

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



**BÁO CÁO BÀI TẬP LỚN**

**BỘ MÔN: CƠ SỞ DỮ LIỆU PHÂN TÁN**

<b>Giảng viên hướng dẫn</b>	<b>: Kim Ngọc Bách</b>
<b>Nhóm lớp học</b>	<b>: 09</b>
<b>Nhóm bài tập lớn</b>	<b>: 17</b>
<b>Thành viên nhóm</b>	<b>: Đỗ Đức Cảnh - B22DCCN086</b>
	<b>Bùi Tiến Dũng - B22DCCN122</b>
	<b>Trần Quang Huy - B22DCCN398</b>

**HÀ NỘI 06/2025**

## PHÂN CÔNG NHIỆM VỤ NHÓM THỰC HIỆN

TT	Công việc / Nhiệm vụ	SV thực hiện
1	- Cài đặt phân mảnh vòng tròn Round Robin - Cài đặt chèn theo phân mảnh RRobinInsert	Đỗ Đức Cảnh
2	- Kết nối đến hệ quản trị DB MySQL - Cài đặt hàm LoadRatings tải dữ liệu vào MySQL	Bùi Tiến Dũng
3	- Cài đặt phân mảnh theo vùng giá trị Range - Cài đặt chèn theo phân mảnh RangeInsert	Trần Quang Huy

# MỤC LỤC

MỤC LỤC .....	1
DANH MỤC CÁC HÌNH VẼ .....	2
LỜI MỞ ĐẦU .....	3
PHẦN 1. TỔNG QUAN BÀI TOÁN .....	4
1.1 Mục tiêu bài toán .....	4
1.2 Phương pháp tiếp cận .....	4
PHẦN 2. CƠ SỞ LÝ THUYẾT .....	5
2.1 Chiến lược phân mảnh .....	5
2.1.1 Phân mảnh theo khoảng (Range Partition) .....	5
2.1.2 Phân mảnh vòng tròn (Round-Robin Partition) .....	7
2.2 Chiến lược chèn dữ liệu .....	10
2.2.1 Chèn theo khoảng (Range Insert) .....	10
2.2.2 Chèn theo vòng tròn (Round-Robin Insert) .....	13
2.3 Công cụ sử dụng .....	15
PHẦN 3. CÀI ĐẶT VÀ KIỂM THỬ .....	16
3.1 Hàm LoadRatings() .....	16
3.1.1 Cách thức hoạt động .....	16
3.1.2 Sự thay đổi trong cơ sở dữ liệu .....	18
3.2 Hàm Range_Partition() .....	20
3.2.1 Cách thức hoạt động .....	21
3.2.2 Sự thay đổi trong cơ sở dữ liệu .....	22
3.3 Hàm RoundRobin_Partition() .....	25
3.3.1 Cách thức hoạt động: .....	25
3.3.2 Sự thay đổi trong cơ sở dữ liệu .....	26
3.4 Hàm RoundRobin_Insert() .....	28
3.4.1 Cách thức hoạt động .....	29
3.4.2 Sự thay đổi trong cơ sở dữ liệu .....	29
3.5 Hàm Range_Insert() .....	31
3.5.1 Cách thức hoạt động .....	31
KẾT LUẬN .....	36

## DANH MỤC CÁC HÌNH VẼ

<i>Hình 1</i> Các thư viện, module được sử dụng .....	16
<i>Hình 2</i> Hàm createdb tạo database .....	17
<i>Hình 3</i> Hàm loadRatings() .....	18
<i>Hình 4</i> MySQL chưa có database dds_assgn1 .....	19
<i>Hình 5</i> Hàm loadRatings được khởi chạy .....	19
<i>Hình 6</i> Dữ liệu được thêm thành công vào MySQL .....	20
<i>Hình 7</i> Hàm rangepartition() .....	21
<i>Hình 8</i> Database khi chưa phân mảnh rangepartition .....	23
<i>Hình 9</i> Database khi đã phân mảnh rangepartition .....	24
<i>Hình 10</i> Hàm rangepartition pass test .....	25
<i>Hình 11</i> Hàm roundrobinpartition() .....	25
<i>Hình 12</i> Database khi chưa phân mảnh roundrobin .....	27
<i>Hình 13</i> Database sau khi phân mảnh roundrobin .....	27
<i>Hình 14</i> Hàm roundrobin_partition pass test .....	28
<i>Hình 15</i> Hàm roundrobininsert() .....	28
<i>Hình 16</i> Database khi chưa insert theo roundrobin .....	30
<i>Hình 17</i> Database sau khi insert theo roundrobin .....	30
<i>Hình 18</i> Hàm roundrobin_insert pass test .....	31
<i>Hình 19</i> Hàm rangeinsert() .....	31
<i>Hình 20</i> Database khi chưa insert theo rangepartition .....	34
<i>Hình 21</i> Database khi đã insert theo rangepartition .....	35
<i>Hình 22</i> Hàm rangeinsert pass test .....	35

## **LỜI MỞ ĐẦU**

Bài tập lớn môn Cơ sở dữ liệu phân tán là cơ hội quý báu để nhóm em áp dụng lý thuyết vào thực tiễn, khám phá các kỹ thuật phân mảnh dữ liệu trên cơ sở dữ liệu quan hệ MySQL thông qua tập dữ liệu quy mô lớn. Với sự hướng dẫn tận tình của thầy Kim Ngọc Bách, chúng em đã hoàn thành việc triển khai các phương pháp phân mảnh ngang (theo khoảng và vòng tròn) cũng như chức năng chèn dữ liệu, từ đó hiểu sâu hơn về cách tối ưu hóa hiệu suất trong hệ thống phân tán. Báo cáo này của nhóm em trình bày về cách giải quyết vấn đề và mô tả quá trình cài đặt phần bài tập lớn. Báo cáo có thể còn thiếu sót, nhóm em rất mong nhận được sự nhận xét của thầy để giúp nhóm hoàn thiện kỹ năng cũng như kiến thức hơn trong tương lai.

# PHẦN 1. TỔNG QUAN BÀI TOÁN

## 1.1 Mục tiêu bài toán

Thông qua bài tập lớn này, nhóm em đặt ra mục tiêu cụ thể như sau: Sử dụng bộ dữ liệu thật với quy mô lớn 10.000.054 dòng từ tệp `ratings.dat` để giúp nhóm có cái nhìn thực tế về cách hệ quản trị cơ sở dữ liệu (CSDL) xử lý dữ liệu. Đồng thời, nhóm hướng tới việc áp dụng các kiến thức đã được học về phân mảnh dữ liệu vào bài tập lớn, bao gồm tìm hiểu và áp dụng cài đặt triển khai thành công hai phương pháp phân mảnh ngang: Phân mảnh theo khoảng giá trị đồng đều của thuộc tính của dữ liệu và phân mảnh theo vòng tròn round-robin.

## 1.2 Phương pháp tiếp cận

Dựa trên mục tiêu bài toán đã nêu, phương pháp tiếp cận của nhóm em được triển khai như sau: Nhóm sử dụng Python kết hợp với thư viện `mysql.connector` để kết nối và thao tác với MySQL, lấy dữ liệu từ tệp `ratings.dat` (quy mô 10.000.054 dòng) hoặc tệp kiểm thử `test_data.dat`. Quá trình bao gồm: (1) Tạo bảng `Ratings` và các bảng phân mảnh trong MySQL thông qua các hàm như `loadratings()` để đọc và chèn dữ liệu theo batch; (2) Thực hiện phân mảnh ngang bằng hàm `Range_Partition()` để chia bảng `Ratings` thành N phân mảnh dựa trên các khoảng giá trị đồng đều của `Rating`, và hàm `RoundRobin_Partition()` để phân phối dữ liệu theo phương pháp vòng tròn round-robin; (3) Xây dựng các hàm `rangeinsert()` và `roundrobininsert()` để chèn bản ghi mới vào đúng phân mảnh phù hợp. Kiểm thử được thực hiện bằng script `main.py` và `testHelper.py` đã được thầy cung cấp để đảm bảo tính hoàn chỉnh, không trùng lặp, và tái cấu trúc, đồng thời đề xuất tối ưu hóa (như batch insert hoặc meta-data) để xử lý dữ liệu lớn hiệu quả.

## PHẦN 2. CƠ SỞ LÝ THUYẾT

### 2.1 Chiến lược phân mảnh

#### 2.1.1 Phân mảnh theo khoảng (Range Partition)

##### 2.1.1.1 Mô tả

Phân mảnh ngang chia dữ liệu trong bảng ratings thành N bảng con (range\_part0, range\_part1, ..., range\_part{numberofpartitions-1}) dựa trên khoảng giá trị của thuộc tính rating. Mỗi bảng con lưu trữ các bản ghi có giá trị rating nằm trong một khoảng xác định, được tính bằng cách chia đều miền giá trị từ 0 đến 5 thành N khoảng đều nhau.

Ứng dụng phù hợp trong các hệ thống cần phân mảnh dữ liệu dựa trên giá trị của một thuộc tính cụ thể (như rating), chẳng hạn trong cơ sở dữ liệu phân tán, khi muốn tối ưu hóa truy vấn dựa trên khoảng giá trị của thuộc tính hoặc phân phối dữ liệu theo đặc tính của thuộc tính để xử lý hiệu quả hơn.

##### 2.1.1.2 Cách thực hiện (Dựa trên hàm 'rangepartition()')

Hàm rangepartition(ratingtablename, numberofpartitions, openconnection) thực hiện phân mảnh dựa trên khoảng giá trị của thuộc tính rating như sau:

Bước 1: Xác định các khoảng giá trị và tạo bảng phân mảnh

- Tính toán độ dài mỗi khoảng bằng cách chia miền giá trị của rating (từ 0 đến 5) thành numberofpartitions phần đều nhau:  $\text{delta} = 5.0 / \text{numberofpartitions}$ .

- Tạo danh sách các ranh giới khoảng: [0, delta, 2\*delta, ..., 5.0].

- Tạo numberofpartitions bảng với tên range\_part0, range\_part1, ..., range\_part{numberofpartitions-1}.

- Mỗi bảng có schema giống bảng ratingtablename:

- + userid (INT)

- + movieid (INT)

- + rating (FLOAT)

- Nếu bảng đã tồn tại, xóa bảng cũ bằng câu lệnh SQL:

DROP TABLE IF EXISTS range\_part{i}

- Tạo bảng mới bằng:

CREATE TABLE range\_part{i} (userid INT, movieid INT, rating FLOAT)

Bước 2: Phân phối dữ liệu

- Với mỗi bảng range\_part{i} (i từ 0 đến numberofpartitions-1):

- Xác định khoảng giá trị [lower\_bound, upper\_bound] tương ứng, trong đó:

$$\text{lower\_bound} = i * \text{delta}$$

$$upper\_bound = (i + 1) * delta$$

- Chèn các bản ghi từ bảng ratingtablename vào range\_part{i} dựa trên giá trị rating:

Nếu  $i = 0$ , chèn các bản ghi có rating thỏa mãn:  $rating \geq lower\_bound$  AND  $rating \leq upper\_bound$ .

Nếu  $i > 0$ , chèn các bản ghi có rating thỏa mãn:  $rating > lower\_bound$  AND  $rating \leq upper\_bound$ .

Câu lệnh SQL:

```
INSERT INTO range_part{i}
```

```
SELECT * FROM ratingtablename
```

```
WHERE rating >= lower_bound AND rating <= upper_bound -- (cho i = 0)
```

Hoặc:

```
INSERT INTO range_part{i}
```

```
SELECT * FROM ratingtablename
```

```
WHERE rating > lower_bound AND rating <= upper_bound -- (cho i > 0)
```

#### 2.1.1.3 Ví dụ

- Giả sử bảng ratingtablename có 10 bản ghi với các giá trị rating là [1.0, 2.5, 3.0, 4.5, 2.0, 3.5, 1.5, 5.0, 4.0, 2.8] và numberofpartitions = 2:

- Tính  $delta = 5.0 / 2 = 2.5$ . Các ranh giới khoảng: [0, 2.5, 5.0].

- Hai bảng phân mảnh: range\_part0 (cho khoảng [0, 2.5]) và range\_part1 (cho khoảng (2.5, 5.0]).

- Phân bổ như sau:

- Bản ghi với rating trong [0, 2.5]:

rating = 1.0, 2.0, 1.5, 2.5 → vào range\_part0.

Bản ghi với rating trong (2.5, 5.0]:

rating = 3.0, 4.5, 3.5, 5.0, 4.0, 2.8 → vào range\_part1.

Kết quả:

range\_part0: Bản ghi với rating = 1.0, 2.0, 1.5, 2.5 (4 bản ghi).

range\_part1: Bản ghi với rating = 3.0, 4.5, 3.5, 5.0, 4.0, 2.8 (6 bản ghi).

- Phân phối này đảm bảo các bản ghi được chia theo khoảng giá trị của rating, phù hợp cho các truy vấn lọc dựa trên khoảng giá trị cụ thể.

#### 2.1.1.4 Ưu điểm



-Tối ưu cho truy vấn theo khoảng: Dữ liệu được phân mảnh dựa trên giá trị của thuộc tính rating, giúp tối ưu hóa các truy vấn lọc theo khoảng giá trị (ví dụ: tìm tất cả bản ghi có rating từ 2.0 đến 3.0).

-Phân phối dựa trên đặc tính dữ liệu: Phân mảnh theo khoảng giá trị của rating cho phép hệ thống phân tán xử lý hiệu quả hơn khi các truy vấn tập trung vào các khoảng giá trị cụ thể.

-Dễ triển khai: Việc xác định các khoảng giá trị và phân phối bản ghi dựa trên điều kiện rating đơn giản, sử dụng các câu lệnh SQL cơ bản.

#### *2.1.1.5 Nhược điểm*

-Phân phối không đều: Số lượng bản ghi trong mỗi phân mảnh phụ thuộc vào phân bố giá trị rating trong bảng gốc. Nếu dữ liệu tập trung vào một vài khoảng giá trị, một số bảng phân mảnh có thể chứa nhiều bản ghi hơn các bảng khác, dẫn đến mất cân bằng tải.

-Không linh hoạt với các truy vấn không dựa trên rating: Vì phân mảnh dựa hoàn toàn vào thuộc tính rating, các truy vấn dựa trên các thuộc tính khác (như userid hoặc movieid) có thể yêu cầu truy cập nhiều hoặc tất cả các bảng phân mảnh, làm giảm hiệu suất.

-Phụ thuộc vào phạm vi giá trị: Hàm giả định giá trị rating nằm trong khoảng  $[0, 5]$ . Nếu dữ liệu có giá trị ngoài khoảng này hoặc phân bố không đều, cần điều chỉnh logic phân mảnh.

#### *2.1.1.6 Đảm bảo tính đúng đắn phân mảnh*

- Đầy đủ: Tất cả bản ghi trong bảng ratingtablename được phân vào một trong numberofpartitions bảng phân mảnh, vì các khoảng  $[0, \text{delta}]$ ,  $(\text{delta}, 2*\text{delta}]$ , ...,  $((N-1)*\text{delta}, 5.0]$  bao phủ toàn bộ miền giá trị của rating từ 0 đến 5.

- Tách biệt: Mỗi bản ghi chỉ thuộc một bảng phân mảnh, vì các khoảng giá trị không chồng lấn (trừ điểm ranh giới, được xử lý bằng điều kiện  $\geq$  cho bảng đầu tiên và  $>$  cho các bảng còn lại).

- Tái thiết: Có thể hợp nhất các bảng phân mảnh bằng câu lệnh UNION ALL để tái tạo bảng ratingtablename gốc, đảm bảo không mất dữ liệu.

### **2.1.2 Phân mảnh vòng tròn (Round-Robin Partition)**

#### *2.1.2.1 Mô tả*

Phân mảnh vòng tròn chia dữ liệu trong bảng 'Ratings' thành N bảng con ('rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part{N-1}') theo cách tuần hoàn, không phụ thuộc vào giá trị của bất kỳ thuộc tính nào trong bảng. Mỗi bản ghi được phân phối lần lượt vào các bảng phân mảnh để đảm bảo số lượng bản ghi trong các bảng là gần bằng nhau.

Ứng dụng phù hợp trong các hệ thống phân tán khi cần phân phối đều dữ liệu để cân bằng tải, đặc biệt khi không có thuộc tính nào được ưu tiên để phân mảnh (như trong trường hợp phân mảnh theo khoảng dựa trên 'Rating').

#### 2.1.2.2 Cách thực hiện (Dựa trên hàm 'roundrobinpartition()')

Hàm 'roundrobinpartition(ratingtablename, numberofpartitions, openconnection)' thực hiện phân mảnh vòng tròn như sau:

Bước 1: Tạo các bảng phân mảnh

- Tạo N bảng với tên 'rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part{N-1}'.

- Mỗi bảng có schema giống bảng 'Ratings':

- UserId (int)
- MovieId (int)
- Rating (float)

- Nếu bảng đã tồn tại, xóa bảng cũ bằng câu lệnh SQL:

```
DROP TABLE IF EXISTS rrobin_part{i}
```

- Tạo bảng mới bằng:

```
CREATE TABLE rrobin_part{i} (userid INT, movieid INT, rating FLOAT)
```

Bước 2: Phân phối dữ liệu

- Sử dụng câu lệnh SQL với 'ROW\_NUMBER()' để gán số thứ tự ('rnum') cho mỗi bản ghi trong bảng 'Ratings', bắt đầu từ 1.

- Phân bổ bản ghi vào bảng 'rrobin\_part{i}' dựa trên công thức:

$$i = (rnum - 1) \bmod N$$

- Câu lệnh SQL:

```
INSERT INTO rrobin_part{i} (userid, movieid, rating)
```

```
SELECT userid, movieid, rating
```

```
FROM (
```

```
  SELECT
```

```
    ROW_NUMBER() OVER () AS rnum,
```

```
    userid,
```

```
    movieid,
```

```
    rating
```

```
  FROM ratings
```

) AS temp

WHERE (rnum-1) % numberofpartitions = i

Với mỗi i từ 0 đến N-1, hàm chèn các bản ghi có  $(\text{rnum} - 1) \bmod N = i$  vào bảng `rrobin\_part{i}`.

#### 2.1.2.3 Ví dụ

Giả sử bảng `Ratings` có 10 bản ghi và  $N = 3$  (tạo 3 phân mảnh: `rrobin\_part0`, `rrobin\_part1`, `rrobin\_part2`):

- Các bản ghi được gán số thứ tự từ 1 đến 10 (rnum).
- Phân bổ như sau:
  - Bản ghi 1 ( rnum = 1 ):  $(1-1) \bmod 3 = 0$ , vào `rrobin\_part0`.
  - Bản ghi 2 ( rnum = 2 ):  $(2-1) \bmod 3 = 1$ , vào `rrobin\_part1`.
  - Bản ghi 3 ( rnum = 3 ):  $(3-1) \bmod 3 = 2$ , vào `rrobin\_part2`.
  - Bản ghi 4 ( rnum = 4 ):  $(4-1) \bmod 3 = 0$ , vào `rrobin\_part0`.
  - Bản ghi 5 ( rnum = 5 ):  $(5-1) \bmod 3 = 1$ , vào `rrobin\_part1`.
  - ...
- Kết quả:
  - `rrobin\_part0`: Bản ghi 1, 4, 7, 10 (4 bản ghi).
  - `rrobin\_part1`: Bản ghi 2, 5, 8 (3 bản ghi).
  - `rrobin\_part2`: Bản ghi 3, 6, 9 (3 bản ghi).

Phân phối này đảm bảo số lượng bản ghi trong mỗi bảng gần bằng nhau (khác biệt tối đa 1 bản ghi).

#### 2.1.2.4 Ưu điểm

- Phân phối đều: Số lượng bản ghi trong mỗi phân mảnh gần bằng nhau, giúp cân bằng tải trong hệ thống phân tán.
- Đơn giản: Không cần phân tích giá trị thuộc tính, chỉ dựa trên thứ tự bản ghi, dễ triển khai.
- Linh hoạt: Phù hợp với nhiều loại dữ liệu, không yêu cầu thuộc tính cụ thể như phân mảnh theo khoảng.

#### 2.1.2.5 Nhược điểm

- Không tối ưu cho truy vấn cụ thể: Vì dữ liệu được phân phối tuần hoàn, không dựa trên giá trị thuộc tính, nên không tối ưu cho các truy vấn lọc theo điều kiện (ví dụ: tìm tất cả bản ghi có `Rating > 4`).
- Phụ thuộc vào thứ tự: Phân mảnh dựa trên `ROW\_NUMBER()`, nên nếu dữ liệu trong bảng `Ratings` thay đổi thứ tự, cần đảm bảo tính nhất quán.

#### 2.1.2.6 Đảm bảo tính đúng đắn phân mảnh

- Đầy đủ: Tất cả bản ghi trong bảng 'Ratings' được phân vào một trong N bảng phân mảnh, không bỏ sót bản ghi nào.
- Tách biệt: Mỗi bản ghi chỉ thuộc một bảng phân mảnh, vì  $(rnum - 1) \bmod N$  chỉ trả về một giá trị duy nhất.
- Tái thiết: Có thể hợp nhất các bảng phân mảnh bằng 'UNION ALL' để tái tạo bảng 'Ratings' gốc.

## 2.2 Chiến lược chèn dữ liệu

### 2.2.1 Chèn theo khoảng (Range Insert)

#### 2.2.1.1 Mô tả

Hàm `rangeinsert(ratingtablename, userid, itemid, rating, openconnection)` chèn một bản ghi mới (`userid, movieid, rating`) vào bảng chính `ratingtablename` và đồng thời vào một trong các bảng phân mảnh dựa trên giá trị của `rating` (`range_part0, range_part1, ..., range_part{N-1}`). Chiến lược phân mảnh dựa trên phạm vi (`range partitioning`) được sử dụng, trong đó giá trị `rating` được chia thành các khoảng đều nhau, mỗi khoảng tương ứng với một bảng phân mảnh. Bản ghi sẽ được chèn vào bảng phân mảnh tương ứng dựa trên giá trị `rating` của nó.

Ứng dụng phù hợp trong các hệ thống cần phân mảnh dữ liệu theo giá trị của một thuộc tính (ở đây là `rating`), giúp tối ưu hóa truy vấn dựa trên phạm vi giá trị, chẳng hạn như tìm kiếm hoặc phân tích dữ liệu trong một khoảng `rating` cụ thể.

#### 2.2.1.2 Cách thực hiện

Hàm `rangeinsert()` thực hiện chèn dữ liệu vào bảng chính và bảng phân mảnh theo khoảng dựa trên giá trị `rating` như sau:

Bước 1: Chèn vào bảng chính

Chèn bản ghi mới vào bảng `ratingtablename` bằng câu lệnh SQL:

```
INSERT INTO {ratingtablename} (userid, movieid, rating) VALUES (%s, %s, %s)
```

Điều này đảm bảo bảng chính chứa toàn bộ dữ liệu gốc, bao gồm bản ghi mới với `userid, movieid, và rating`.

Bước 2: Xác định phân mảnh đích

Đếm số bảng phân mảnh có tiền tố `range_part`:

```
SELECT COUNT(*) FROM information_schema.tables
```

```
WHERE table_schema = DATABASE() AND table_name LIKE 'range_part%';
```

Kết quả trả về số lượng phân mảnh `numberofpartitions (N)`.

Tính kích thước khoảng `rating` cho mỗi phân mảnh:

$$\text{delta} = 5.0 / \text{numberofpartitions}$$

- Giả sử rating nằm trong khoảng [0, 5], mỗi phân mảnh sẽ chứa các giá trị rating trong một khoảng delta.

- Tính chỉ số phân mảnh dựa trên giá trị rating:

$\text{index} = \min(\text{int}(\text{rating} / \text{delta}) - 1 \text{ if } \text{rating} == \text{int}(\text{rating}) \text{ else } \text{int}(\text{rating} / \text{delta}), \text{numberofpartitions} - 1)$

Công thức chia rating cho delta để xác định phân mảnh tương ứng.

- Nếu rating là số nguyên, trừ 1 để điều chỉnh chỉ số (đảm bảo rating = 5 rơi vào phân mảnh cuối).

- Giới hạn index để không vượt quá numberofpartitions - 1.

- Nếu index nhỏ hơn 0 (trường hợp rating rất nhỏ), đặt index = 0.

- Tên bảng phân mảnh được xác định là range\_part{index}.

Bước 3: Chèn vào bảng phân mảnh:

Chèn bản ghi vào bảng phân mảnh range\_part{index}:

INSERT INTO {table\_name} (userid, movieid, rating) VALUES (%s, %s, %s)

Cam kết giao dịch (commit) để lưu thay đổi vào cơ sở dữ liệu.

### 2.2.1.3 Ví dụ

- Giả sử có 5 phân mảnh (range\_part0, range\_part1, ..., range\_part4), và ratingtablename = Ratings. Với numberofpartitions = 5, ta có:

-->  $\text{delta} = 5.0 / 5 = 1.0$ , các khoảng rating cho các phân mảnh:

range\_part0: [0, 1)

range\_part1: [1, 2)

range\_part2: [2, 3)

range\_part3: [3, 4)

range\_part4: [4, 5]

Trường hợp 1: Chèn bản ghi (userid=100, movieid=1, rating=3.5)

Bước 1: Chèn vào bảng Ratings:

INSERT INTO Ratings (userid, movieid, rating) VALUES (100, 1, 3.5)

Bước 2: Tính chỉ số phân mảnh:

$\text{delta} = 1.0$

rating = 3.5, không phải số nguyên, nên  $\text{index} = \text{int}(3.5 / 1.0) = 3$

--> Bảng đích: range\_part3

Bước 3: Chèn vào range\_part3

```
INSERT INTO range_part3 (userid, movieid, rating) VALUES (100, 1, 3.5)
```

Trường hợp 2: Chèn bản ghi (userid=200, movieid=2, rating=5.0)

Bước 1: Chèn vào bảng Ratings:

```
INSERT INTO Ratings (userid, movieid, rating) VALUES (200, 2, 5.0)
```

Bước 2: Tính chỉ số phân mảnh:

rating = 5.0, là số nguyên, nên  $\text{index} = \text{int}(5.0 / 1.0) - 1 = 4$

--> Bảng đích: range\_part4

Bước 3: Chèn vào range\_part4

```
INSERT INTO range_part4 (userid, movieid, rating) VALUES (200, 2, 5.0)
```

Trường hợp 3: Chèn bản ghi (userid=300, movieid=3, rating=0.5)

Bước 1: Chèn vào bảng Ratings:

```
INSERT INTO Ratings (userid, movieid, rating) VALUES (300, 3, 0.5)
```

Bước 2: Tính chỉ số phân mảnh:

rating = 0.5, không phải số nguyên, nên  $\text{index} = \text{int}(0.5 / 1.0) = 0$

--> Bảng đích: range\_part0

Bước 3: Chèn vào range\_part0:

```
INSERT INTO range_part0 (userid, movieid, rating) VALUES (300, 3, 0.5)
```

#### 2.2.1.4 Ưu điểm

- Phân mảnh theo ngữ nghĩa: Dữ liệu được phân phối dựa trên giá trị rating, giúp các truy vấn tìm kiếm theo khoảng rating (ví dụ: tìm tất cả bản ghi có rating từ 3 đến 4) chỉ cần truy cập một hoặc vài bảng phân mảnh, cải thiện hiệu suất.

- Đơn giản trong thiết kế: Công thức tính chỉ số phân mảnh dựa trên rating dễ hiểu và dễ triển khai.

- Tính linh hoạt: Có thể điều chỉnh số lượng phân mảnh bằng cách thay đổi số bảng range\_part, và khoảng rating được tự động chia đều.

#### 2.2.1.5 Nhược điểm

- Phân phối không đều: Nếu các giá trị rating không phân bố đồng đều (ví dụ: nhiều rating tập trung quanh 3-4), một số bảng phân mảnh có thể chứa nhiều bản ghi hơn, gây mất cân bằng tải.

- Hiệu suất với số lượng lớn bảng phân mảnh: Việc kiểm tra số bảng phân mảnh qua information\_schema.tables có thể tốn thời gian khi cơ sở dữ liệu có nhiều bảng.

- Phụ thuộc vào giá trị rating: Nếu rating nằm ngoài khoảng [0, 5] hoặc có giá trị không hợp nhiều, logic xác định phân mảnh có thể gặp lỗi hoặc cần điều chỉnh thêm.

#### 2.2.1.6 Đảm bảo tính đúng đắn

-Đầy đủ: Mỗi bản ghi được chèn vào cả bảng chính (Ratings) và một bảng phân mảnh, đảm bảo không bỏ sót dữ liệu.

-Tách biệt: Mỗi bản ghi chỉ được chèn vào một bảng phân mảnh duy nhất, dựa trên giá trị rating và công thức tính chỉ số.

-Tái thiết: Có thể hợp nhất dữ liệu từ các bảng phân mảnh (range\_part\*) và so sánh với bảng chính để kiểm tra tính toàn vẹn.

-Xử lý trường hợp đặc biệt: Hàm xử lý tốt các giá trị rating số nguyên và không nguyên, đồng thời đảm bảo index không vượt quá số phân mảnh hoặc âm.

### 2.2.2 Chèn theo vòng tròn (Round-Robin Insert)

#### 2.2.2.1 Mô tả

Hàm `roundrobininsert(ratingtablename, userid, itemid, rating, openconnection)` chèn một bản ghi mới ('userid', 'movieid', 'rating') vào bảng 'Ratings' và đồng thời vào một trong các bảng phân mảnh vòng tròn ('rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part{N-1}') theo cách tuần hoàn. Chiến lược này đảm bảo bản ghi mới được phân phối đều vào các phân mảnh hiện có, duy trì tính cân bằng của phân mảnh vòng tròn.

Ứng dụng phù hợp trong hệ thống phân tán khi cần chèn dữ liệu mới mà vẫn giữ phân phối đều giữa các phân mảnh, không dựa vào giá trị của thuộc tính nào.

#### 2.2.2.2 Cách thực hiện (Dựa trên hàm `roundrobininsert()`)

Hàm `roundrobininsert()` thực hiện chèn dữ liệu theo vòng tròn như sau:

Bước 1: Chèn vào bảng chính

- Chèn bản ghi mới vào bảng 'Ratings' bằng câu lệnh SQL:

```
INSERT INTO ratings (userid, movieid, rating) VALUES (userid, itemid, rating)
```

- Điều này đảm bảo bảng 'Ratings' chứa toàn bộ dữ liệu gốc, bao gồm cả bản ghi mới.

Bước 2: Xác định phân mảnh đích

- Đếm tổng số bản ghi hiện tại trong bảng 'Ratings' để xác định số thứ tự của bản ghi mới ('total\_rows'):

```
SELECT COUNT(*) FROM ratings
```

- Đếm số phân mảnh N bằng cách kiểm tra số bảng có tiền tố 'rrobin\_part':

```
SELECT COUNT(*) FROM information_schema.tables
```

WHERE table\_schema = DATABASE() AND table\_name LIKE 'rrobin\_part%';

- Tính chỉ số phân mảnh cho bản ghi mới:

$$index = (total\_rows - 1) \bmod N$$

- 'total\_rows - 1' được sử dụng vì bản ghi mới vừa được chèn, và nó sẽ có số thứ tự là 'total\_rows'.

- Công thức modulo chia dư đảm bảo bản ghi được phân phối tuần hoàn vào các bảng 'rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part{N-1}'.

Bước 3: Chèn vào bảng phân mảnh

- Chèn bản ghi vào bảng 'rrobin\_part{index}':

INSERT INTO rrobin\_part{index} (userid, movieid, rating)

VALUES (userid, itemid, rating)

#### 2.2.2.3 Ví dụ

Giả sử bảng 'Ratings' hiện có 20 bản ghi, và có  $N = 5$  phân mảnh vòng tròn ('rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part4'). Khi chèn một bản ghi mới (userid=100, movieid=1, rating=3)

Bước 1: Chèn vào bảng 'Ratings'. Sau khi chèn, bảng có 21 bản ghi ('total\_rows = 21').

Bước 2: Tính chỉ số phân mảnh:

$$index = (total\_rows - 1) \bmod N = (21 - 1) \bmod 5 = 20 \bmod 5 = 0$$

=> Bản ghi sẽ được chèn vào 'rrobin\_part0'.

Bước 3: Thực hiện chèn:

INSERT INTO rrobin\_part0 (userid, movieid, rating) VALUES (100, 1, 3)

Kết quả:

- Bảng 'Ratings' có thêm bản ghi: (100, 1, 3).

- Bảng 'rrobin\_part0' có thêm bản ghi: (100, 1, 3).

Nếu chèn bản ghi tiếp theo:

-  $index = total\_rows = 22$ ,  $(22 - 1) \bmod 5 = 21 \bmod 5 = 1$ , bản ghi được chèn vào 'rrobin\_part1'.

- Quá trình tiếp tục tuần hoàn, đảm bảo phân phối đều.

#### 2.2.2.4 Ưu điểm

- Duy trì tính cân bằng: Bản ghi mới được phân phối đều vào các phân mảnh, giữ số lượng bản ghi trong mỗi bảng gần bằng nhau (khác biệt tối đa 1 bản ghi).



- Đơn giản: Chỉ cần đếm số bản ghi và số phân mảnh, không cần phân tích giá trị thuộc tính như trong phân mảnh theo khoảng.

- Tính nhất quán: Đảm bảo bản ghi mới được chèn vào đúng phân mảnh theo thứ tự tuần hoàn, phù hợp với chiến lược phân mảnh vòng tròn ban đầu.

#### 2.2.2.5 Nhược điểm

- Hiệu suất với dữ liệu lớn: Việc đếm `'total_rows'` và số phân mảnh mỗi lần chèn có thể làm giảm hiệu suất khi bảng `'Ratings'` có hàng triệu bản ghi (như 10 triệu bản ghi trong tập MovieLens).

- Phụ thuộc vào số bản ghi: Nếu bảng `'Ratings'` bị xóa hoặc thêm dữ liệu không qua hàm `'roundrobininsert()'`, số thứ tự có thể không nhất quán, dẫn đến sai lệch phân mảnh.

#### 2.2.2.6 Đảm bảo tính đúng đắn

- Đầy đủ: Mỗi bản ghi mới được chèn vào cả bảng `'Ratings'` và một bảng phân mảnh, đảm bảo không bỏ sót dữ liệu.

- Tách biệt: Mỗi bản ghi chỉ được chèn vào một bảng phân mảnh duy nhất, nhờ công thức modulo.

- Tái thiết: Có thể hợp nhất các bảng phân mảnh với bảng `'Ratings'` để kiểm tra tính toàn vẹn.

## 2.3 Công cụ sử dụng

Nhóm em đã sử dụng các công cụ sau để làm bài tập:

- Ngôn ngữ lập trình: Python 3.12, được chọn nhờ tính linh hoạt và hỗ trợ mạnh mẽ cho xử lý dữ liệu, đặc biệt phù hợp với các thao tác trên cơ sở dữ liệu.

- Thư viện: `'mysql.connector'`, thư viện Python dùng để kết nối và thực hiện các truy vấn SQL với MySQL, đảm bảo giao tiếp hiệu quả giữa mã nguồn và cơ sở dữ liệu.

- Hệ quản trị cơ sở dữ liệu: MySQL, được sử dụng để lưu trữ và quản lý bảng `'Ratings'` cùng các bảng phân mảnh, cung cấp môi trường thực tế cho việc mô phỏng phân mảnh dữ liệu.

- Dữ liệu đầu vào: Tập dữ liệu MovieLens từ tệp `'ratings.dat'` (10.000.054 dòng) để mô phỏng quy mô lớn, và tệp `'test_data.dat'` (20 dòng) để kiểm thử nhanh chóng.

- Môi trường phát triển: Máy dùng hệ điều hành Windows, đã cài đặt Python và MySQL, đảm bảo tính tương thích và dễ dàng triển khai.

## PHẦN 3. CÀI ĐẶT VÀ KIỂM THỬ

### 3.1 Hàm LoadRatings()

#### 3.1.1 Cách thức hoạt động

Trước hết ta cần kết nối chương trình Python với cơ sở dữ liệu MySQL, cần cài đặt thư viện mysql-connector-python bằng lệnh:

*pip install mysql-connector-python*

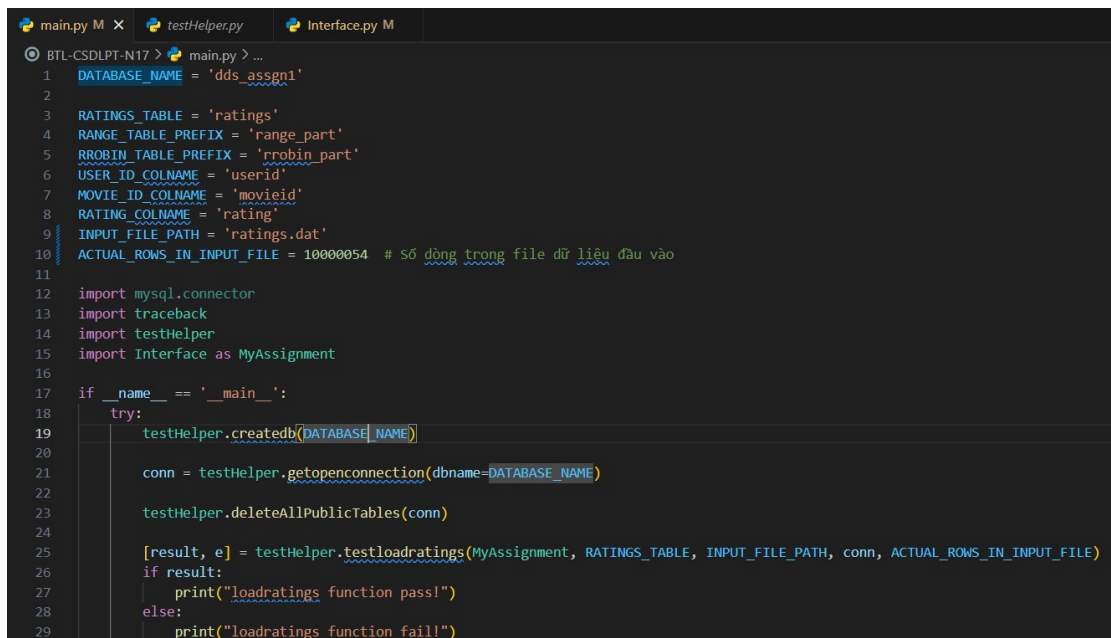
Tiếp theo ta cần import các thư viện và module cần thiết:

import mysql.connector: kết nối với MySQL từ Python.

import traceback: phục vụ xử lý lỗi.

import testHelper: module hỗ trợ kiểm thử

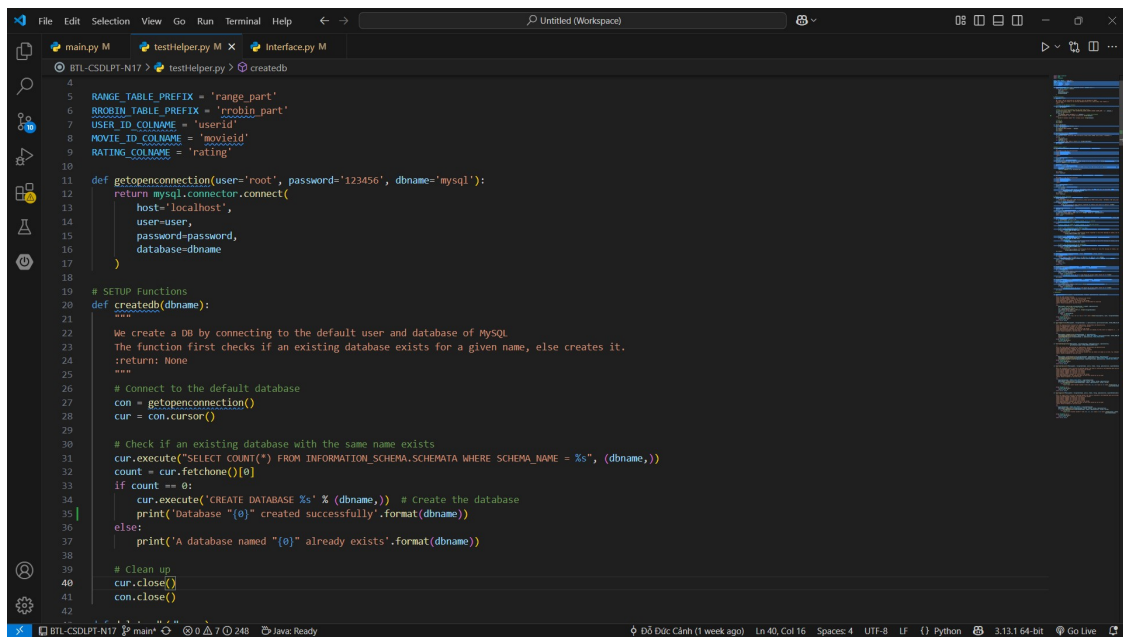
import Interface: module chính chứa các hàm cần cài đặt, trong đó có loadratings()



```
main.py M X testHelper.py Interface.py M
BTU-CSDLPT-N17 > main.py > ...
1 DATABASE_NAME = 'dds_assign1'
2
3 RATINGS_TABLE = 'ratings'
4 RANGE_TABLE_PREFIX = 'range_part'
5 RRROBIN_TABLE_PREFIX = 'rrobin_part'
6 USER_ID_COLNAME = 'userid'
7 MOVIE_ID_COLNAME = 'movieid'
8 RATING_COLNAME = 'rating'
9 INPUT_FILE_PATH = 'ratings.dat'
10 ACTUAL_ROWS_IN_INPUT_FILE = 10000054 # Số dòng trong file dữ liệu đầu vào
11
12 import mysql.connector
13 import traceback
14 import testHelper
15 import Interface as MyAssignment
16
17 if __name__ == '__main__':
18     try:
19         testHelper.createdb(DATABASE_NAME)
20
21         conn = testHelper.getopenconnection(dbname=DATABASE_NAME)
22
23         testHelper.deleteAllPublicTables(conn)
24
25         [result, e] = testHelper.testloadratings(MyAssignment, RATINGS_TABLE, INPUT_FILE_PATH, conn, ACTUAL_ROWS_IN_INPUT_FILE)
26         if result:
27             print("loadratings function pass!")
28         else:
29             print("loadratings function fail!")
```

Hình 1 Các thư viện, module được sử dụng

Trong quá trình khởi tạo hệ thống, chương trình cần đảm bảo rằng cơ sở dữ liệu cần thiết đã tồn tại để thực hiện các thao tác tiếp theo. Để làm điều này, hàm createdb(dbname) được sử dụng nhằm kiểm tra và tạo mới cơ sở dữ liệu MySQL nếu chưa tồn tại.



Hình 2 Hàm `createdb` tạo database

Cụ thể, hàm thực hiện các bước như sau:

- + Kết nối đến MySQL Server thông qua hàm `getopenconnection()`.
- + Truy vấn bảng hệ thống `INFORMATION_SCHEMA.SCHEMATA` để kiểm tra xem cơ sở dữ liệu có tên `dbname` đã tồn tại hay chưa.
- + Nếu chưa tồn tại, hàm thực hiện lệnh `CREATE DATABASE` để tạo mới cơ sở dữ liệu, thông báo đã tạo thành công cơ sở dữ liệu.
- + Nếu đã tồn tại, chương trình chỉ in thông báo và không thực hiện thêm thao tác tạo.
- + Đóng kết nối và cursor sau khi hoàn tất để giải phóng tài nguyên.

Hàm `getopenconnection()` được xây dựng nhằm thực hiện chức năng thiết lập kết nối đến cơ sở dữ liệu MySQL.

Cụ thể, hàm sử dụng thư viện `mysql.connector` để thiết lập kết nối với MySQL tại địa chỉ `localhost`, với các thông số mặc định:

`user='root'`: tên người dùng đăng nhập MySQL.

`password='123456'`: mật khẩu đăng nhập.

`dbname='mysql'`: tên cơ sở dữ liệu mặc định để kết nối ban đầu (có thể thay đổi khi cần).

Hàm trả về một đối tượng kết nối (connection object) để có thể sử dụng tiếp trong các hàm khác như tạo cơ sở dữ liệu, truy vấn dữ liệu, phân mảnh bảng, hay kiểm thử chức năng. Việc đóng gói thao tác kết nối vào một hàm riêng giúp mã nguồn gọn gàng, dễ bảo trì và tái sử dụng.

Hàm `loadratings` có nhiệm vụ đọc dữ liệu từ tệp `ratings.dat` và nạp vào bảng `ratings` trong cơ sở dữ liệu MySQL.

```

17 def loadratings(ratingtablename, ratingsfilepath, openconnection):
18     create_db(DATABASE_NAME)
19     cur = openconnection.cursor()
20     # Xóa bảng nếu đã tồn tại
21     cur.execute("DROP TABLE IF EXISTS {}".format(ratingtablename))
22     # Tạo bảng ratings
23     cur.execute("CREATE TABLE {} (userid INT, movieid INT, rating FLOAT)".format(ratingtablename))
24     # Đọc và chèn dữ liệu
25     batch = []
26     count = 0
27
28     with open(ratingsfilepath, "r", encoding="utf-8") as f:
29         for line in f:
30             parts = line.strip().split("::")
31             if len(parts) == 4:
32                 user, movie, rating, _ = parts
33                 batch.append((int(user), int(movie), float(rating)))
34                 count += 1
35
36             if len(batch) == 1000000:
37                 cur.executemany(
38                     f"INSERT IGNORE INTO {ratingtablename} (userid, movieid, rating) VALUES (%s, %s, %s)",
39                     batch
40                 )
41                 openconnection.commit()
42                 print(f"Đã thêm {count} bản ghi vào bảng {ratingtablename}.") # In tổng số dòng đã thêm
43                 batch.clear()
44
45     if batch:
46         cur.executemany(
47             f"INSERT IGNORE INTO {ratingtablename} (userid, movieid, rating) VALUES (%s, %s, %s)",
48             batch
49         )
50         openconnection.commit()
51         print(f"Đã thêm {count} bản ghi vào bảng {ratingtablename}.") # Thông báo cho batch cuối
52
53     cur.close()
54     print(f"Đã tải dữ liệu thành công: {count} bản ghi đã thêm vào bảng {ratingtablename}.")
55

```

Hình 3 Hàm loadRatings()

Mục đích:

- + Tạo bảng ratings gồm 3 trường: userid, movieid, rating.
- + Chèn dữ liệu từ file vào bảng một cách hiệu quả bằng cơ chế batch insert.

Chi tiết hoạt động:

- Tạo bảng:

- + Tạo bảng mới với 3 cột userid INT, movieid INT, rating float

- Đọc dữ liệu từ tệp:

- + Mỗi dòng trong file ratings.dat có định dạng: userid::movieid::rating::timestamp
- + Hàm chỉ lấy 3 giá trị userid, movieid, rating.

- Chèn dữ liệu:

- + Sử dụng list batch để gom dữ liệu.
- + Mỗi 1.000.000 dòng sẽ được chèn một lần bằng executemany nhằm tăng tốc độ và giảm số lần truy vấn

- Xử lý phần dư cuối cùng:

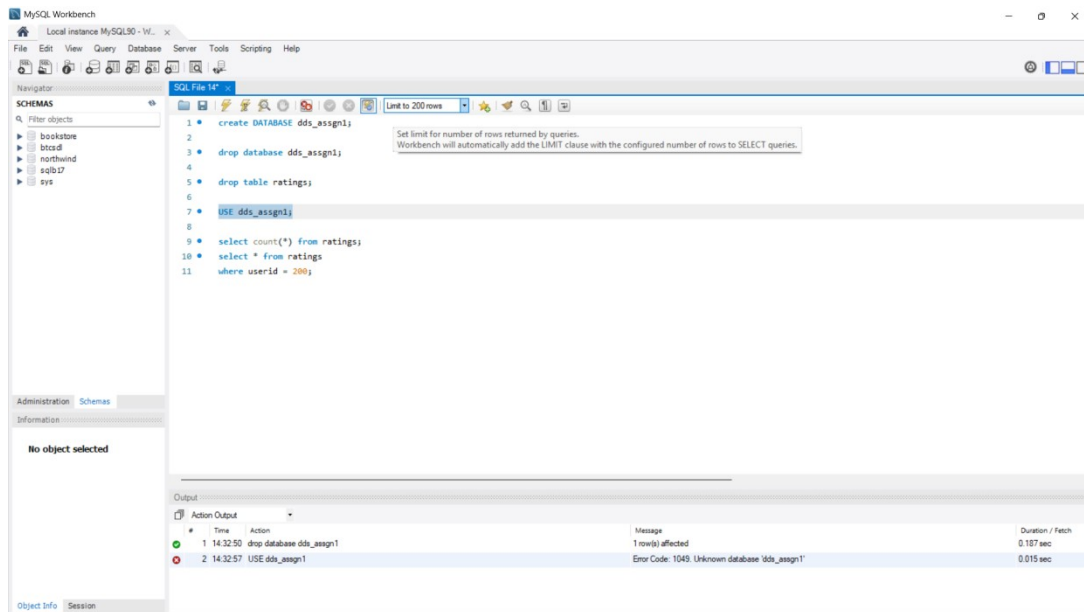
- + Sau vòng lặp nếu còn dữ liệu trong batch thì chèn nốt.

- In thông báo và đóng kết nối:

- + In số dòng đã chèn để dễ kiểm tra
- + Đóng cursor để giải phóng tài nguyên.

### 3.1.2 Sự thay đổi trong cơ sở dữ liệu

Trạng thái ban đầu: Trong cơ sở dữ liệu chưa có database dds\_assgn1



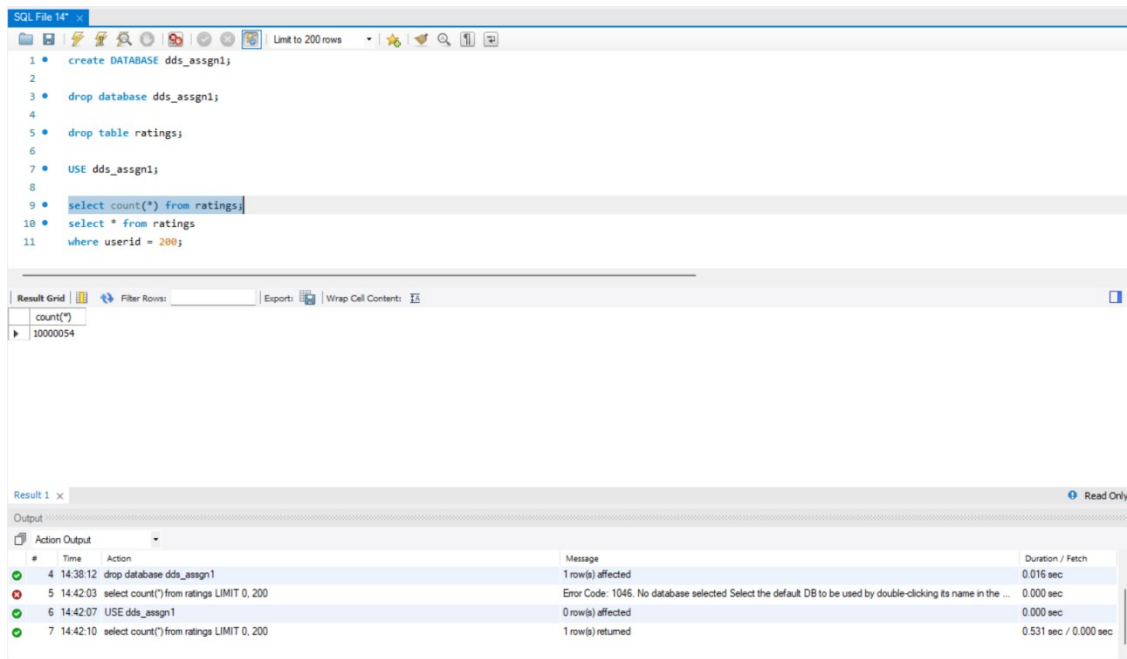
Hình 4 MySQL chưa có database dds\_assgn1

- Sau khi chạy hàm:

```
PS D:\BTL-CSDLPT-N17> & C:/Python313/python.exe d:/BTL-CSDLPT-N17/main.py
Database "dds_assgn1" created successfully
Đã thêm 1000000 bản ghi vào bảng ratings.
Đã thêm 2000000 bản ghi vào bảng ratings.
Đã thêm 3000000 bản ghi vào bảng ratings.
Đã thêm 4000000 bản ghi vào bảng ratings.
Đã thêm 5000000 bản ghi vào bảng ratings.
Đã thêm 6000000 bản ghi vào bảng ratings.
Đã thêm 7000000 bản ghi vào bảng ratings.
Đã thêm 8000000 bản ghi vào bảng ratings.
Đã thêm 9000000 bản ghi vào bảng ratings.
Đã thêm 10000000 bản ghi vào bảng ratings.
Đã thêm 10000054 bản ghi vào bảng ratings.
Tải dữ liệu thành công: 10000054 bản ghi đã thêm vào bảng ratings.
loadratings function pass!
```

Hình 5 Hàm loadRatings được khởi chạy

Kiểm thử pass testcase, 10.000.054 bản ghi đã thêm vào bảng ratings.



*Hình 6 Dữ liệu được thêm thành công vào MySQL*

- Bảng `ratings` chứa toàn bộ dữ liệu từ `ratings.dat` (10.000.054 bản ghi).

### 3.2 Hàm Range\_Partition()

Hàm `rangepartition()` được thiết kế để chia dữ liệu từ bảng `ratingstablename` thành N bảng phân vùng (`range_part0`, `range_part1`, ..., `range_part{N-1}`) dựa trên khoảng giá trị của cột `rating`.



```

59 def rangepartition(ratingtablename, numberofpartitions, openconnection):
60     """
61     Hàm để tạo các phân vùng của bảng chính dựa trên khoảng giá trị rating.
62     """
63     cur = openconnection.cursor()
64     delta = 5.0 / numberofpartitions
65     boundaries = [i * delta for i in range(numberofpartitions + 1)]
66     for i in range(numberofpartitions):
67         table_name = f"range_part{i}"
68         cur.execute(f"DROP TABLE IF EXISTS {table_name}")
69         cur.execute(f"""
70             CREATE TABLE {table_name} (
71                 userid INT,
72                 movieid INT,
73                 rating FLOAT
74             )
75         """)
76         lower_bound = boundaries[i]
77         upper_bound = boundaries[i + 1]
78         if i == 0:
79             cur.execute(f"""
80                 INSERT INTO {table_name}
81                 SELECT * FROM {ratingtablename}
82                 WHERE rating >= %s AND rating <= %s
83             """, (lower_bound, upper_bound))
84         else:
85             cur.execute(f"""
86                 INSERT INTO {table_name}
87                 SELECT * FROM {ratingtablename}
88                 WHERE rating > %s AND rating <= %s
89             """, (lower_bound, upper_bound))
90     openconnection.commit()
91     cur.close()

```

Hình 7 Hàm rangepartition()

### 3.2.1 Cách thức hoạt động

- Khởi tạo con trỏ:

cur = openconnection.cursor(): Mở một con trỏ SQL để thực thi các câu lệnh trên cơ sở dữ liệu, cho phép tương tác với database.

- Tính toán ranh giới phân vùng:

+delta = 5.0 / numberofpartitions:

- Tính khoảng cách giá trị rating giữa các phân vùng, với giả định thang điểm rating từ 0 đến 5.

+boundaries = [i \* delta for i in range(numberofpartitions + 1)]

- Tạo danh sách các ranh giới (boundaries) cho các phân vùng, ví dụ nếu numberofpartitions = 5, thì boundaries = [0, 1, 2, 3, 4, 5].

- Vòng lặp tạo bảng phân vùng:

+ for i in range(numberofpartitions): Lặp từ i = 0 đến i = numberofpartitions-1 để tạo từng bảng phân vùng.

+ table\_name = f'range\_part{i}': Tạo tên bảng phân vùng, ví dụ range\_part0, range\_part1, v.v.

+ cur.execute(f'DROP TABLE IF EXISTS {table\_name}'): Xóa bảng nếu đã tồn tại để tránh xung đột dữ liệu cũ.

-Tạo bảng mới:

+ cur.execute(f'CREATE TABLE {table\_name} (userid INT, movieid INT, rating FLOAT)'):

- Tạo bảng mới với schema: userid INT, movieid INT, rating FLOAT. Đây là cấu trúc giống bảng ratingtablename nhưng chỉ chứa các cột cần thiết (không bao gồm cột khác như Timestamp nếu có).

- Chèn dữ liệu theo khoảng giá trị:

+ lower\_bound = boundaries[i]

+ upper\_bound = boundaries[i + 1]

Xác định khoảng giá trị rating cho phân vùng hiện tại.

Nếu i == 0:

+ cur.execute(f'INSERT INTO {table\_name} SELECT \* FROM {ratingtablename} WHERE rating >= %s AND rating <= %s', (lower\_bound, upper\_bound))

Chèn dữ liệu từ bảng chính (ratingtablename) vào bảng phân vùng, với điều kiện rating nằm trong khoảng [lower\_bound, upper\_bound] (bao gồm cả giá trị lower\_bound).

Nếu i > 0:

+ cur.execute(f'INSERT INTO {table\_name} SELECT \* FROM {ratingtablename} WHERE rating > %s AND rating <= %s', (lower\_bound, upper\_bound))

- Chèn dữ liệu với điều kiện rating nằm trong khoảng (lower\_bound, upper\_bound] (không bao gồm lower\_bound).

- Sự khác biệt giữa >= và > đảm bảo các giá trị biên chỉ thuộc về một phân vùng, tránh trùng lặp dữ liệu.

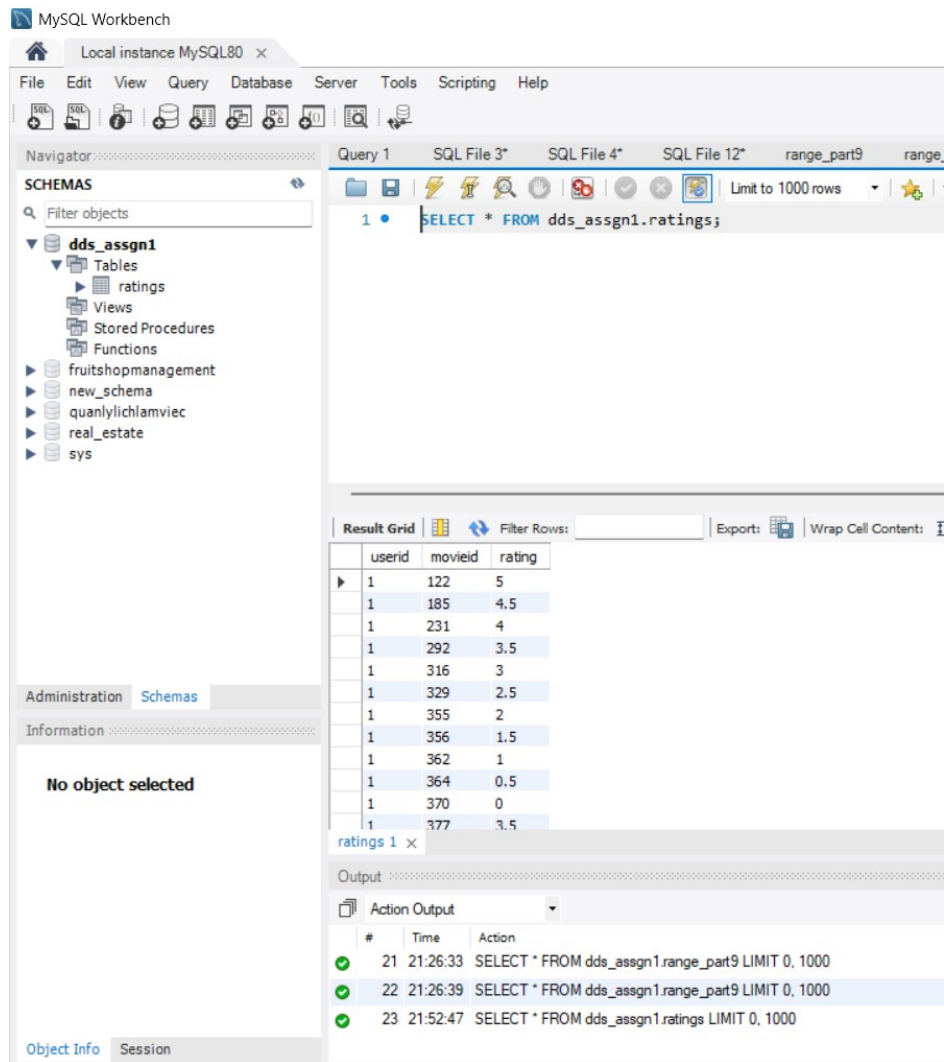
- Commit và đóng:

+ openconnection.commit(): Lưu tất cả các thay đổi (xóa bảng, tạo bảng, chèn dữ liệu) vào cơ sở dữ liệu.

+ cur.close(): Đóng con trỏ để giải phóng tài nguyên.

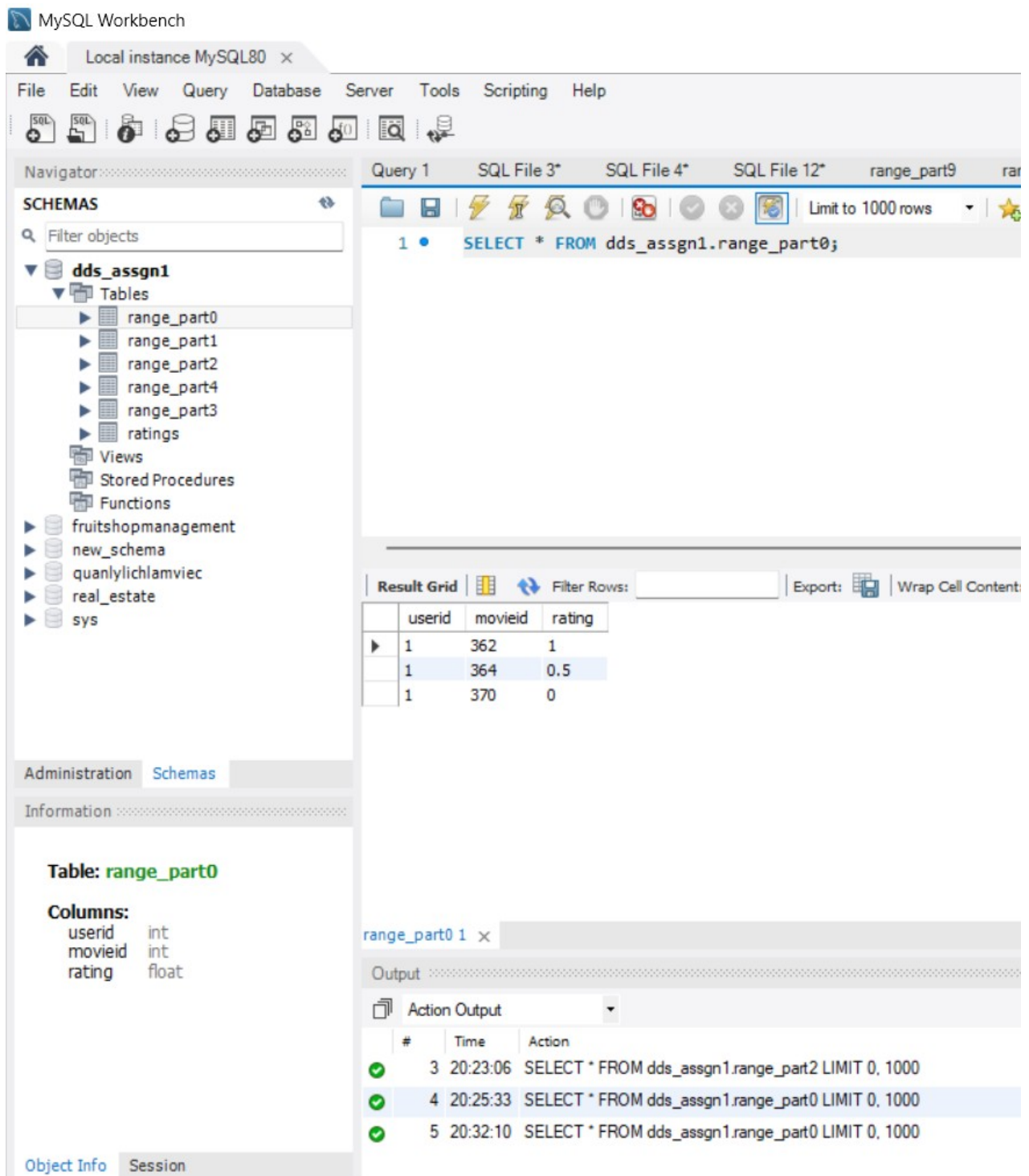
### **3.2.2 Sự thay đổi trong cơ sở dữ liệu**





Hình 8 Database khi chưa phân mảnh rangepartition

- Bảng 'ratings' chứa toàn bộ dữ liệu từ 'test\_data.dat' (20 bản ghi).
- Không có bảng nào có tiền tố 'range\_part'



Hình 9 Database khi đã phân mảnh rangepartition

- Tạo bảng phân mảnh: N bảng mới ('range\_part0' đến 'range\_part{N-1}') được tạo.
- Phân phối dữ liệu: Các bản ghi từ 'ratings' được chia cho bảng range\_part theo rating

Với N=5 và 20 bản ghi:

- 'range\_part0': Bản ghi có rating từ 0 đến 1.
- 'range\_part1': Bản ghi có rating lớn hơn 1 nhỏ hơn bằng 2.

- 'range\_part2': Bản ghi có rating lớn hơn 2 nhỏ hơn bằng 3.
- 'range\_part3': Bản ghi có rating lớn hơn 3 nhỏ hơn bằng 4.
- 'range\_part4': Bản ghi có rating lớn hơn 4 nhỏ hơn bằng 5.
- Bảng gốc 'ratings' không thay đổi, vẫn giữ nguyên 20 bản ghi.
- Kiểm thử pass test case

```
C:\Users\Tranq\OneDrive\Desktop\BTL-CSDLPT-N17>"C:/Program Files/
main.py
A database named "dds_assgn1" already exists
Đã thêm 20 dòng vào bảng ratings.
Tải dữ liệu thành công: 20 dòng được chèn vào bảng ratings.
loadratings function pass!
rangepartition function pass!
```

Hình 10 Hàm rangepartition pass test

### 3.3 Hàm RoundRobin\_Partition()

Hàm 'roundrobinpartition()' được thiết kế để chia dữ liệu từ bảng 'ratings' thành N bảng phân mảnh vòng tròn ('rrobin\_part0', 'rrobin\_part1', ..., 'rrobin\_part{N-1}') theo cách phân phối tuần hoàn dựa trên số thứ tự của bản ghi.

```
def roundrobinpartition(ratingtablename, numberofpartitions, openconnection):
    """
    Hàm để tạo các phân vùng của bảng chính theo phương pháp round-robin.
    """
    cur = openconnection.cursor()
    # Tạo bảng phân mảnh
    for i in range(numberofpartitions):
        table_name = f"rrobin_part{i}"
        cur.execute("DROP TABLE IF EXISTS {}".format(table_name))
        cur.execute("CREATE TABLE {} (userid INT, movieid INT, rating FLOAT)".format(table_name))
        cur.execute(f"""INSERT INTO {table_name} (userid, movieid, rating) SELECT userid, movieid, rating
        FROM (
            SELECT
                ROW_NUMBER() OVER () AS rnum,
                userid,
                movieid,
                rating
            FROM ratings
        ) AS temp
        WHERE (rnum-1) % {numberofpartitions} = {i}
        """)
        print(f"Đã tạo phân vùng {table_name} với round-robin.")
    openconnection.commit()
    cur.close()
```

Hình 11 Hàm roundrobinpartition()

#### 3.3.1 Cách thức hoạt động:

- Khởi tạo con trỏ:

cur = openconnection.cursor(): Mở một con trỏ SQL để thực hiện các câu lệnh trên cơ sở dữ liệu.

- Vòng lặp tạo bảng phân mảnh:

+ `for i in range(numberofpartitions):` Lặp từ  $i = 0$  đến  $i = N-1$  (với  $N$  là `numberofpartitions`).

+ `table_name = f'rrobin_part{i}'`: Tạo tên bảng phân mảnh, ví dụ `rrobin_part0`, `rrobin_part1`, v.v.

+ `cur.execute("DROP TABLE IF EXISTS {0}".format(table_name))`: Xóa bảng nếu đã tồn tại để tránh xung đột dữ liệu cũ.

- Tạo bảng mới:

+ `cur.execute("CREATE TABLE {0} (userid INT, movieid INT, rating FLOAT)".format(table_name))`

Tạo bảng mới với schema: `userid INT, movieid INT, rating FLOAT`. Đây là cấu trúc giống bảng `ratings`, nhưng chỉ chứa các cột được yêu cầu (không có `Timestamp`).

- Chèn dữ liệu theo vòng tròn:

+ `cur.execute("INSERT INTO {table_name} (userid, movieid, rating)`

`SELECT userid, movieid, rating`

`FROM (SELECT ROW_NUMBER() OVER () AS rnum, userid, movieid, rating FROM ratings) AS temp`

`WHERE (rnum-1) % {numberofpartitions} = {i}"))`

- Thông báo:

+ `print(f'Đã tạo phân vùng {table_name} với round-robin.')`: In thông báo xác nhận tạo thành công mỗi bảng.

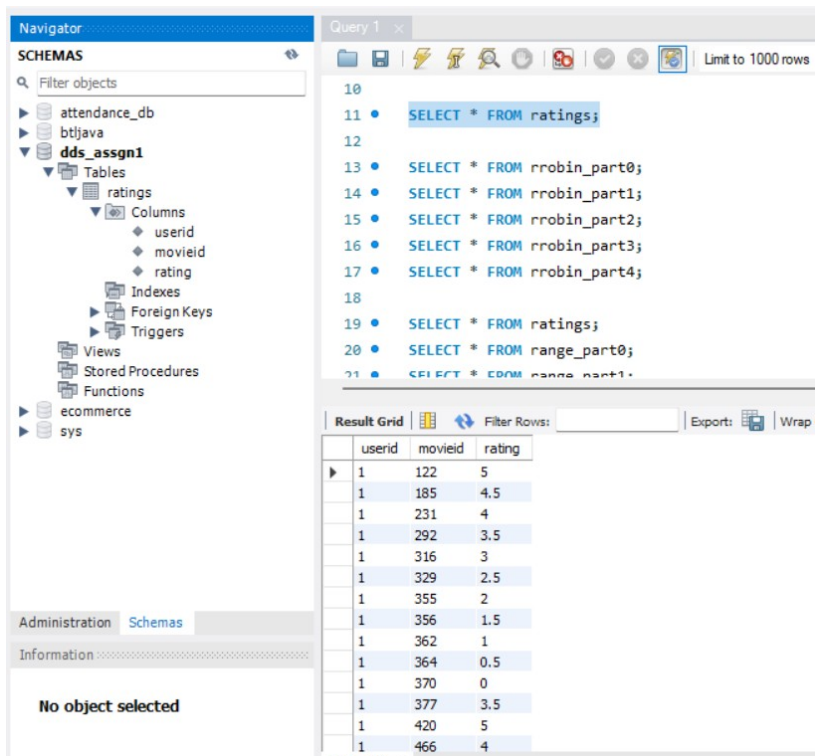
- Commit và đóng:

+ `openconnection.commit()`: Lưu các thay đổi vào cơ sở dữ liệu.

+ `cur.close()`: Đóng con trỏ để giải phóng tài nguyên.

### 3.3.2 Sự thay đổi trong cơ sở dữ liệu

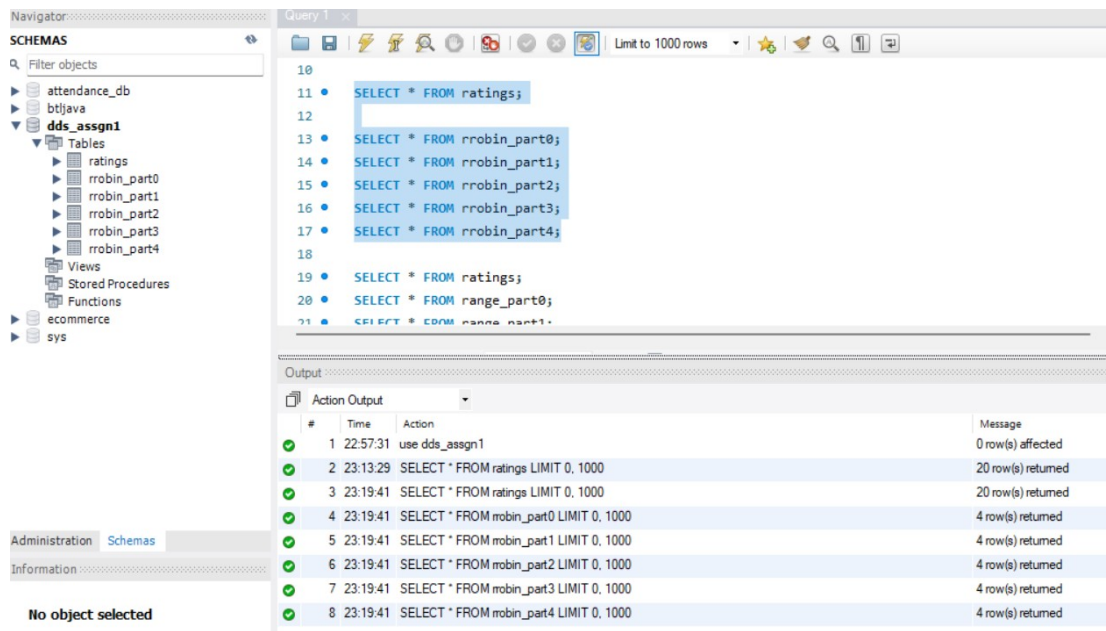
Trạng thái ban đầu:



Hình 12 Database khi chưa phân mảnh roundrobin

- Bảng 'ratings' chứa toàn bộ dữ liệu từ 'test\_data.dat' (20 bản ghi).
- Không có bảng nào có tiền tố 'rrobin\_part'.

Sau khi chạy hàm:



Hình 13 Database sau khi phân mảnh roundrobin

- Tạo bảng phân mảnh: N bảng mới ('rrobin\_part0' đến 'rrobin\_part{N-1}') được tạo.



- Phân phối dữ liệu: Các bản ghi từ `ratings` được chia đều:
- Với  $N = 5$  và 20 bản ghi:
  - + `rrobin\_part0`: Bản ghi 1, 6, 11, 16 (4 bản ghi).
  - + `rrobin\_part1`: Bản ghi 2, 7, 12, 17.
  - + `rrobin\_part2`: Bản ghi 3, 8, 13, 18.
  - + `rrobin\_part3`: Bản ghi 4, 9, 14, 19.
  - + `rrobin\_part4`: Bản ghi 5, 10, 15, 20.
- Mỗi bảng chứa khoảng 4 bản ghi.
- Bảng gốc `ratings` không thay đổi, vẫn giữ 20 bản ghi.

*Kiểm thử: Pass test case, 5 bảng được tạo, mỗi bảng chứa 4 bản ghi, tổng cộng 20 bản ghi, không trùng lặp.*

```
PS C:\Users\hienn\OneDrive\Máy tính\CSDL\BTL-CSDLPT-N17> python -
Đã tạo phân vùng rrobin_part0 với round-robin.
Đã tạo phân vùng rrobin_part1 với round-robin.
Đã tạo phân vùng rrobin_part2 với round-robin.
Đã tạo phân vùng rrobin_part3 với round-robin.
Đã tạo phân vùng rrobin_part4 với round-robin.
roundrobinpartition function pass!
Nhấn enter để xóa tất cả các bảng? █
```

Hình 14 Hàm `roundrobin_partition pass test`

### 3.4 Hàm `RoundRobin_Insert()`

Hàm `roundrobininsert()` được thiết kế để chèn một bản ghi mới (userid, itemid, rating) vào bảng `ratings` và đồng thời vào một trong các bảng phân mảnh vòng tròn (`rrobin\_part0`, `rrobin\_part1`, ..., `rrobin\_part{N-1}`) theo cách phân phối tuần hoàn. Điều này đảm bảo duy trì tính cân bằng của phân mảnh vòng tròn khi thêm dữ liệu mới.

```
def roundrobininsert(ratingtablename, userid, itemid, rating, openconnection):
    """
    Hàm để chèn một dòng mới vào bảng chính và phân vùng round-robin.
    """
    cur = openconnection.cursor()
    # Chèn vào bảng chính
    cur.execute("INSERT INTO {0} (userid, movieid, rating) VALUES ({1}, {2}, {3})".format(ratingtablename, userid, itemid, rating))
    # Tìm phân mảnh round-robin
    cur.execute(f"SELECT COUNT(*) FROM {ratingtablename}")
    total_rows = cur.fetchone()[0]
    cur.execute("SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = DATABASE() AND table_name LIKE 'rrobin_part%'")
    numberofpartitions = cur.fetchone()[0]
    table_name = f"rrobin_part{((total_rows - 1) % numberofpartitions)}"
    cur.execute(f"""
        INSERT INTO {table_name} (userid, movieid, rating)
        VALUES ({userid}, {itemid}, {rating})
    """, (userid, itemid, rating))
    openconnection.commit()
    cur.close()
```

Hình 15 Hàm `roundrobininsert()`

### 3.4.1 Cách thức hoạt động

- Khởi tạo con trỏ:

`cur = openconnection.cursor()`: Mở một con trỏ SQL để thực hiện các câu lệnh trên cơ sở dữ liệu.

- Chèn vào bảng chính:

+ `cur.execute("INSERT INTO {0} (userid, movieid, rating) VALUES ({1}, {2}, {3})".format(ratingtablename, userid, itemid, rating))`

Chèn bản ghi mới vào bảng `ratings` với các giá trị (userid, itemid, rating)

- Xác định phân mảnh đích:

+ `total_rows = cur.fetchone()[0] FROM (SELECT COUNT(*) FROM {ratingtablename})`

Đếm tổng số bản ghi hiện tại trong bảng `ratings` sau khi chèn.

+ `numberofpartitions = cur.fetchone()[0] FROM (SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = DATABASE() AND table_name LIKE 'rrobin_part%')`

Đếm số bảng phân mảnh vòng tròn (`rrobin\_part%`) để lấy N số phân mảnh.

+ `table_name = "rrobin_part" + str((total_rows - 1) % numberofpartitions)`

Tính chỉ số phân mảnh dựa trên công thức  $(total\_rows - 1) \bmod N$

(total\_rows - 1) là số thứ tự của bản ghi mới (vì bản ghi vừa chèn tăng số lượng lên 1). Kết quả là tên bảng phân mảnh, ví dụ `rrobin\_part0`, `rrobin\_part1`, v.v.

- Chèn vào bảng phân mảnh:

+ `cur.execute("INSERT INTO {table_name} (userid, movieid, rating) VALUES (%s, %s, %s)", (userid, itemid, rating))`

Chèn bản ghi vào bảng `table\_name` (ví dụ: `rrobin\_part0`) với tham số hóa (`%s`). Sử dụng tuple (userid, itemid, rating) để truyền giá trị.

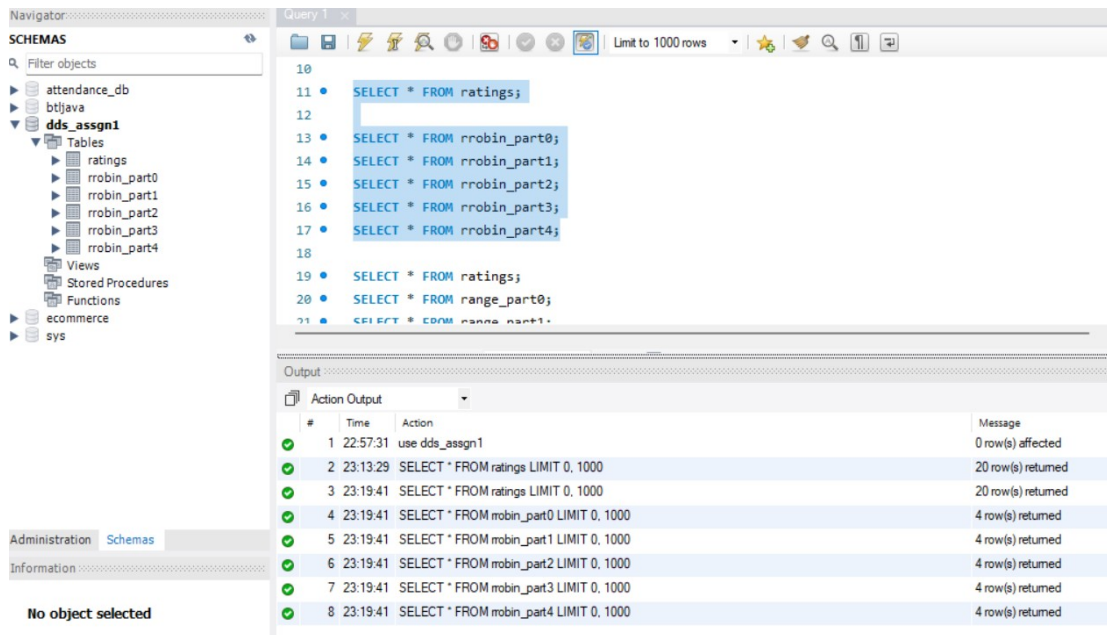
- Commit và đóng:

+ `openconnection.commit()`: Lưu các thay đổi vào cơ sở dữ liệu.

+ `cur.close()`: Đóng con trỏ để giải phóng tài nguyên.

### 3.4.2 Sự thay đổi trong cơ sở dữ liệu

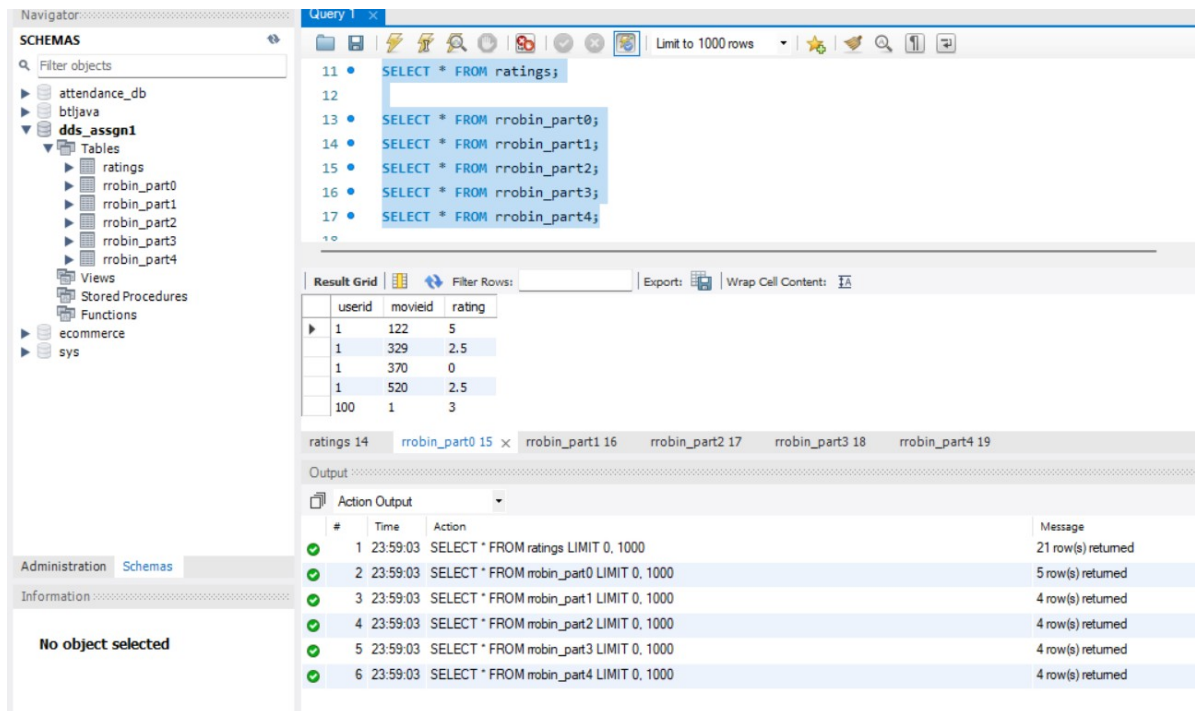
Trạng thái ban đầu:



Hình 16 Database khi chưa insert theo roundrobin

- Bảng 'ratings' chứa 20 bản ghi từ 'test\_data.dat'.
- 5 bảng phân mảnh 'rrobin\_part0' đến 'rrobin\_part4' đã được tạo bởi 'roundrobinpartition()', mỗi bảng có 4 bản ghi.

Sau khi chạy hàm với bản ghi mới (userid=100, itemid=1, rating=3):



Hình 17 Database sau khi insert theo roundrobin

- Bảng 'ratings':
- + Tăng lên 21 bản ghi, thêm bản ghi (100, 1, 3).



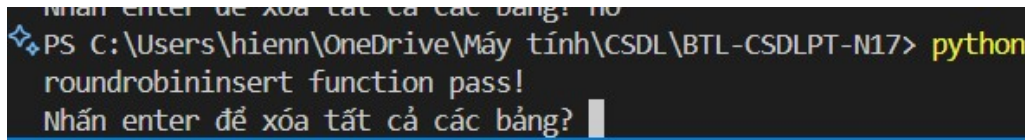
- Bảng phân mảnh:

+ Trước đó `total_rows = 20`, sau khi chèn `total_rows = 21`.

$(21 - 1) \bmod 5 = 20 \bmod 5 = 0$ , nên bản ghi được chèn vào ``rrobin_part0``.

+ ``rrobin_part0`` tăng lên 5 bản ghi, các bảng khác vẫn 4 bản ghi.

*Kiểm thử: Pass test case, bản ghi (100, 1, 3) xuất hiện trong ``rrobin_part0`` và ``ratings``.*



Hình 18 Hàm `roundrobin_insert` pass test

### 3.5 Hàm `Range_Insert()`

Hàm `rangeinsert(ratingtablename, userid, itemid, rating, openconnection)` được thiết kế để chèn một bản ghi mới (`userid, itemid, rating`) vào bảng chính (`ratingtablename`) và đồng thời vào một trong các bảng phân mảnh (`range_part0, range_part1, ..., range_part{N-1}`) dựa trên giá trị của `rating`. Điều này đảm bảo rằng các bản ghi được phân phối vào các bảng phân mảnh theo các khoảng giá trị của `rating`, duy trì tính tổ chức của phân mảnh theo phạm vi

```
138 def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
139     """
140     Hàm để chèn một dòng mới vào bảng chính và phân vùng dựa trên rating.
141     """
142     cur = openconnection.cursor()
143     # Chèn vào bảng chính
144     cur.execute(f"""
145         INSERT INTO {ratingtablename} (userid, movieid, rating)
146         VALUES (%s, %s, %s)
147     """, (userid, itemid, rating))
148     # Tìm phân mảnh dựa trên rating
149     cur.execute("SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = DATABASE() AND table_name LIKE 'range_part%'")
150     numberofpartitions = cur.fetchone()[0]
151     delta = 5.0 / numberofpartitions
152     index = min(int(rating / delta) - 1 if rating == int(rating) else int(rating / delta), numberofpartitions - 1)
153     if index < 0:
154         index = 0
155     table_name = f"range_part{index}"
156     cur.execute(f"""
157         INSERT INTO {table_name} (userid, movieid, rating)
158         VALUES (%s, %s, %s)
159     """, (userid, itemid, rating))
160     openconnection.commit()
161     cur.close()
```

Hình 19 Hàm `rangeinsert()`

#### 3.5.1 Cách thức hoạt động

- Khởi tạo con trỏ

+ `cur = openconnection.cursor()`

- Hàm mở một con trỏ SQL (`cur`) thông qua kết nối cơ sở dữ liệu (`openconnection`) để thực hiện các câu lệnh SQL.

- Chèn vào bảng chính:

```
cur.execute(f"INSERT INTO {ratingtablename} (userid, movieid, rating)
VALUES (%s, %s, %s)
", (userid, itemid, rating))
```

+Hàm sử dụng câu lệnh INSERT INTO để chèn một bản ghi mới vào bảng chính (được xác định bởi tham số ratingtablename).

+ Các cột userid, movieid, và rating được điền giá trị từ các tham số userid, itemid, và rating.

+ Sử dụng tham số hóa (%s) và truyền giá trị qua tuple (userid, itemid, rating) để tránh lỗi SQL injection và đảm bảo an toàn.

- Xác định phân mảnh đích:

```
cur.execute("SELECT COUNT(*) FROM information_schema.tables WHERE
table_schema = DATABASE() AND table_name LIKE 'range_part%'")
```

```
numberofpartitions = cur.fetchone()[0]
```

```
delta = 5.0 / numberofpartitions
```

```
index = min(int(rating / delta) - 1 if rating == int(rating) else int(rating / delta),
numberofpartitions - 1)
```

```
if index < 0:
```

```
    index = 0
```

```
table_name = f"range_part{index}"
```

- Đếm số bảng phân mảnh:

+ Hàm thực hiện truy vấn để đếm số bảng phân mảnh có tên bắt đầu bằng range\_part trong cơ sở dữ liệu hiện tại (sử dụng information\_schema.tables).

+ Kết quả được lưu vào numberofpartitions.

- Tính khoảng giá trị (delta):

+ Thang điểm rating nằm trong khoảng [0, 5], hàm chia đều khoảng này thành numberofpartitions phần, mỗi phần có độ dài delta = 5.0 / numberofpartitions.

- Xác định chỉ số phân mảnh (index):

+ Chỉ số phân mảnh được tính dựa trên giá trị rating:

+ Nếu rating là số nguyên, sử dụng int(rating / delta) - 1 để xác định phân mảnh.

+ Nếu rating không phải số nguyên, sử dụng int(rating / delta).

+ Kết quả index được giới hạn tối đa bằng numberofpartitions - 1 để đảm bảo không vượt quá số phân mảnh.

+ Nếu index nhỏ hơn 0 (trường hợp rating rất nhỏ), đặt index = 0.

- Xác định tên bảng phân mảnh:

+ Tên bảng phân mảnh được tạo bằng cách nối chuỗi "range\_part" với index, ví dụ: range\_part0, range\_part1, v.v.

- Chèn vào bảng phân mảnh

```
cur.execute(f"""
```

```
    INSERT INTO {table_name} (userid, movieid, rating)
```

```
    VALUES (%s, %s, %s)
```

```
""", (userid, itemid, rating))
```

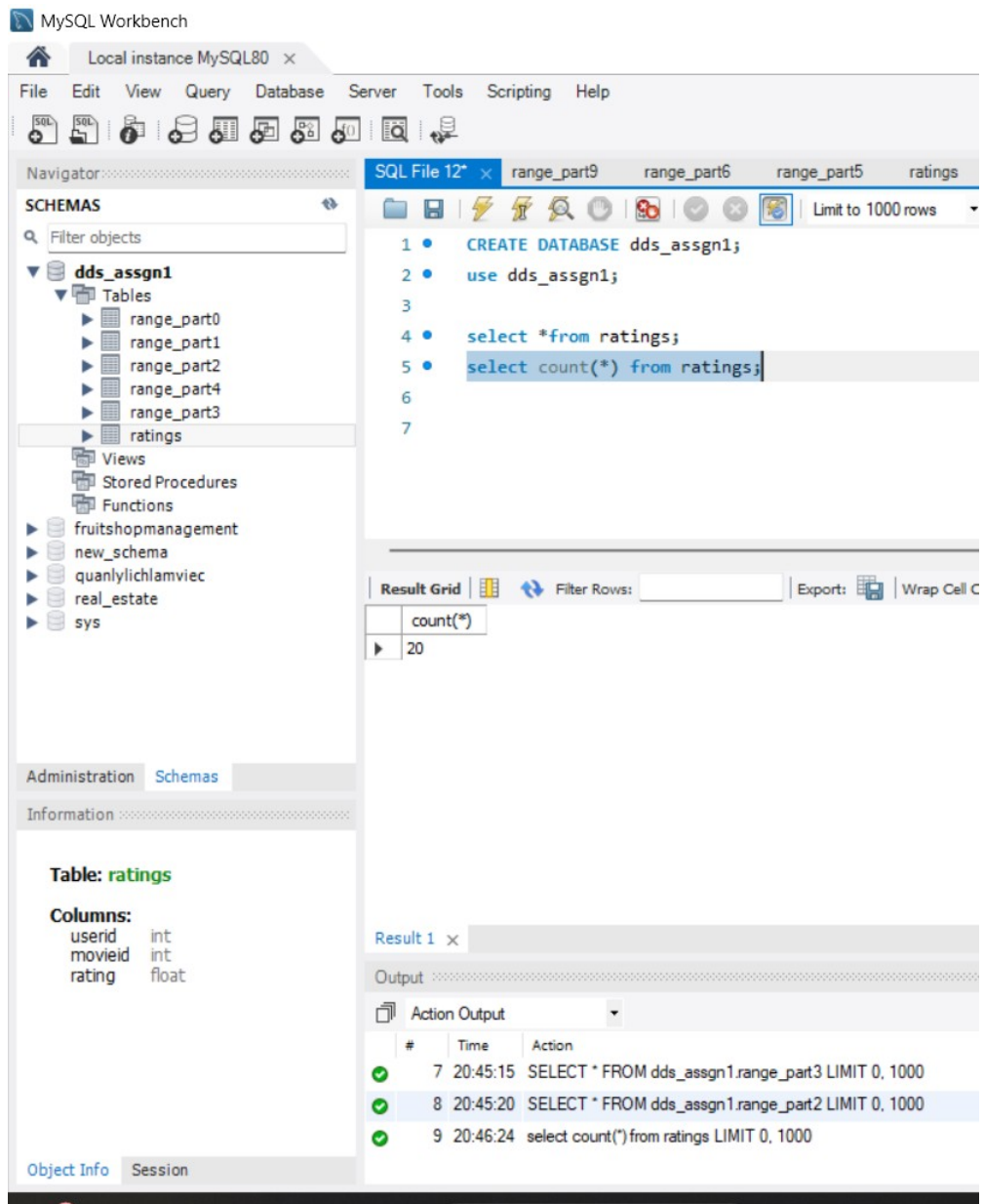
+ Hàm chèn bản ghi vào bảng phân mảnh (table\_name) tương ứng với giá trị rating.

+ Tương tự bảng chính, câu lệnh sử dụng tham số hóa (%s) và tuple (userid, itemid, rating) để truyền giá trị an toàn.

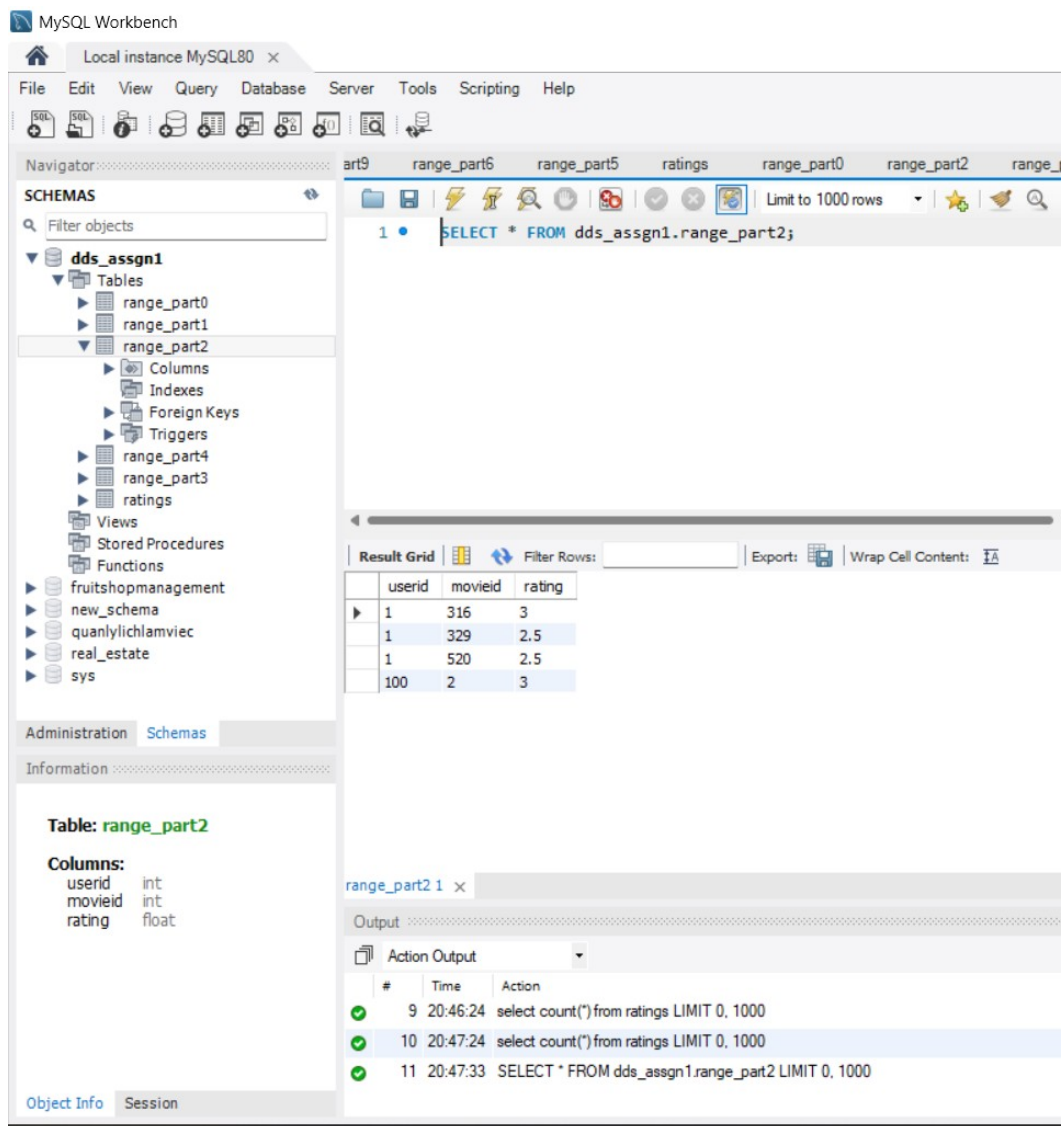
- Commit và đóng:

+ openconnection.commit(): Lưu tất cả các thay đổi (bao gồm chèn vào bảng chính và bảng phân mảnh) vào cơ sở dữ liệu.

+ cur.close(): Đóng con trỏ để giải phóng tài nguyên.



Hình 20 Database khi chưa insert theo rangepartition



Hình 21 Database khi đã insert theo rangepartition

- Bảng 'ratings':

+ Tăng lên 21 bản ghi, thêm bản ghi (100, 1, 3).

- Bảng phân mảnh:

Bảng range\_part2 đã được chèn thêm bản ghi có rating là 3.

- Kiểm thử: Pass test case, bản ghi (100, 1, 3) xuất hiện trong 'range\_part2' và 'ratings'.

```
C:\Users\Tranq\OneDrive\Desktop\BTL-CSDLPT-N17>"C:/Program Files/Python39/python.exe" main.py
A database named "dds_assgn1" already exists
Đã thêm 20 dòng vào bảng ratings.
Tải dữ liệu thành công: 20 dòng được chèn vào bảng ratings.
loadratings function pass!
rangepartition function pass!
rangeinsert function pass!
```

Hình 22 Hàm rangeinsert pass test

## KẾT LUẬN

Qua quá trình thực hiện bài tập lớn, nhóm chúng em đã có cơ hội tìm hiểu và áp dụng các phương pháp phân mảnh dữ liệu trong hệ quản trị cơ sở dữ liệu quan hệ, cụ thể là phân mảnh ngang theo Range và Round Robin. Việc hiện thực bài toán trên hệ quản trị MySQL (hoặc PostgreSQL) cùng với ngôn ngữ lập trình Python đã giúp chúng em hiểu rõ hơn về quy trình tổ chức và tối ưu hóa việc lưu trữ dữ liệu lớn.

Thông qua việc triển khai các hàm LoadRatings, Range\_Partition, RoundRobin\_Partition, cũng như các hàm chèn dữ liệu Range\_Insert và RoundRobin\_Insert, chúng em đã học được cách xử lý tệp dữ liệu lớn, cách phân chia dữ liệu một cách hiệu quả và đảm bảo tính toàn vẹn trong quá trình chèn dữ liệu mới.

Dù còn một số khó khăn nhất định trong quá trình cài đặt, xử lý lỗi và kiểm thử, nhóm đã cố gắng hoàn thành đầy đủ yêu cầu của bài tập lớn, đồng thời rút ra được nhiều kinh nghiệm quý báu trong việc phát triển và kiểm thử phần mềm liên quan đến cơ sở dữ liệu.