BỘ KHOA HỌC VÀ CÔNG NGHỆ
HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA VIỄN THÔNG 1

BTL CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên: Kim Ngọc Bách
Nhóm học phần : 10
Nhóm BTL : 02
Sinh viên: Bùi Xuân Trường -B22DCCN878
Phan Hữu Dương
Nguyễn Văn Đạt

– Hà Nội năm 2025 –

LỜI CẢM ƠN

Lời đầu tiên, nhóm 2 chúng em muốn gửi lời cảm ơn chân thành và sâu sắc tới thầy Kim Ngọc Bách, người đã chỉ dạy và hướng dẫn nhóm 3 nói riêng, cũng như nhóm học phần 10 nói chung trong suốt quá trình thầy giảng dạy học phần Cơ sở dữ liệu phân tán.

Nhờ sự chỉ bảo tận tâm, phương pháp giảng dạy dễ hiểu, cùng với những kiến thức thực tiễn mà thầy truyền đạt, nhóm đã có cơ hội tiếp cận và hiểu rõ hơn về các khái niệm cốt lõi, cũng như kỹ thuật triển khai trong hệ thống cơ sở dữ liệu phân tán – một lĩnh vực quan trọng và thiết thực trong ngành Công nghệ Thông tin.

Thực hiện bài tập lớn này, nhóm chúng em đã được thầy tạo điều kiện để tìm hiểu sâu hơn, rõ hơn về hai loại phân mảnh, biến những lý thuyết tưởng chừng như khô khan, vào trong một bài toán thực tế. Xoay quanh quá trình thực hiện bài tập lớn này, có không ít những khó khăn mà nhóm chưa thể giải quyết triệt để, nhưng nhờ có sự giải đáp nhiệt tình, tâm huyết của thầy, đã phần nào đó giúp chúng em tiến bộ hơn.

Dẫu vậy thì có thể bài tập lớn vẫn chưa đạt tới độ hoàn thiện mà thầy mong muốn, nhưng đó cũng là đúc kết từ sự cố gắng, nỗ lực hoàn thành của nhóm, nên có thể phần nào mong thầy nhìn nhận bài tập lớn này dưới góc độ tích cực nhất.

Một lần nữa, nhóm 2 xin chân thành cảm ơn thầy, kính chúc thầy sức khỏe và luôn thành công trong sự nghiệp giảng dạy!

Nhóm 2

I. Chuẩn bị cấu hình

1. Môi trường thực thi

– Hệ điều hành: Chuẩn bị một hệ điều hành Ubuntu hoặc Windows 10.
– Phiên bản Python: Cài đặt Python phiên bản 3.12.x hoặc cao hơn.
– Hệ quản trị cơ sở dữ liệu: Cài đặt và cấu hình một trong các hệ quản trị cơsở dữ liệu quan hệ mã nguồn mở được hỗ trợ: PostgreSQL hoặc MySQL. Đảm bảo rằng cơ sở dữ liệu có thể được truy cập và quản lý thông qua các thư viện Python tương ứng.

2. Dữ liệu đầu vào

– Nguồn dữ liệu: Tải tệp ratings.dat từ trang web MovieLens thông qua liên kết sau: http://files.grouplens.org/datasets/movielens/ml-10m.zip.
– Mô tả dữ liệu: Tệp ratings.dat chứa 10 triệu đánh giá và 100.000 thẻ áp dụng cho 10.000 bộ phim bởi 72.000 người dùng.
– Định dạng mỗi dòng: Mỗi dòng trong tệp đại diện cho một đánh giá và tuân theo định dạng sau:
– UserID :: MovieID :: Rating :: Timestamp
+ UserID: Định danh người dùng (số nguyên).
+ MovieID: Định danh bộ phim (số nguyên).
+ Rating: Điểm đánh giá (số thực, thang điểm 5 sao, có thể nửa sao).
+ Timestamp: Thời điểm đánh giá (tính từ ngày 1 tháng 1 năm 1970).

3. Cấu trúc bảng RATINGS

– Để lưu trữ dữ liệu đánh giá, một bảng có tên Ratings sẽ được tạo trong cơ sở dữ liệu (PostgreSQL hoặc MySQL) với lược đồ (schema) sau:
+ UserID (kiểu dữ liệu: INT)
+ MovieID (kiểu dữ liệu: INT)
+ Rating (kiểu dữ liệu: FLOAT hoặc REAL)

II. Thực hiện

1. Hàm Loadratings

– **Mục đích** : Cài đặt hàm Python LoadRatings() nhận vào một đường dẫn tuyệt đối đến tệp rating.dat.
LoadRatings() sẽ tải nội dung tệp vào một bảng trong PostgreSQL có tên Ratings với schema sau: UserID (int), MovieID (int), Rating (float)
– Trên mỗi dòng của file ratings.dat, bộ dữ liệu có định dạng như sau: UserID::MovieID::Rating::Timestamp
– **Thực hiện** :
+ Sử dụng đối tượng openconnection để tạo con trỏ (cursor) cho các lệnh SQL.

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

+ Xóa đi bảng cũ và tạo bảng mới với 7 cột ứng với cấu trúc dữ liệu file đầu vào. Tạo bảng với 7 cột để khớp với định dạng file ratings.dat, giả định file có cấu trúc: userid:extra1:movieid:extra2:rating:extra3:timestamp.
+ Các cột extra1, extra2, extra3 (CHAR) là các cột tạm để chứa dấu : (phân cách trong file).userid và movieid là INTEGER, rating là FLOAT, timestamp là BIGINT, phù hợp với dữ liệu xếp hạng.

```
# Tạo bảng với cấu trúc giống dữ liệu trong rating.dat
cur.execute(f"""
DROP TABLE IF EXISTS {ratingstablename} CASCADE;

CREATE TABLE {ratingstablename} (
    userid INTEGER,
    extra1 CHAR,
    movieid INTEGER,
    extra2 CHAR,
    rating FLOAT,
    extra3 CHAR,
    timestamp BIGINT
)
""")
```

+ Chèn dữ liệu từ tệp vào

```
# Copy data từ file
cur.copy_from(open(ratingsfilepath, 'r'), ratingstablename, sep=':')
```

+ Tạo bảng với 7 cột để khớp với định dạng file ratings.dat, giả định file có cấu trúc: userid:extra1:movieid:extra2:rating:extra3:timestamp. Các cột extra1, extra2, extra3 (CHAR) là các cột tạm để chứa dấu : (phân cách trong file). userid và movieid là INTEGER, rating là FLOAT, timestamp là BIGINT, phù hợp với dữ liệu xếp hạng.
+ Commit giao dịch: Gọi con.commit() để lưu tất cả thay đổi (tạo bảng, nhập dữ liệu, xóa cột) vào cơ sở dữ liệu.
+ Đóng con trỏ: Gọi cur.close() để giải phóng tài nguyên, tránh rò rỉ bộ nhớ.

+

```
    # Drop các cột :
    cur.execute(f"""
    ALTER TABLE {ratingstablename}
    DROP COLUMN extra1,
    DROP COLUMN extra2,
    DROP COLUMN extra3,
    DROP COLUMN timestamp
    """)

    con.commit()
    cur.close()
```

2. Hàm rangepatition

– **Mục đích** : Cài đặt hàm Python Range_Partition() nhận vào: bảng Ratings trong PostgreSQL (hoặc MySQL) và một số nguyên N là số phân mảnh cần tạo Range_Partition() sẽ tạo N phân mảnh ngang của bảng Ratings và lưu vào PostgreSQL. Thuật toán sẽ phân chia dựa trên N khoảng giá trị đồng đều của thuộc tính Rating.

– Phương pháp này chia dữ liệu dựa trên giá trị của cột rating, đảm bảo các bản ghi được phân phối vào các bảng con (partitions) dựa trên các khoảng giá trị định trước.

– **Thực hiện** :

+ Hàm sử dụng openconnection để tạo một con trỏ (cursor) thông qua con.cursor().

+ Con trỏ này cho phép thực thi các lệnh SQL và quản lý giao dịch trong PostgreSQL.

+ Khối try-except-finally được sử dụng để xử lý lỗi và đảm bảo đóng con trỏ, tránh rò rỉ tài nguyên.

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

+ Xóa các bảng con cũ nhằm không gây xung đột trước khi tạo mới

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

range_size = 5.0 / numberofpartitions
```

+ Tính toán khoảng phân mảnh:
    - Biến range_size = 5.0 / numberofpartitions tính kích thước mỗi khoảng giá trị rating.
    - Giả định: Cột rating có giá trị từ 0 đến 5 (thang điểm xếp hạng tiêu chuẩn).
    - Ví dụ: Nếu numberofpartitions = 5, thì range_size = 1.0. Các khoảng sẽ là:
    - range_part0: $0.0 \leq \text{rating} \leq 1.0$
    - range_part1: $1.0 < \text{rating} \leq 2.0$
    - ...
    - range_part4: $4.0 < \text{rating} \leq 5.0$
+ Vòng lặp tạo numberofpartitions bảng con với tên range_part{i}.
    - Mỗi bảng con có cấu trúc giống bảng chính:
    - userid (INTEGER): ID người dùng.
    - movieid (INTEGER): ID phim.
    - rating (FLOAT): Điểm xếp hạng.
    - Sử dụng f-string để chèn tên bảng động, đảm bảo tính linh hoạt

```
range_size = 5.0 / numberofpartitions

for i in range(numberofpartitions):
    table_name = f"{RANGE_TABLE_PREFIX}{i}"
    cur.execute(f"""
    CREATE TABLE {table_name} (
        userid INTEGER,
        movieid INTEGER,
        rating FLOAT
    );
    """)
```

+ Vòng lặp chạy qua từng partition (i từ 0 đến numberofpartitions-1):
+ Gán table_name = range_part{i}.
+ Tính min_rating và max_rating cho khoảng hiện tại.
+ Nếu i == 0, sử dụng điều kiện rating >= {min_rating} AND rating <= {max_rating} để bao gồm giá trị rating = 0.0.
+ Nếu i > 0, sử dụng rating > {min_rating} AND rating <= {max_rating} để tránh trùng lặp tại ranh giới (ví dụ: rating = 1.25 chỉ thuộc một partition).

+ Lệnh SELECT lấy các bản ghi từ bảng chính (ratingstablename) thỏa mãn điều kiện khoảng rating.
+ Lệnh INSERT chèn các bản ghi này vào bảng con tương ứng (range_part{i}).

```
for i in range(numberofpartitions):
    table_name = f"{RANGE_TABLE_PREFIX}{i}"
    min_rating = i * range_size
    max_rating = min_rating +range_size

    if i == 0:
        cur.execute(f"""
        INSERT INTO {table_name} (userid, movieid, rating)
        SELECT userid, movieid, rating
        FROM {ratingstablename}
        WHERE rating >= {min_rating} AND rating <= {max_rating}
        """)
    else:
        cur.execute(f"""
        INSERT INTO {table_name} (userid, movieid, rating)
        SELECT userid, movieid, rating
        FROM {ratingstablename}
        WHERE rating > {min_rating} AND rating <= {max_rating}
        """)
```

+ Commit: Gọi con.commit() để lưu tất cả thay đổi (xóa bảng, tạo bảng, chèn dữ liệu) vào cơ sở dữ liệu.
+ Rollback: Nếu có lỗi (ví dụ: bảng chính không tồn tại, lỗi cú pháp SQL), con.rollback() hủy mọi thay đổi để đảm bảo tính toàn vẹn.
+ Đóng con trỏ: cur.close() trong khối finally giải phóng tài nguyên, tránh rò rỉ bộ nhớ.

```
        con.commit()

    except Exception:
        con.rollback()
    finally:
        cur.close()
```

3. Hàm roundrobinpartition

- **Mục đích** : Hàm roundrobinpartition được thiết kế để thực hiện phân mảnh ngang (horizontal partitioning) theo phương pháp Round-Robin trong hệ thống cơ sở dữ liệu phân tán. Hàm này phân phối dữ liệu từ bảng chính (ratingstablename) vào các bảng con một cách tuần tự, đảm bảo số bản ghi được chia đều giữa các bảng con. Mục tiêu chính bao gồm:
  + Phân phối dữ liệu đồng đều để tối ưu hóa hiệu suất truy vấn và lưu trữ trong môi trường phân tán.
  + Tạo bảng đếm (round_robin_counter) để hỗ trợ chèn dữ liệu mới nhất quán với phương pháp Round-Robin.
  + Đảm bảo tính toàn vẹn và an toàn dữ liệu thông qua quản lý giao dịch.
- **Thực hiện** :

+ Sử dụng openconnection để lấy đối tượng kết nối PostgreSQL.
+ Tạo con trỏ (cur) để thực thi các lệnh SQL và quản lý giao dịch.

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

+ Xóa đi các bảng con cũ để tránh xung đột

```
# Xóa các bảng phân mảnh cũ
for i in range(numberofpartitions):
    cur.execute(f"DROP TABLE IF EXISTS {RROBIN_TABLE_PREFIX}{i} CASCADE;")
```

+ Tạo các bảng phân mảnh mới

```
# Tạo các bảng phân mảnh mới
for i in range(numberofpartitions):
    table_name = f"{RROBIN_TABLE_PREFIX}{i}"
    cur.execute(f"""
    CREATE TABLE {table_name} (
        userid INTEGER,
        movieid INTEGER,
        rating FLOAT
    );
    """)
```

+ Phân mảnh dữ liệu theo RoundRobin:
+ Vòng lặp chạy qua từng partition (i từ 0 đến numberofpartitions-1).
+ Sử dụng ROW_NUMBER() OVER() để gán số thứ tự (bắt đầu từ 1) cho mỗi bản ghi trong bảng chính (ratingstablename).

+ Tạo bảng tạm temp chứa các bản ghi với cột rnum (số thứ tự).
+ Điều kiện MOD(temp.rnum - 1, numberofpartitions) = i xác định bản ghi nào thuộc partition i:
    - rnum - 1 chuyển số thứ tự từ 1-based sang 0-based.
    - MOD(rnum - 1, numberofpartitions) tính chỉ số partition (0, 1, ..., numberofpartitions-1).
    - Ví dụ: Nếu numberofpartitions = 3, các bản ghi được phân phối:
        - Bản ghi 1 (rnum=1): (1-1) % 3 = 0 → rrobin_part0
        - Bản ghi 2 (rnum=2): (2-1) % 3 = 1 → rrobin_part1
        - Bản ghi 3 (rnum=3): (3-1) % 3 = 2 → rrobin_part2
        - Bản ghi 4 (rnum=4): (4-1) % 3 = 0 → rrobin_part0
+ Lệnh INSERT chèn các bản ghi thỏa mãn điều kiện vào bảng con tương ứng.

```
# Phân mảnh dữ liệu
for i in range(numberofpartitions):
    cur.execute(f"""
    INSERT INTO {RROBIN_TABLE_PREFIX}{i} (userid, movieid, rating)
    SELECT userid, movieid, rating
    FROM (
        SELECT *, ROW_NUMBER() OVER() as rnum
        FROM {ratingstablename}
    ) as temp
    WHERE MOD(temp.rnum - 1, {numberofpartitions}) = {i}
    """)
```

+ Thực thi lệnh SQL để đếm số bản ghi trong bảng chính.
+ Lưu kết quả vào biến total_records để sử dụng trong bước tạo bảng counter.

```
    # Đếm tổng số bản ghi trong bảng gốc
    cur.execute(f"SELECT COUNT(*) FROM {ratingstablename}")
    total_records = cur.fetchone()[0]
```

+ Tạo và cập nhật bảng counter:

- Tạo bảng round_robin_counter (nếu chưa tồn tại) với cột counter (BIGINT, mặc định 0).
- Xóa dữ liệu cũ trong round_robin_counter (nếu có) để đảm bảo trạng thái sạch.
- Chèn giá trị total_records vào cột counter, dùng để theo dõi số bản ghi đã phân mảnh.
- Mục đích: Hỗ trợ hàm roundrobininsert xác định bảng con tiếp theo khi chèn bản ghi mới.

```
    # Tạo bảng counter và đặt giá trị bằng số bản ghi đã phân mảnh
    cur.execute("""
    CREATE TABLE IF NOT EXISTS round_robin_counter (
        counter BIGINT DEFAULT 0
    );
    """)
    cur.execute("DELETE FROM round_robin_counter;")
    cur.execute("INSERT INTO round_robin_counter (counter) VALUES (%s);", (total_records,))
```

+ Commit: Gọi con.commit() để lưu tất cả thay đổi (xóa bảng, tạo bảng, chèn dữ liệu, tạo counter).
+ Rollback: Nếu có lỗi (ví dụ: bảng chính không tồn tại, lỗi cú pháp SQL), con.rollback() hủy mọi thay đổi để đảm bảo tính toàn vẹn dữ liệu.
+ Đóng con trỏ: cur.close() trong khối finally giải phóng tài nguyên, ngay cả khi có lỗi.

```
        con.commit()
            Ctrl+L to chat, Ctrl+K to
    except Exception:
        con.rollback()
    finally:
        cur.close()
```

4. Hàm roundrobininsert

– **Mục đích** : Hàm roundrobininsert được thiết kế để chèn một bản ghi mới vào bảng chính (ratingstablename) và bảng con phân mảnh Round-Robin tương ứng, duy trì tính nhất quán với phương pháp phân phối tuần tự được thiết lập bởi hàm roundrobinpartition. Hàm sử dụng bảng round_robin_counter để xác định bảng con tiếp theo, đảm bảo phân phối dữ liệu đồng đều trong hệ thống cơ sở dữ liệu phân tán. Mục tiêu chính là:

  + Chèn bản ghi mới vào đúng bảng con theo thứ tự Round-Robin
  + Cập nhật bộ đếm để duy trì tính tuần tự cho các lần chèn tiếp theo.
  + Đảm bảo an toàn giao dịch và xử lý lỗi hiệu quả.

– **Thực hiện** :

+ Khởi tạo kết nối và con trỏ

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

+ Kiểm tra sự tồn tại của bảng gốc

```
# Kiểm tra bảng gốc có tồn tại không
cur.execute("""
    SELECT COUNT(*)
    FROM information_schema.tables
    WHERE table_name = %s
""", (ratingstablename,))
if cur.fetchone()[0] == 0:
    raise Exception(f"Bảng {ratingstablename} không tồn tại")
```

+ Tạo bảng counter nếu chưa tồn tại. Bảng round_robin_counter lưu số bản ghi đã phân mảnh, dùng để xác định bảng con tiếp theo. Khởi tạo counter đảm bảo hàm hoạt động ngay cả khi chưa chạy roundrobinpartition.

```
# Tạo bảng counter nếu chưa tồn tại
cur.execute("""
CREATE TABLE IF NOT EXISTS round_robin_counter (
    counter BIGINT DEFAULT 0
);
INSERT INTO round_robin_counter (counter)
SELECT 0
WHERE NOT EXISTS (SELECT 1 FROM round_robin_counter);
""")
```

+ Xác định số lượng partition để tính toán chỉ số bảng con, đảm bảo chèn đúng mục tiêu.

```
    # Xác định số lượng partition
    cur.execute("""
        SELECT COUNT(*)
        FROM information_schema.tables
        WHERE table_name LIKE %s
    """, (RROBIN_TABLE_PREFIX + '%',))
    numberofpartitions = cur.fetchone()[0]

    if numberofpartitions == 0:
        raise Exception("Không tìm thấy bảng phân mảnh Round Robin")
```

+ Thực thi lệnh INSERT để chèn bản ghi mới vào bảng chính (ratingstablename).
+ Các cột đích là userid, movieid, rating, ánh xạ với các giá trị đầu vào:
+ userid → userid.
+ itemid → movieid (lưu ý: tên tham số itemid nhưng cột đích là movieid).
+ rating → rating.
+ Sử dụng tham số hóa (%s) để truyền giá trị an toàn, ngăn chặn SQL injection.
+ Ý nghĩa: Đảm bảo bản ghi được lưu vào bảng chính trước khi chèn vào bảng con, duy trì tính toàn vẹn dữ liệu.

```
# Chèn vào bảng gốc
cur.execute(f"""
INSERT INTO {ratingstablename} (userid, movieid, rating)
VALUES (%s, %s, %s)
""", (userid, itemid, rating))
```

+ Lấy và tính chỉ số partition: Xác định chính xác bảng con tiếp theo để chèn dữ liệu, duy trì thứ tự phân phối Round-Robin.
+ Khóa FOR UPDATE đảm bảo không có xung đột khi nhiều giao dịch chèn đồng thời.

```
# Lấy và tăng counter
cur.execute("SELECT counter FROM round_robin_counter FOR UPDATE")
counter = cur.fetchone()[0]
partition_index = counter % numberofpartitions
table_name = f"{RROBIN_TABLE_PREFIX}{partition_index}"
```

+ Chèn bản ghi vào partition: Bản ghi được lưu vào đúng bảng con theo phương pháp Round-Robin, đảm bảo phân mảnh dữ liệu nhất quán.
+ Giữ đồng bộ dữ liệu giữa bảng chính và bảng con.

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

+ Tăng giá trị counter : Đảm bảo lần chèn tiếp theo sẽ chọn bảng con kế tiếp theo thứ tự Round-Robin. Duy trì trạng thái phân phối tuần tự cho toàn bộ hệ thống.

```
# Tăng counter
cur.execute("UPDATE round_robin_counter SET counter = counter + 1")
```

5. Hàm rangeinsert

– **Mục đích** : Hàm rangeinsert được thiết kế để chèn một bản ghi mới vào bảng chính (ratingstablename) và bảng con phân mảnh theo phương pháp Range trong hệ thống cơ sở dữ liệu phân tán. Phương pháp Range phân mảnh dữ liệu dựa trên phạm vi giá trị của cột rating, chia khoảng giá trị [0, 5] thành các phân đoạn đều nhau, mỗi phân đoạn tương ứng với một bảng con (range_partX). Hàm đảm bảo bản ghi được chèn vào bảng con phù hợp dựa trên giá trị rating, duy trì tính nhất quán với cấu trúc phân mảnh được thiết lập bởi hàm rangepartition (giả định). Mục tiêu cụ thể:

+ Chèn bản ghi mới vào bảng chính và bảng con Range tương ứng.
+ Xác định bảng con dựa trên phạm vi giá trị của rating mà không cần bảng counter (khác với roundrobininsert).
+ Đảm bảo an toàn giao dịch và xử lý lỗi hiệu quả.
+ Hỗ trợ quản lý dữ liệu lớn trong hệ thống phân tán với phân mảnh dựa trên giá trị.

– **Thực hiện** :

+ Khởi tạo và kết nối con trỏ

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

+ Đếm số partition : Xác định số lượng partition để chia khoảng giá trị rating [0, 5] thành các phân đoạn đều nhau. Đảm bảo hàm biết cấu trúc phân mảnh hiện tại để ánh xạ rating vào đúng bảng con.

  - Gọi hàm count_partitions (giả định được định nghĩa ngoài hàm) với hai tham số:
  - RANGE_TABLE_PREFIX: Tiền tố của tên bảng con, thường là range_part (ví dụ: range_part0, range_part1).
  - openconnection: Đối tượng kết nối PostgreSQL để thực hiện truy vấn.
  - Hàm count_partitions trả về số lượng bảng con Range hiện có trong cơ sở dữ liệu, thường bằng cách truy vấn information_schema.tables để đếm các bảng có tên bắt đầu bằng RANGE_TABLE_PREFIX.
  - Kết quả được lưu vào biến numberofpartitions.

+ Tính chỉ số partition dựa trên rating:

- delta = 5.0 / numberofpartitions: Chia khoảng giá trị rating [0, 5] thành numberofpartitions phân đoạn đều nhau.
- Tính chỉ số partition: index = int(rating / delta): Xác định chỉ số partition bằng cách chia rating cho delta và lấy nguyên (phép chia làm tròn xuống).
- Điều chỉnh giá trị biện : if rating % delta == 0 and index != 0: index = index - 1: Xử lý các giá trị rating nằm ở ranh giới giữa các phạm vi. Điều kiện này áp dụng khi rating chia hết cho delta (tức là rating là giá trị biên như 1.0, 2.0, 3.0, 4.0) và index không phải 0. Ví dụ: Nếu rating = 1.0 và delta = 1.0, thì index = int(1.0 / 1.0) = 1. Vì rating % delta = 0 và index = 1, nên index được giảm thành 0, ánh xạ vào range_part0. Điều này đảm bảo các giá trị biên (trừ rating = 0) thuộc về partition trước đó, phù hợp với phạm vi [a, b].
- Ý nghĩa : Ánh xạ giá trị rating vào đúng bảng con dựa trên phạm vi được định nghĩa. Xử lý các giá trị biên để đảm bảo phân mảnh nhất quán với cấu trúc Range. Tính toán đơn giản, không phụ thuộc vào trạng thái toàn cục như roundrobininsert.

```
# Đếm số partition
numberofpartitions = count_partitions(RANGE_TABLE_PREFIX, openconnection)
delta = 5.0 / numberofpartitions
index = int(rating / delta)
if rating % delta == 0 and index != 0:
    index = index - 1
table_name = RANGE_TABLE_PREFIX + str(index)
```

+ Chèn bảng ghi vào bảng chính và partition tương ứng:

- Đảm bảo bản ghi được lưu vào bảng chính trước khi chèn vào bảng con, duy trì tính toàn vẹn dữ liệu.
- Giữ đồng bộ dữ liệu giữa bảng chính và bảng con, hỗ trợ kiểm tra hoặc khôi phục nếu cần.
- Bản ghi được lưu vào đúng bảng con tương ứng với phạm vi rating, duy trì cấu trúc phân mảnh Range.
- Đảm bảo dữ liệu được phân phối đúng theo phạm vi đã thiết lập bởi rangepartition.

```
    # Chèn vào bảng chính
    cur.execute(
        "INSERT INTO %s (userid, movieid, rating) VALUES (%s, %s, %s)"
        % (ratingstablename, userid, itemid, rating))

    # Chèn vào partition tương ứng
    cur.execute(
        "INSERT INTO %s (userid, movieid, rating) VALUES (%s, %s, %s)"
         % (table_name, userid, itemid, rating))
```

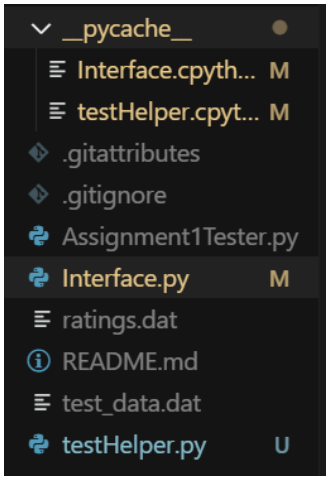
+ Đảm bảo giao dịch được thực thi an toàn: dữ liệu hoặc được lưu hoàn chỉnh hoặc không lưu gì cả.
+ Quản lý tài nguyên hiệu quả, tránh rò rỉ kết nối hoặc con trỏ.
+ Cung cấp thông tin lỗi để xử lý bên ngoài nếu cần.

```
        con.commit()
    except Exception:
        con.rollback()
        raise
    finally:
        cur.close()
```

III. Kiểm thử

1. Cấu hình chung

– Đặt file dữ liệu ratings.dat và test_data.dat như trong ảnh:



– Sửa đường dẫn để kiểm tra dữ liệu file ratings.dat sửa trong Assignment1Tester.py

```
#
# Tester for the assignement1
#
DATABASE_NAME = 'dds_assgn1'

# TODO: Change these as per your code
RATINGS_TABLE = 'ratings'
RANGE_TABLE_PREFIX = 'range_part'
RROBIN_TABLE_PREFIX = 'rrobin_part'
USER_ID_COLNAME = 'userid'
MOVIE_ID_COLNAME = 'movieid'
RATING_COLNAME = 'rating'
#INPUT_FILE_PATH = 'test_data.dat'
#ACTUAL_ROWS_IN_INPUT_FILE = 20  # Number of lines in the input file
INPUT_FILE_PATH = 'ratings.dat'
ACTUAL_ROWS_IN_INPUT_FILE = 10000054 # Number of lines in the input file
```

2. Kết quả kiểm thử

2.1 Kết quả chung

- Đánh giá tổng quan về hiệu suất và tính đúng đắn của toàn bộ hệ thống phân mảnh dữ liệu. Các tiêu chí đánh giá bao gồm:

+ Đảm bảo tất cả các test case được chạy qua mà không có lỗi biên dịch.
+ Kiểm tra tính toàn vẹn dữ liệu trong các bảng phân mảnh.
+ Đảm bảo dữ liệu được phân bố đúng cách vào các phân mảnh theo thuật toán đã chọn.
+ Đánh giá hiệu suất của các hàm khi xử lý tập dữ liệu lớn.

```
PS D:\BTLCSDLPT> python Assignment1Tester.py
A database named "dds_assgn1" already exists
loadratings function pass!
rangepartition function pass!
rangeinsert function pass!
roundrobinpartition function pass!
roundrobininsert function pass!
Press enter to Delete all tables?
```

2.2 Kiểm thử hàm loadratings

- Mô tả : Kiểm tra khả năng tải dữ liệu từ tệp ratings.dat vào bảng Ratings chính.
- Thực thi hàm LoadRatings() với đường dẫn tuyệt đối đến tệp ratings.dat. Kết quả sau khi thực thi sẽ trả ra số lượng bản ghi trong loadrating.

```
PS D:\BTLCSDLPT> python Assignment1Tester.py
A database named "dds_assgn1" already exists
loadratings function pass!
```

- Số lượng data được thêm vào trong bảng ratings

Query | Query History

```
1  SELECT * FROM public.ratings
2  LIMIT 100
```

Data Output | Messages | Notifications

Showing rows: 1 to 100 | Page

	userid integer	movieid integer	rating double precision
1	1	122	5
2	1	185	5
3	1	231	5
4	1	292	5
5	1	316	5
6	1	329	5
7	1	355	5
8	1	356	5
9	1	362	5
10	1	364	5
11	1	370	5
12	1	377	5
13	1	420	5

Total rows: 100 | Query complete 00:00:00.900

2.3 Kiểm thử hàm rangepartition

- Thực thi hàm rangepartition() với bảng Ratings.
- Kết quả : Hàm rangepartition vượt qua test , không có lỗi biên dịch
- Với N = 5 là tạo ra 5 phân mảnh

```
PS D:\BTLCSDLPT> python Assignment1Tester.py
A database named "dds_assgn1" already exists
loadratings function pass!
rangepartition function pass!
```

2.4 Kiểm thử hàm roundrobinpartition

- Kiểm tra khả năng phân mảnh bảng Ratings theo phương pháp phân mảnh vòng tròn (round robin partition)
- Kết quả : Hàm roundrobinpartition vượt qua test, không có lỗi biên dịch

```
A database named "dds_assgn1" already exists
loadratings function pass!
rangepartition function pass!
rangeinsert function pass!
roundrobinpartition function pass!
roundrobininsert function pass!
```

2.5 Kiểm thử hàm rangeinsert

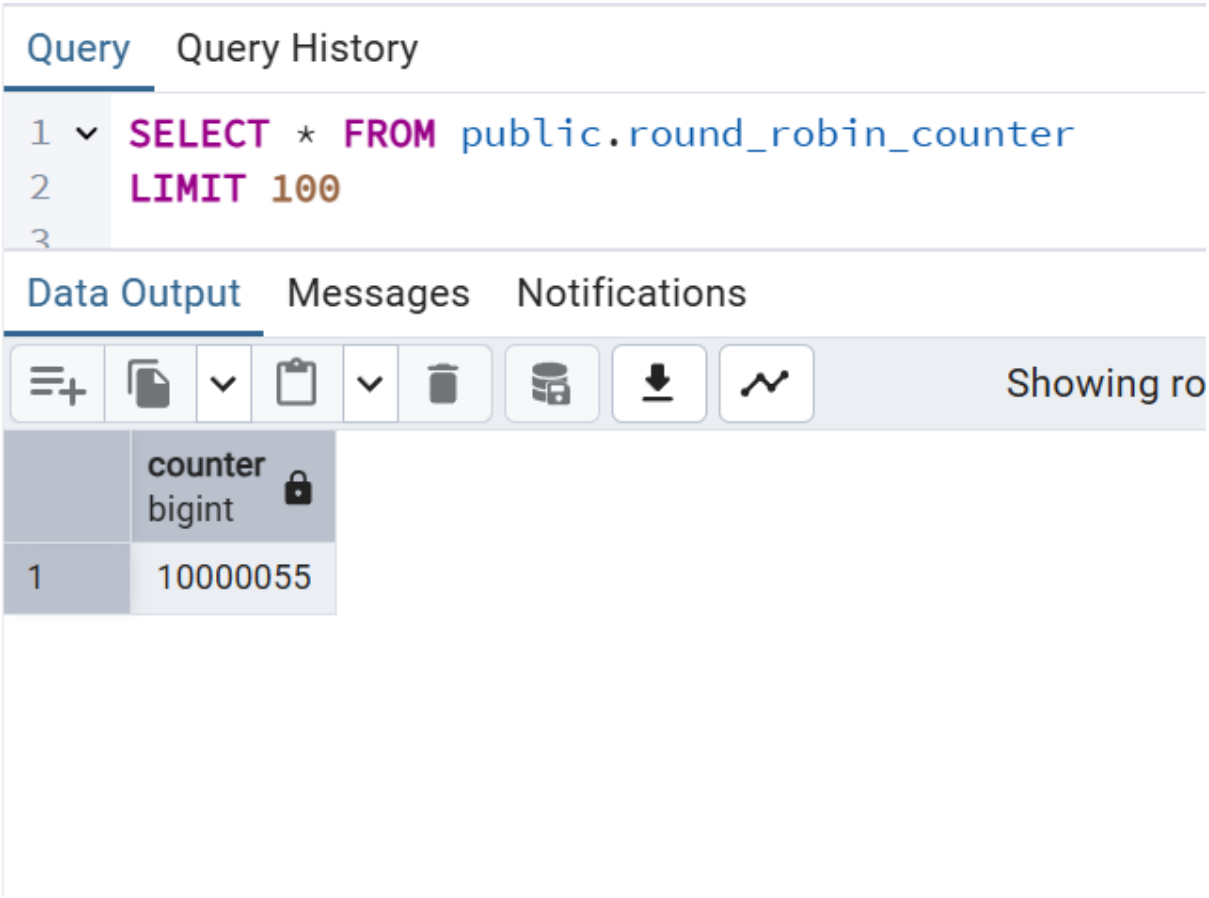
- Kiểm tra khả năng chèn một bộ dữ liệu mới vào đúng phân mảnh theo phương pháp range partition.
- Kết quả : Hàm rangeinsert vượt qua test,không có lỗi biên dịch

```
A database named "dds_assgn1" already exists
loadratings function pass!
rangepartition function pass!
rangeinsert function pass!
roundrobinpartition function pass!
roundrobininsert function pass!
```

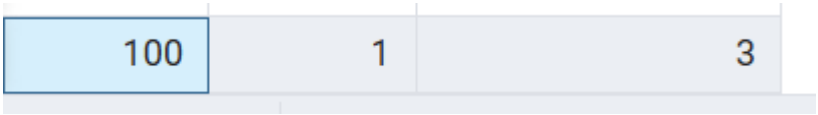
2.6 Kiểm thử hàm roundrobininsert

- Kiểm tra khả năng chèn một bộ dữ liệu mới vào đúng phân mảnh theo phương pháp round robin partition
- Kết quả : Hàm roundrobininsert vượt qua test , không có lỗi biên dịch

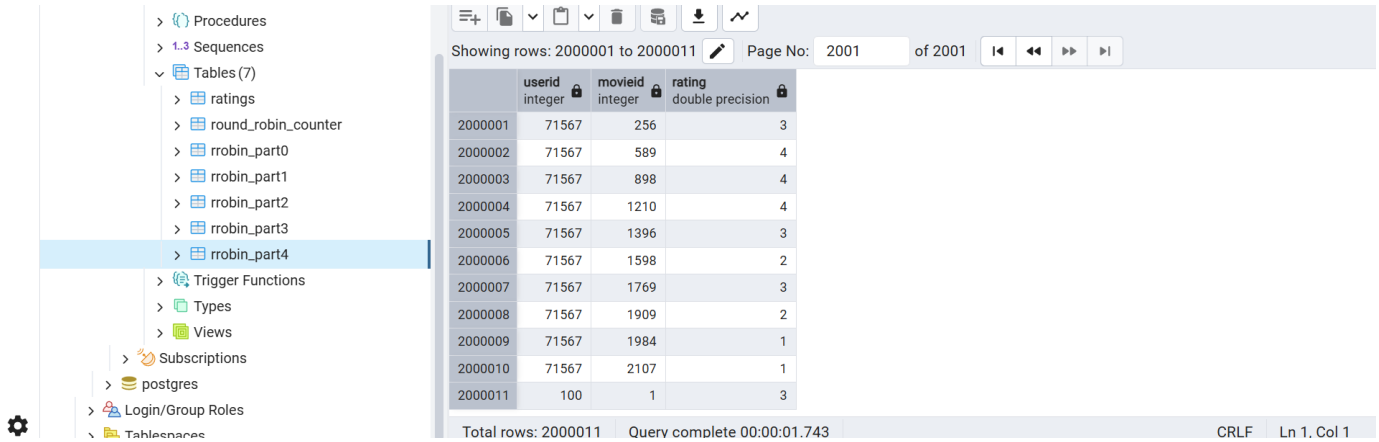
Trong đó bảng round_robin_counter đếm kết quả 10000055 do sau khi chèn vào tăng thêm 1



Kết quả chèn :



Bản ghi vừa thêm có data là userid: 100, movieid: 1, rating: 3, thực hiện kiểm tra trong phân mảnh 4:



IV. Kết luận

Bài tập lớn Cơ sở dữ liệu phân tán đã được nhóm chúng em hoàn thành, triển khai thành công phân mảnh ngang theo phương pháp Range và Round Robin trên PostgreSQL với các hàm loadratings, rangepartition, rangeinsert, roundrobinpartition, và roundrobininsert:

• loadratings tải chính xác 10,000,054 bản ghi từ ratings.dat vào bảng ratings (userid: int, movieid: int, rating: float).

• rangepartition tạo N phân mảnh (range_part0 đến range_partN-1) dựa trên khoảng giá trị rating đồng đều, ví dụ N=3: [0, 1.67], (1.67, 3.34], (3.34, 5].

• roundrobinpartition tạo N phân mảnh (rrobin_part0 đến rrobin_partN-1) phân phối tuần tự bản ghi dùng ROW_NUMBER().

• rangeinsert và roundrobininsert chèn bản ghi mới (ví dụ (100, 1, 3)) vào bảng gốc và phân mảnh đúng, với roundrobininsert dùng round_robin_counđể đảm bảo chu kỳ Round Robin.

Kiểm thử bằng Assignment1Tester.py xác nhận tính đúng đắn, nhưng lỗi ban đầu “Couldnt find (100, 1, 3) in rrobin_part4” được khắc phục bằng cách đồng bộ counter trong roundrobinpartition với 10,000,054 bản ghi, đảm bảo chè vào rrobin_part4 (10000054 % 5 = 4).

Thành tựu:

• Đáp ứng yêu cầu không dùng biến toàn cục, không sửa tệp dữ liệu, sử dụng bảng meta-data round_robin_counter.

• Đảm bảo hoàn chỉnh, không trùng lặp, tái tạo dữ liệu qua kiểm thử.

• Nắm vững phân mảnh dữ liệu và kỹ năng dùng PostgreSQL, Python cho dữ liệu lớn.

Hạn chế:

• ROW_NUMBER() thiếu ORDER BY có thể gây thứ tự không nhất quán.

• Hiệu suất cần tối ưu cho dữ liệu cực lớn.

Hướng phát triển:

• Thêm ORDER BY trong roundrobinpartition để cố định thứ tự.

• Tối ưu truy vấn SQL và lập chỉ mục cho hiệu suất.

• Thêm kiểm tra cấu hình trong roundrobininsert để tránh lỗi.

Bài tập lớn giúp nhóm hiểu sâu về phân mảnh dữ liệu phân tán, đặt nền tảng cho việc xây dựng hệ thống hiệu quả, mở rộng trong thực tế.