieid)
Creating index: CREATE INDEX CONCURRENTLY idx_ratings_rating ON ratings(rating)
Successfully loaded 10,000,054 records into table ratings
Database settings reset completed
Data loading completed successfully!
Enter number of partitions for RANGE partitioning: 
```

4.2.4 Hàm Range_Partition()

- Thực hiện phân vùng dữ liệu theo khoảng giá trị (range partitioning) cho bảng ratings.
- Hàm **rangepartition()** chia dữ liệu từ bảng ratings gốc thành nhiều bảng con (partitions) dựa trên phạm vi giá trị của cột **rating**.

1. Khởi tạo

```
def rangepartition(ratingtablename, numberofpartitions, openconnection):  
    """
```

Parameters:

- **ratingtablename**: Tên bảng ratings gốc
- **numberofpartitions**: Số lượng partition muốn tạo
- **openconnection**: Connection đến PostgreSQL database

2. Xóa các partition cũ (nếu có)

```
# Drop old partitions safely  
for i in range(numberofpartitions):  
    partition_name = f"{RANGE_TABLE_PREFIX}{i}"  
    try:  
        cursor.execute(f"DROP TABLE IF EXISTS {partition_name}")  
    except Exception as e:  
        print(f"Warning: Could not drop table {partition_name}: {e}")  
        openconnection.rollback()
```

Mục đích:

- Dọn dẹp các bảng partition cũ trước khi tạo mới
- RANGE_TABLE_PREFIX = 'range_part' → tạo các bảng có tên: range_part0, range_part1, ...
- Error handling để tránh crash nếu bảng không tồn tại

3. Tính toán phạm vi phân vùng

```
# Get min and max rating
cursor.execute(f"SELECT MIN(rating), MAX(rating) FROM {ratingtablename}")
min_rating, max_rating = cursor.fetchone()

# Calculate partition size
range_size = (max_rating - min_rating) / numberofpartitions
```

Cách hoạt động:

- Tìm giá trị rating nhỏ nhất và lớn nhất trong bảng
- Chia đều khoảng giá trị thành **numberofpartitions** phần
- Ví dụ: rating từ 1.0 đến 5.0, chia 4 partitions → mỗi partition có range = 1.0

4. Tạo các bảng partition

```
for i in range (numberofpartitions):
    partition_name = f"{RANGE_TABLE_PREFIX}{i}"
    cursor.execute(f"DROP TABLE IF EXISTS {partition_name} CASCADE;")

for i in range(numberofpartitions):
    partition_name = f"{RANGE_TABLE_PREFIX}{i}"
    lower_bound = min_rating + i * range_size
    upper_bound = min_rating + (i + 1) * range_size

    # Handle last partition with inclusive upper bound
    if i == numberofpartitions - 1:
        condition = f"rating >= {lower_bound} AND rating <= {upper_bound}"
    else:
        condition = f"rating >= {lower_bound} AND rating < {upper_bound}"

    cursor.execute(f"""
        CREATE TABLE IF NOT EXISTS {partition_name} (
            userid INT,
            movieid INT,
            rating FLOAT
        )
    """)
```

Logic phân vùng:

- Partition 0: rating >= min_rating AND rating < min_rating + range_size
- Partition 1: rating >= min_rating + range_size AND rating < min_rating + 2*range_size ...

- Partition cuối: rating \geq lower_bound AND rating \leq max_rating (inclusive cho boundary cuối)

Ví dụ cụ thể: Nếu có rating từ 1.0 đến 5.0, chia 4 partitions:

- range_part0: rating \geq 1.0 AND rating $<$ 2.0
- range_part1: rating \geq 2.0 AND rating $<$ 3.0
- range_part2: rating \geq 3.0 AND rating $<$ 4.0
- range_part3: rating \geq 4.0 AND rating \leq 5.0

5. Chèn dữ liệu vào các partition

```
cursor.execute(f"""
    INSERT INTO {partition_name}
    SELECT userid, movieid, rating FROM {ratingtablename}
    WHERE {condition}
""")
```

Hoạt động:

- Copy dữ liệu từ bảng gốc vào từng partition dựa theo điều kiện range
- Mỗi partition chỉ chứa dữ liệu trong phạm vi rating tương ứng

6. Commit và hoàn tất

```
openconnection.commit()
print(f"Created {numberofpartitions} range partitions")
print(f"--- Finished RANGE partitioning ---\n")
```

4.2.5 Hàm RoundRobin_Partition()

Hàm roundrobinpartition chia đều dữ liệu từ bảng ratings gốc vào N bảng con theo cách tuần tự (round-robin), đảm bảo mỗi partition có số lượng record gần như bằng nhau.

1. Khởi tạo và validation

```
def roundrobinpartition(ratingtablename: str, N: int, open_connection):
    print(f"\n--- Starting ROUND ROBIN partitioning with {N} partitions ---")
    start_time = time.time()

    if not isinstance(N, int) or N <= 0:
        print(f"Error: Number of partitions N ({N}) must be a positive integer (N >= 1).")
        return
```

Parameters:

- **ratingtablename**: Tên bảng ratings gốc
- **N**: Số lượng partitions muốn tạo
- **open_connection**: Connection đến PostgreSQL database

Validation: Kiểm tra N phải là số nguyên dương

2. Cleanup các bảng cũ

```
# Drop old Round Robin partition tables and metadata table (if they exist)
for i in range(N):
    partition_name = f"{RROBIN_TABLE_PREFIX}{i}"
    cursor.execute(f"DROP TABLE IF EXISTS {partition_name};")
cursor.execute("DROP TABLE IF EXISTS rrobin_metadata;")
open_connection.commit()
```

Mục đích:

- Xóa các partition cũ (rrobin_part0, rrobin_part1, ...)
- Xóa bảng metadata cũ để tạo mới
- RROBIN_TABLE_PREFIX = 'rrobin_part'

3. Tạo bảng metadata

```
# Create metadata table to store insertion index and number of partitions
cursor.execute("""
    CREATE TABLE rrobin_metadata (
        id SERIAL PRIMARY KEY,
        current_insert_index BIGINT NOT NULL DEFAULT 0,
        num_partitions INT NOT NULL
    );
""")
cursor.execute("INSERT INTO rrobin_metadata (num_partitions) VALUES (%s);", (N,))
```

Chức năng metadata table:

- **current_insert_index**: Theo dõi vị trí insert tiếp theo (cho single inserts sau này)
- **num_partitions**: Lưu số lượng partitions
- Được sử dụng bởi hàm **roundrobininsert** để biết insert vào partition nào

4. Tạo các bảng partition

```
# Create N child tables (partitions) with schema similar to Ratings
for i in range(N):
    partition_name = f"{RROBIN_TABLE_PREFIX}{i}"
    cursor.execute(f"""
        CREATE TABLE {partition_name} (
            UserID INT,
            MovieID INT,
            Rating FLOAT,
            PRIMARY KEY (UserID, MovieID, Rating)
        );
    """)
```

Đặc điểm:

- Tạo N bảng con có cấu trúc giống bảng gốc
- Composite primary key để tránh duplicate
- Tên bảng: rrobin_part0, rrobin_part1, ..., rrobin_partN-1

5. Phân phối dữ liệu theo Round Robin

```
# Insert data into partitions using SQL directly (optimized)
total_records_processed = 0
for i in range(N):
    partition_name = f"{RROBIN_TABLE_PREFIX}{i}"
    cursor.execute(f"""
        INSERT INTO {partition_name} ({USER_ID_COLNAME}, {MOVIE_ID_COLNAME}, {RATING_COLNAME})
        SELECT UserID, MovieID, Rating
        FROM (
            SELECT UserID, MovieID, Rating,
                   ROW_NUMBER() OVER (ORDER BY UserID, MovieID, Rating) as rn
            FROM {ratingtablename}
        ) AS numbered_ratings
        WHERE (rn - 1) % {N} = {i};
    """)
    rows_inserted = cursor.rowcount
    total_records_processed += rows_inserted
```

Cách hoạt động:

Bước 1: Đánh số thứ tự

```
ROW_NUMBER() OVER (ORDER BY UserID, MovieID, Rating) as rn
```

- Gán số thứ tự cho mỗi record (1, 2, 3, ...)
- Sắp xếp theo UserID, MovieID, Rating để đảm bảo tính nhất quán

Bước 2: Phân chia Round Robin

```
WHERE (rn - 1) % {N} = {i};
```

- **Partition 0:** $(rn-1) \% N = 0 \rightarrow$ records có $rn = 1, N+1, 2N+1, \dots$
- **Partition 1:** $(rn-1) \% N = 1 \rightarrow$ records có $rn = 2, N+2, 2N+2, \dots$
- **Partition i:** $(rn-1) \% N = i \rightarrow$ records có $rn = i+1, N+i+1, 2N+i+1, \dots$

Ví dụ với $N=3$:

- Record 1 \rightarrow Partition 0 $(1-1) \% 3 = 0$
- Record 2 \rightarrow Partition 1 $(2-1) \% 3 = 1$
- Record 3 \rightarrow Partition 2 $(3-1) \% 3 = 2$
- Record 4 \rightarrow Partition 0 $(4-1) \% 3 = 0$
- Record 5 \rightarrow Partition 1 $(5-1) \% 3 = 1$
- ...

6. Cập nhật metadata và hoàn tất

```
# Update the final insertion index in the metadata table
cursor.execute("UPDATE rrobin_metadata SET current_insert_index = %s WHERE id = 1;", (total_records_processed,))

# Reset current_insert_index to 0 for subsequent single inserts by tester
cursor.execute("UPDATE rrobin_metadata SET current_insert_index = 0 WHERE id = 1;")

open_connection.commit()
```

Mục đích:

- Lưu tổng số records đã xử lý
- Reset index về 0 để chuẩn bị cho các single inserts tiếp theo
- Đảm bảo hàm **roundrobininsert** hoạt động đúng

4.2.6 Hàm Range_Insert()

Hàm **rangeinsert** xác định partition đúng để chèn một record mới dựa trên giá trị rating, sau đó thực hiện insert vào partition tương ứng.

1. Khởi tạo

```
def rangeinsert(ratingtablename, userid, movieid, rating, openconnection):
    """
    Insert a new rating using range partitioning.

    Args:
        ratingtablename: Name of the ratings table
        userid: User ID
        movieid: Movie ID
        rating: Rating value
        openconnection: Database connection
    """
    cursor = openconnection.cursor()
```

Parameters:

- **ratingtablename**: Tên bảng gốc (để lấy min/max rating)
- **userid**: ID của user
- **movieid**: ID của movie
- **rating**: Giá trị rating (quyết định partition nào)
- **openconnection**: Database connection

2. Lấy thông tin phạm vi rating

```
# Get min and max ratings
cursor.execute(f"SELECT MIN(rating), MAX(rating) FROM {ratingtablename}")
min_rating, max_rating = cursor.fetchone()
```

Mục đích:

- Cần min/max rating để có thể tính toán partition boundaries
- Sử dụng cùng logic như khi tạo partitions ban đầu
- Đảm bảo consistency với range partitioning scheme

3. Xác định số lượng partitions

```
# Get number of partitions from existing range partition tables
cursor.execute(f"""
    SELECT COUNT(*) FROM information_schema.tables
    WHERE table_name LIKE '{RANGE_TABLE_PREFIX}%'
""")
numberofpartitions = cursor.fetchone()[0]

if numberofpartitions == 0:
    raise Exception("No range partitions found. Please run rangepartition first.")
```

Cách hoạt động:

- Query `information_schema.tables` để đếm các bảng có tên bắt đầu với `range_part`
- `RANGE_TABLE_PREFIX = 'range_part'`
- Validation: Đảm bảo partitions đã được tạo trước khi insert

4. Tính toán partition đích

```
# Calculate partition number
range_size = (max_rating - min_rating) / numberofpartitions
partition_num = int((rating - min_rating) / range_size)
partition_name = f"{RANGE_TABLE_PREFIX}{partition_num}"
```

Logic tính toán:

Bước 1: Tính kích thước mỗi partition

$$\text{range_size} = (\text{max_rating} - \text{min_rating}) / \text{numberofpartitions}$$

Bước 2: Xác định partition index

$$\text{partition_num} = \text{int}((\text{rating} - \text{min_rating}) / \text{range_size})$$

Ví dụ cụ thể:

- Giả sử: `min_rating = 1.0`, `max_rating = 5.0`, `numberofpartitions = 4`
- `range_size = (5.0 - 1.0) / 4 = 1.0`
- Partition boundaries:
 - Partition 0: [1.0, 2.0)
 - Partition 1: [2.0, 3.0)
 - Partition 2: [3.0, 4.0)
 - Partition 3: [4.0, 5.0]

Tính toán với `rating = 3.5`:

$$\text{partition_num} = \text{int}((3.5 - 1.0) / 1.0) = \text{int}(2.5) = 2$$
$$\text{partition_name} = \text{"range_part2"}$$

5. Xác minh partition tồn tại

```

# Check if partition exists
cursor.execute(f"""
    SELECT EXISTS (
        SELECT FROM information_schema.tables
        WHERE table_name = '{partition_name}'
    )
""")
if not cursor.fetchone()[0]:
    raise Exception(f"Partition {partition_name} does not exist")

```

Mục đích:

- Double-check partition table thực sự tồn tại
- Tránh SQL error khi insert vào bảng không tồn tại
- Safety check để đảm bảo data integrity

6. Thực hiện insert

```

# Insert into appropriate partition
cursor.execute(
    f"INSERT INTO {partition_name} (userid,movieid,rating) VALUES (%s, %s, %s)", (userid, movieid, rating)
)

openconnection.commit()
print(f"Inserted rating into range partition {partition_num}")

```

Đặc điểm:

- Sử dụng parameterized query để tránh SQL injection
- Commit ngay sau insert
- Log thông tin để debug/monitoring

7. Exception handling

```

except Exception as e:
    openconnection.rollback()
    print(f"Error inserting rating: {e}")
    raise
finally:
    cursor.close()

```

Error handling:

- Rollback transaction nếu có lỗi

- Log error message
- Đảm bảo cursor được đóng (resource cleanup)

4.2.7 Hàm RoundRobin_Insert()

Hàm roundrobininsert() dùng để chèn một bản ghi mới vào đúng partition (phân vùng) trong hệ thống phân vùng Round Robin.

1. Khởi tạo và lấy metadata

```
def roundrobininsert(ratingtablename, UserID: int, MovieID: int, Rating: float, openconnection):
```

Parameters:

- **ratingtablename**: Tên bảng ratings gốc (không được sử dụng trực tiếp trong hàm)
- **UserID**: ID của người dùng
- **MovieID**: ID của phim
- **Rating**: Điểm đánh giá
- **openconnection**: Kết nối database PostgreSQL

```
# Get current insertion index and number of partitions from metadata table
cursor.execute("SELECT current_insert_index, num_partitions FROM rrobin_metadata WHERE id = 1 FOR UPDATE;")
metadata = cursor.fetchone()
```

- Truy vấn bảng **rrobin_metadata** để lấy:
 - **current_insert_index**: Chỉ số chèn hiện tại (đếm tổng số bản ghi đã chèn)
 - **num_partitions**: Số lượng partition (N)
- Sử dụng **FOR UPDATE** để khóa hàng, tránh race condition khi có nhiều thao tác chèn đồng thời

2. Xác định partition đích

```
# Determine the target partition table name
partition_index = current_insert_index % N
target_table = f"{RROBIN_TABLE_PREFIX}{partition_index}"
```

- Sử dụng phép chia lấy dư (%) để xác định partition nào sẽ nhận bản ghi mới
- Ví dụ: Nếu có 3 partition (N=3) và current_insert_index = 7:
 - partition_index = 7 % 3 = 1

- `target_table = "rrobin_part1"`

3. Chèn dữ liệu

```
# Insert the new record into the target partition table
cursor.execute(f"""
    INSERT INTO {target_table} (UserID, MovieID, Rating)
    VALUES (%s, %s, %s);
""", (UserID, MovieID, Rating))
```

- Chèn bản ghi vào partition được xác định

4. Cập nhật metadata

```
# Update insertion index in the metadata table
new_insert_index = current_insert_index + 1
cursor.execute("UPDATE rrobin_metadata SET current_insert_index = %s WHERE id = 1;", (new_insert_index,))
```

- Tăng **current_insert_index** lên 1 để chuẩn bị cho lần chèn tiếp theo
- Đảm bảo bản ghi tiếp theo sẽ được chèn vào partition kế tiếp theo thứ tự Round Robin

Ví dụ minh họa

Giả sử có 3 partition (rrobin_part0, rrobin_part1, rrobin_part2):

| Lần chèn | current_insert_index | partition_index | Target partition |
|----------|----------------------|-----------------|------------------|
| 1 | 0 | $0 \% 3 = 0$ | rrobin_part0 |
| 2 | 1 | $1 \% 3 = 1$ | rrobin_part1 |
| 3 | 2 | $2 \% 3 = 2$ | rrobin_part2 |
| 4 | 3 | $3 \% 3 = 0$ | rrobin_part0 |
| 5 | 4 | $4 \% 3 = 1$ | rrobin_part1 |

CHƯƠNG 5. KIỂM THỬ VÀ ĐÁNH GIÁ

5.1 Kết quả kiểm thử

5.1.1 Test với dữ liệu mẫu nhỏ (file test_data.dat)

- Bảng kết quả test các hàm

| Hàm | Kết quả |
|------------------------|---------|
| LoadRatings() | pass |
| Range_Partition() | pass |
| Range_Insert() | pass |
| RoundRobin_Partition() | pass |
| RoundRobin_Insert() | pass |

- Thống kê phân bố dữ liệu sau phân mảnh
 - Range_Partition():

Số phân mảnh:

```
dds_assgn1=# \dt range_part*
              List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | range_part0    | table | postgres
 public | range_part1    | table | postgres
 public | range_part2    | table | postgres
 public | range_part3    | table | postgres
 public | range_part4    | table | postgres
(5 rows)
```

Tổng số lượng bản ghi:

```

dds_assgn1=# SELECT SUM(total_records) AS total_records_in_range_partitions
FROM (
    SELECT COUNT(*) AS total_records FROM range_part0
    UNION ALL
    SELECT COUNT(*) AS total_records FROM range_part1
    UNION ALL
    SELECT COUNT(*) AS total_records FROM range_part2
    UNION ALL
    SELECT COUNT(*) AS total_records FROM range_part3
    UNION ALL
    SELECT COUNT(*) AS total_records FROM range_part4
) AS subquery_counts;
total_records_in_range_partitions
-----
21

```

Số bản ghi ở mỗi phân mảnh:

```

dds_assgn1=# SELECT COUNT (*) FROM range_part0;
count
-----
3

```

```

dds_assgn1=# SELECT COUNT (*) FROM range_part1;
count
-----
3

```

```

dds_assgn1=# SELECT COUNT (*) FROM range_part2;
count
-----
4

```

```

dds_assgn1=# SELECT COUNT (*) FROM range_part3;
count
-----
5

```

```

dds_assgn1=# SELECT COUNT (*) FROM range_part4;
count
-----
6

```

=> Bảng thống kê:

| Tên phân mảnh | Số lượng bản ghi |
|---------------|------------------|
| range_part0 | 3 |
| range_part1 | 3 |
| range_part2 | 4 |
| range_part3 | 5 |
| range_part4 | 6 |
| Tổng | 21 |

- RoundRobin_Partition():

Số phân mảnh:

```
dds_assgn1=# \dt rrobin_part*
          List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | rrobin_part0   | table | postgres
 public | rrobin_part1   | table | postgres
 public | rrobin_part2   | table | postgres
 public | rrobin_part3   | table | postgres
 public | rrobin_part4   | table | postgres
(5 rows)
```

Tổng số lượng bản ghi:

```
dds_assgn1=# SELECT SUM(count) FROM (
  SELECT COUNT(*) AS count FROM rrobin_part0
  UNION ALL SELECT COUNT(*) AS count FROM rrobin_part1
  UNION ALL SELECT COUNT(*) AS count FROM rrobin_part2
  UNION ALL SELECT COUNT(*) AS count FROM rrobin_part3
  UNION ALL SELECT COUNT(*) AS count FROM rrobin_part4
) AS total_rrobin_counts;
 sum
-----
   21
(1 row)
```

Số bản ghi ở mỗi phân mảnh:

```
dds_assgn1=# SELECT COUNT(*) FROM rrobin_part0;
 count
-----
     5
```

```
dds_assgn1=# SELECT COUNT(*) FROM rrobin_part1;
 count
-----
     4
```

```
dds_assgn1=# SELECT COUNT(*) FROM rrobin_part2;
 count
-----
     4
```

```
dds_assgn1=# SELECT COUNT(*) FROM rrobin_part3;
 count
-----
     4
```

```
dds_assgn1=# SELECT COUNT(*) FROM rrobin_part4;
 count
-----
     4
```

=> Bảng thống kê:

| Tên phân mảnh | Số lượng bản ghi |
|---------------|------------------|
| rrobin_part0 | 5 |
| rrobin_part1 | 4 |
| rrobin_part2 | 4 |
| rrobin_part3 | 4 |
| rrobin_part4 | 4 |
| Tổng | 21 |

- Đo thời gian thực thi
 - Range_Partition():

```
--- Hoàn thành phân vùng RANGE trong 0.01 giây ---
rangepartition function pass!
```

- Range_Insert():

```
Đã chèn đánh giá (3) vào phân vùng RANGE 'range_part2' trong 0.0017 giây.
rangeinsert function pass!
```

- RoundRobin_Partition():

```
[ratings] RoundRobinPartition hoàn thành trong 0.05 giây.
roundrobinpartition function pass!
```

- RoundRobin_Insert():

```
[Round Robin Insert] Insert (100, 1, 3) vào rrobin_part0 hoàn thành trong 0.0033 giây. (Index: 0)
roundrobininsert function pass!
```

5.1.2 Test với dữ liệu đầy đủ 10 triệu đánh giá

- Bảng kết quả test các hàm

| Hàm | Kết quả |
|-------------------|---------|
| LoadRatings() | pass |
| Range_Partition() | pass |
| Range_Insert() | pass |

| | |
|------------------------|------|
| RoundRobin_Partition() | pass |
| RoundRobin_Insert() | pass |

- Thống kê phân bố dữ liệu sau phân mảnh

- Range_Partition():

Số phân mảnh:

```

dds_assgn1=# \dt range_part*
          List of relations
Schema |      Name      | Type  | Owner
-----+-----+-----+-----
public | range_part0    | table | postgres
public | range_part1    | table | postgres
public | range_part2    | table | postgres
public | range_part3    | table | postgres
public | range_part4    | table | postgres
(5 rows)

```

Tổng số lượng bản ghi:

```

dds_assgn1=# SELECT SUM(record_count) FROM (
    SELECT COUNT(*) AS record_count FROM range_part0
    UNION ALL
    SELECT COUNT(*) AS record_count FROM range_part1
    UNION ALL
    SELECT COUNT(*) AS record_count FROM range_part2
    UNION ALL
    SELECT COUNT(*) AS record_count FROM range_part3
    UNION ALL
    SELECT COUNT(*) AS record_count FROM range_part4
) AS all_range_partitions_counts;
 sum
-----
10000055

```

Số bản ghi mỗi phân mảnh:

```

dds_assgn1=# SELECT COUNT(*) FROM range_part0;
count
-----
 479168
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM range_part1;
count
-----
 908584
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM range_part2;
count
-----
2726855
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM range_part3;
count
-----
3755614
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM range_part4;
count
-----
2129834
(1 row)

```

=> Bảng thống kê:

| Tên phân mảnh | Số lượng bản ghi |
|---------------|------------------|
| range_part0 | 479168 |
| range_part1 | 908584 |
| range_part2 | 2726855 |
| range_part3 | 3755614 |
| range_part4 | 2129834 |

- RoundRobin_Partition(),: Số phân mảnh:

```

dds_assgn1=# \dt rrobin_part*
               List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | rrobin_part0   | table | postgres
 public | rrobin_part1   | table | postgres
 public | rrobin_part2   | table | postgres
 public | rrobin_part3   | table | postgres
 public | rrobin_part4   | table | postgres
(5 rows)

```

Tổng số lượng bản ghi:

```

dds_assgn1=# SELECT SUM(record_count) FROM (
    SELECT COUNT(*) AS record_count FROM rrobin_part0
    UNION ALL
    SELECT COUNT(*) AS record_count FROM rrobin_part1
    UNION ALL
    SELECT COUNT(*) AS record_count FROM rrobin_part2
    UNION ALL
    SELECT COUNT(*) AS record_count FROM rrobin_part3
    UNION ALL
    SELECT COUNT(*) AS record_count FROM rrobin_part4
) AS all_rrobin_partitions_counts;
      sum
-----
 10000055
(1 row)

```

Số bản ghi ở mỗi phân mảnh:


```

dds_assgn1=# SELECT COUNT(*) FROM rrobin_part0;
count
-----
2000012
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM rrobin_part1;
count
-----
2000011
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM rrobin_part2;
count
-----
2000011
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM rrobin_part3;
count
-----
2000011
(1 row)

dds_assgn1=# SELECT COUNT(*) FROM rrobin_part4;
count
-----
2000010
(1 row)

```

=> Bảng thống kê:

| Tên phân mảnh | Số lượng bản ghi |
|---------------|------------------|
| rrobin_part0 | 2000012 |
| rrobin_part1 | 2000011 |
| rrobin_part2 | 2000011 |
| rrobin_part3 | 2000011 |
| rrobin_part4 | 2000010 |
| Tổng | 10000055 |

- Đo thời gian thực thi
 - Range_Partition():

```
--- Hoàn thành phân vùng RANGE trong 51.75 giây ---  
rangepartition function pass!
```

- Range_Insert():

```
Đã chèn đánh giá (3) vào phân vùng RANGE 'range_part2' trong 0.8353 giây.  
rangeinsert function pass!
```

- RoundRobin_Partition():

```
--- Hoàn thành phân vùng ROUND ROBIN trong 125.52 giây ---  
roundrobinpartition function pass!
```

- RoundRobin_Insert():

```
[Chèn Round Robin] Đã chèn (100, 1, 3) vào 'rrobin_part0' trong 0.0113 giây. (Chỉ số: 0)  
roundrobininsert function pass!
```

5.2 Đánh giá hiệu suất

- So sánh hiệu suất giữa Range và Round Robin

Hiệu suất của Hàm Range_Partition() và Range_Insert()

- Hàm Range_Partition():
 - Thời gian thực thi: Hàm rangepartition() hoàn thành quá trình phân chia toàn bộ tập dữ liệu gốc vào các phân mảnh Range trong 51.75 giây.
- Hàm Range_Insert():
 - Thời gian thực thi: Hoàn thành quá trình chèn một bản ghi đơn lẻ vào phân mảnh Range phù hợp trong 0,8353 giây. Với thời gian hoàn thành dưới 1 giây, điều này chứng tỏ cơ chế xác định mảnh đích dựa trên giá trị rating và thực hiện lệnh chèn là nhanh chóng, phù hợp cho các tác vụ chèn dữ liệu phát sinh.

Hiệu suất của Hàm RoundRobin_Partition() và RoundRobin_Insert()

- Hàm RoundRobin_Partition():
 - Thời gian thực thi: mất 125.52 giây. So với rangepartition(), thời gian lâu hơn đáng kể (khoảng 2.4 lần).
- Hàm RoundRobin_Insert():

- Thời gian thực thi: hiệu suất vượt trội trong việc chèn một bản ghi mới, chỉ mất 0.0113 giây. Thời gian này nhanh hơn đáng kể so với `Range_Insert()` (0.8353 giây).
- So sánh Tổng thể và Kết luận

Dựa trên kết quả thực thi:

- Hiệu suất Khởi tạo (Partitioning): `Range Partitioning (Range_Partition())` cho thấy hiệu quả cao hơn đáng kể trong việc khởi tạo ban đầu và phân phối tập dữ liệu lớn. Điều này đặc biệt có lợi trong các tình huống cần thiết lập hệ thống phân mảnh nhanh chóng.
- Hiệu suất Chèn Dữ liệu Mới (Insertion): `Round Robin Partitioning (RoundRobin_Insert())` vượt trội hơn hẳn về tốc độ chèn từng bản ghi đơn lẻ. Với thời gian chèn gần như tức thì, cơ chế này rất phù hợp cho các hệ thống có tần suất ghi dữ liệu cao và liên tục.