HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN

MÔN HỌC: CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên hướng dẫn: Kim Ngọc Bách

Nhóm 18

Thành viên nhóm

Trần Văn Đức	B22DCCN246
Nguyễn Phúc Hưng	B22DCCN414
Nguyễn Đức Mạnh	B22DCCN522

Hà Nội 2025

MỤC LỤC

LỜI CẢM ƠN

Cảm ơn thầy Kim Ngọc Bách đã quan tâm đến báo cáo bài tập lớn cơ sở dữ liệu phân tán của Nhóm 18. Chúng em muốn bày tỏ lòng biết ơn sâu sắc đến gia đình, người yêu và bạn bè vì sự hỗ trợ và động viên không ngừng. Đặc biệt, chúng em muốn gửi lời tri ân đặc biệt tới thầy cô đã dành thời gian và kiến thức quý báu để hướng dẫn chúng em. Cuối cùng, chúng em cũng muốn tỏ lòng biết ơn bản thân mình vì sự quyết tâm và nỗ lực không ngừng nghỉ trong suốt quá trình này. Đây là một hành trình đáng nhớ, và nhóm 18 rất vui vì kết quả mà chúng em đã đạt được.

DANH MỤC HÌNH ẢNH

1. Giới Thiệu

1.1. Mục tiêu

Bài tập lớn mô phỏng các phương pháp phân mảnh ngang trong cơ sở dữ liệu phân tán sử dụng PostgreSQL, tập trung vào dữ liệu đánh giá phim từ tập dữ liệu MovieLens. Mục tiêu bao gồm:

- Tải dữ liệu từ tệp ratings.dat (10 triệu dòng) hoặc test_data.dat (20 dòng) vào bảng Ratings.
- Phân mảnh ngang bảng Ratings theo hai phương pháp: **range** (dựa trên khoảng giá trị rating) và **round-robin** (phân phối tuần tự).
- Hỗ trợ chèn bản ghi mới vào bảng chính và các aphân mảnh tương ứng.
- Đảm bảo tính chất phân mảnh (hoàn chỉnh, rời rạc, tái tạo) thông qua công cụ kiểm tra tự động (assignment_tester.py).
- Phân tích hiệu năng và so sánh hai phương pháp phân mảnh.

1.2. Ý nghĩa

Phân mảnh ngang giúp xử lý dữ liệu lớn hiệu quả, giảm tải cho từng nút trong hệ phân tán, tăng tốc độ truy vấn, và hỗ trợ mở rộng. Bài tập cung cấp kinh nghiệm thực tế về phân mảnh, tối ưu hóa, và kiểm tra trong cơ sở dữ liệu phân tán.

2. Cơ Sở Lý Thuyết

2.1. Cơ sở dữ liệu phân tán

Cơ sở dữ liệu phân tán lưu trữ dữ liệu trên nhiều nút (node) để:

- Tăng hiệu suất thông qua song song hóa.
- Cải thiện khả năng mở rộng (scale-out).
- Đảm bảo độ tin cậy và khả năng chịu lỗi. Dữ liệu được **phân mảnh** (fragmentation) và **phân phối** (allocation) để tối ưu hóa truy vấn và quản lý.

2.2. Phân mảnh ngang

Phân mảnh ngang chia một bảng thành các phân mảnh, mỗi phân mảnh chứa một tập hợp con của bản ghi dựa trên một tiêu chí. Các tính chất quan trọng:

- **Hoàn chỉnh**: Mỗi bản ghi trong bảng gốc thuộc ít nhất một phân mảnh.
- **Rời rạc**: Không có bản ghi trùng lặp giữa các phân mảnh.
- **Tái tạo**: Có thể tái tạo bảng gốc bằng cách hợp các phân mảnh.

2.3. Phân mảnh range

Nguyên lý của Range Partitioning

- **Mục tiêu:** Phân chia dữ liệu dựa trên các khoảng giá trị của một hoặc nhiều thuộc tính. Mỗi phân vùng (partition) chứa dữ liệu nằm trong một khoảng giá trị cụ thể.
- **Cách hoạt động:** Dữ liệu được chia thành các phân vùng dựa trên giá trị của một cột (ví dụ: cột "Ngày" hoặc cột "Tên"). Mỗi phân vùng tương ứng với một khoảng giá trị nhất định.

Ưu điểm:

- **Phù hợp với truy vấn phạm vi:** Tối ưu hóa các truy vấn liên quan đến khoảng giá trị (ví dụ: truy vấn theo thời gian hoặc theo thứ tự bảng chữ cái).
- **Dễ quản lý:** Các phân vùng được xác định rõ ràng dựa trên khoảng giá trị, giúp việc quản lý dữ liệu trở nên dễ dàng hơn.

Nhược điểm:

- **Cân bằng tải không đều:** Nếu dữ liệu tập trung vào một khoảng giá trị cụ thể, một số phân vùng có thể chứa nhiều dữ liệu hơn các phân vùng khác, dẫn đến mất cân bằng tải.
- **Khó điều chỉnh:** Khi khoảng giá trị thay đổi, cần phải điều chỉnh lại các phân vùng, có thể gây tốn kém chi phí.

Kết luận: Range Partitioning là một phương pháp hiệu quả để tối ưu hóa các truy vấn phạm vi, nhưng cần lưu ý đến việc cân bằng tải và quản lý các khoảng giá trị để tránh mất cân bằng trong hệ thống.

2.4. Phân mảnh round-robin

Nguyên lý của Round-robin Partitioning

- **Mục tiêu:** Phân chia dữ liệu một cách đồng đều giữa các site (node) hoặc phân vùng (partition) mà không cần xem xét đặc điểm của dữ liệu.
- **Cách hoạt động:** Dữ liệu được phân bổ luân phiên theo thứ tự giữa các site. Ví dụ: nếu có 3 site, bản ghi đầu tiên được gán cho site 1, bản ghi thứ hai cho site 2, bản ghi thứ ba cho site 3, bản ghi thứ tư lại quay về site 1, và tiếp tục như vậy.

Ưu điểm:

- **Đơn giản:** Dễ triển khai và không yêu cầu phân tích dữ liệu phức tạp.
- **Cân bằng tải:** Đảm bảo dữ liệu được phân bổ đều giữa các site, giúp cân bằng tải xử lý.

Nhược điểm:

- **Không tối ưu hóa truy vấn:** Không xem xét đặc điểm của truy vấn hoặc dữ liệu, có thể dẫn đến hiệu suất không tối ưu.
- **Không phù hợp với truy vấn phức tạp:** Các truy vấn liên quan đến nhiều bản ghi có thể cần truy cập dữ liệu từ nhiều site, làm tăng chi phí truyền tải.

Kết luận: Round-robin Partitioning là một phương pháp đơn giản và hiệu quả để cân bằng tải, nhưng có thể không phù hợp với các hệ thống yêu cầu tối ưu hóa cao dựa trên đặc điểm truy vấn.

2.5. Ví dụ minh họa

Giả sử bảng Ratings có 6 bản ghi:

UserID	MovieID	Rating
1	122	5.0
2	185	3.0
3	231	1.0
4	292	4.0
5	316	2.0
6	329	3.5

- **Range (**N=2**)**:
    - range_part0 ([0, 2.5]): (3, 231, 1.0), (5, 316, 2.0).
    - range_part1 ((2.5, 5]): (1, 122, 5.0), (2, 185, 3.0), (4, 292, 4.0), (6, 329, 3.5).
- **Round-robin (**N=2**)**:
    - rrobin_part0: (1, 122, 5.0), (3, 231, 1.0), (5, 316, 2.0).
    - rrobin_part1: (2, 185, 3.0), (4, 292, 4.0), (6, 329, 3.5).

2.6. Ứng dụng trong bài tập

Phân mảnh ngang giúp xử lý 10 triệu bản ghi từ ratings.dat hiệu quả hơn. Range phù hợp để phân tích rating theo khoảng, trong khi round-robin đảm bảo phân phối đều, giảm tải cho từng phân mảnh.

3. Phân tích yêu cầu

3.1. Yêu cầu dữ liệu đầu vào

- **Tệp dữ liệu**:
  - test_data.dat(Thầy đã cung cấp): 20 dòng, dùng để kiểm tra nhanh.
  - ratings.dat: 10 triệu dòng, tải từ trang MovieLens.
- **Định dạng**: Mỗi dòng có dạng UserID::MovieID::Rating::Timestamp.
  - UserID: ID người dùng (int).
  - MovieID: ID phim (int).
  - Rating: Điểm đánh giá (float, 0–5, chia nửa sao).
  - Timestamp: Thời gian (bỏ qua).

  Ví dụ:
  1::122::5::838985046
  1::185::5::838983525
  1::231::5::838983392

- **Quy mô**: ratings.dat chứa 10 triệu đánh giá từ 72.000 người dùng cho 10.000 phim.

3.2. Yêu cầu chức năng

Cần triển khai 5 hàm Python:

1. **LoadRatings(ratingstable, ratingsfilepath, connection)**: Tải dữ liệu vào bảng Ratings (schema: UserID INTEGER, MovieID INTEGER, Rating FLOAT).
2. **Range_Partition(ratingstable, N, connection)**: Phân mảnh ngang thành N bảng range_partX.
3. **RoundRobin_Partition(ratingstable, N, connection)**: Phân mảnh ngang thành N bảng rrobin_partX.
4. **Range_Insert(ratingstable, userid, itemid, rating, connection)**: Chèn bản ghi vào Ratings và phân mảnh range tương ứng.
5. **RoundRobin_Insert(ratingstable, userid, itemid, rating, connection)**: Chèn bản ghi vào Ratings và phân mảnh round-robin tương ứng.

3.3. Yêu cầu kiểm tra

Dựa trên assignment_tester.py và testHelper.py:

- **LoadRatings**: Tải đúng 20 dòng từ test_data.dat,hoặc 10000054 dòng từ ratings.dat.
- **Range_Partition**: Tạo N = 5 phân mảnh (e.g., [0, 1], (1, 2], ..., (4, 5]).
- **RoundRobin_Partition**: Tạo N = 5 phân mảnh, phân phối đều.
- **Range_Insert**: Chèn đúng (e.g., rating=3 vào range_part2).
- **RoundRobin_Insert**: Chèn đúng dựa trên thứ tự chèn.
- **Tính chất**:
    - Hoàn chỉnh: Tổng bản ghi trong phân mảnh bằng bảng gốc.
    - Rời rạc: Không trùng lặp.
    - Tái tạo: Hợp các phân mảnh tạo lại bảng gốc.

4. Thiết kế hệ thống

4.1. Schema cơ sở dữ liệu

- Bảng Ratings:
    - UserID (INTEGER): ID người dùng.
    - MovieID (INTEGER): ID phim.
    - Rating (FLOAT): Điểm đánh giá (0–5).
- Bảng range_partX:
    - Schema giống Ratings.
    - Chứa bản ghi trong khoảng rating (e.g., N=5: [0, 1], (1, 2], ..., (4, 5]).
- Bảng rrobin_partX:
    - Schema giống Ratings.
    - Phân phối tuần tự.
- Bảng metadata:
    - partitiontype (VARCHAR): Tên khóa (e.g., rr_index).
    - partitioncount: Số phân vùng
    - totalinserts: số dòng hiện tại của bảng ratings

4.2. Thiết kế thuật toán

4.2.1. LoadRatings

- **Input**: ratingsfilepath, openconnection.
- **Output**: Bảng Ratings chứa dữ liệu.
- **Mã giả**:

  Drop table Ratings if exists

Create table Ratings (UserID INTEGER, MovieID INTEGER, Rating FLOAT, extra columns for parsing)
Open file ratingsfilepath
Use copy_from to load file content into Ratings
Drop extra columns (extra1, extra2, extra3, Timestamp)
Commit changes

- **Ví dụ**: Với dòng 1::122::5::838985046, chèn (1, 122, 5.0).

4.2.2. Range_Partition

- **Input**: numberofpartitions (N), openconnection.
- **Output**: N bảng range_partX.
- **Mã giả**:

```
If N <= 0, print error and exit
Initialize partitionmetadata table
Calculate delta = 5.0 / N
Drop existing range_part* tables
For i from 0 to N-1:
   Create table range_part{i} (UserID INTEGER, MovieID INTEGER, Rating FLOAT)
   If i = 0:
      Insert records from Ratings where Rating >= i*delta and Rating <= (i+1)*delta
   Else:
      Insert records from Ratings where Rating > i*delta and Rating <= (i+1)*delta
Update partitionmetadata with type='range', count=N
Commit changes
```

- **Ví dụ**: Với N=2, range_part0 chứa Rating từ [0, 2.5], range_part1 chứa (2.5, 5].

4.2.3. RoundRobin_Partition

- **Input**: numberofpartitions (N), openconnection.
- **Output**: N bảng rrobin_partX.
- **Mã giả**:

```
If N <= 0, print error and exit
```

```
Initialize partitionmetadata table
Drop existing rr_part* tables
For i from 0 to N-1:
   Create table rr_part{i} (UserID INTEGER, MovieID INTEGER, Rating
FLOAT)
   Insert records from Ratings where (ROW_NUMBER() - 1) % N = i
Update partitionmetadata with type='rrobin', count=N
Commit changes
```

- **Ví dụ**: Với N=2, bản ghi thứ 1, 3, 5 vào rrobin_part0; thứ 2, 4, 6 vào rrobin_part1.

4.2.4. Range_Insert

- **Input**: userid, itemid, rating, openconnection.
- **Output**: Bản ghi mới trong Ratings và phân mảnh range.
- **Mã giả**:

```
If rating not in [0, 5] or userid, itemid not integer, print error and exit
Get N from partitionmetadata for type='range'
If N = 0, print error and exit
delta = 5.0 / N
If rating == 5.0:
   index = N - 1
Elif rating == 0:
   index = 0
Else:
   index = int(rating / delta)
   If rating <= index * delta:
      index -= 1
Insert (userid, itemid, rating) into Ratings
Insert (userid, itemid, rating) into range_part{index}
Commit changes
```

- **Ví dụ**: Với rating=3, N=5, chèn vào range_part2.

4.2.5. RoundRobin_Insert

- **Input**: userid, itemid, rating, openconnection.
- **Output**: Bản ghi mới trong Ratings và phân mảnh round-robin.

- **Mã giả**:

  If rating not in [0, 5] or userid, itemid not integer, print error and exit
  Get N and row_count from partitionmetadata for type='rrobin'
  If N = 0, print error and exit
  Calculate index = row_count % N
  Insert (userid, itemid, rating) into Ratings
  Insert (userid, itemid, rating) into rr_part{index}
  Update row_count in partitionmetadata
  Commit changes

- **Ví dụ**: Với rr_index=0, N=5, chèn vào rrobin_part0.

4.3. Tổ chức mã nguồn

Project được tổ chức thành các file:

Tên file	**Vai trò chính**
Interface.py	Chứa toàn bộ các hàm xử lý chính liên quan tới phân mảnh, chèn dữ liệu.
assigment_tester.py	Dùng để gọi hàm từ Interface.py để thực hiện phân mảnh và nạp dữ liệu.
testHelper.py	Tạo đầu vào cho các hàm kiểm thử, lưu kết quả đầu ra và đối chiếu tính đúng đắn.
ratings.dat	Tệp dữ liệu chính dùng để load vào bảng ratings

test_data.dat	Dữ liệu mẫu dùng để kiểm thử

5. Chi tiết triển khai

5.1. Môi trường triển khai

- **Hệ điều hành**: Ubuntu 20.04.
- **PostgreSQL**: Phiên bản 16.
- **Python**: 3.12.x, psycopg2-binary.
- **Dữ liệu**: test_data.dat (20 dòng), ratings.dat (10 triệu dòng).

5.2. Các bước cài đặt

- Thiết lập **venv** cho dự án bằng cách :
  - cài đặt **sudo apt install python3-venv -y**
  - Tạo venv bằng lệnh **python3 -m venv venv**
  - Kích hoạt môi trường ảo với lệnh **source venv/bin/activate**
  - Cài đặt thư viện **psycopg2-binary** bằng lệnh **pip install psycopg2-binary**
- Thiết lập **postgreSQL**
  - Cài đặt bằng lệnh **sudo apt install postgresql postgresql-contrib**
  - Đặt mật khẩu cho tài khoản mặc định postgres **ALTER USER postgres WITH PASSWORD '1';**
- Cài đặt dữ liệu
  - test_data.dat: Trong repository.
  - ratings.dat: Tải từ MovieLens(trong respository).

5.3. Triển khai code

Thực hiện triển khai các hàm trong file Interface.py, 2 file assigment_tester.py và testHelper.py được giữ nguyên logic

5.3.1. getopenconnection()

Mô tả: Hàm getopenconnection được sử dụng để thiết lập kết nối tới cơ sở dữ liệu PostgreSQL sử dụng thư viện psycopg2. Nếu cơ sở dữ liệu chưa tồn tại, hàm này sẽ tự động tạo mới rồi kết nối lại, đảm bảo quá trình thực thi các hàm tiếp theo không bị gián đoạn do thiếu cơ sở dữ liệu.

Tham số đầu vào:

- user: Tên người dùng PostgreSQL.
- password: Mật khẩu người dùng.
- dbname: Tên cơ sở dữ liệu.

Mã nguồn:

```
def getopenconnection(user='postgres', password='1', dbname='csdlpt'):
    try:
        conn = psycopg2.connect(dbname=dbname, user=user, password=password, host='localhost')
        print("Kết nối thành công đến database:", dbname)
        return conn
    except OperationalError as e:
        if 'database "{}" does not exist'.format(dbname) in str(e):
            print(f"Database {dbname} chưa tồn tại, sẽ tạo mới.")
            create_db(dbname, user=user, password=password)
            return getopenconnection(user=user, password=password, dbname=dbname)
        else:
            print("Lỗi khi kết nối đến database:", e)
            return None
```

Hình 5.3.1 getopenconnection()

5.3.2. create_db()

Mô tả: Hàm create_db được sử dụng để tạo cơ sở dữ liệu PostgreSQL mới nếu cơ sở dữ liệu chưa tồn tại. Hàm này kết nối đến cơ sở dữ liệu postgres mặc định để kiểm tra và thực hiện lệnh tạo cơ sở dữ liệu mục tiêu.

Tham số đầu vào:

- dbname: Tên cơ sở dữ liệu cần tạo.
- user: Tên người dùng PostgreSQL.
- password: Mật khẩu tương ứng với người dùng.

Mã nguồn:

```
def create_db(dbname='postgres', user='postgres', password='1'):
    con = getopenconnection(user=user, password=password, dbname='postgres')
    if con is None:
        print("Không thể kết nối đến postgres để tạo database")
        return
    con.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    cur = con.cursor()
    cur.execute('SELECT COUNT(*) FROM pg_catalog.pg_database WHERE datname=%s;', (dbname,))
    count = cur.fetchone()[0]
    if count == 0:
        cur.execute('CREATE DATABASE %s;' % dbname)
        print("Database %s created." % dbname)
    else:
        print("Database %s already exists." % dbname)
    cur.close()
```

Hình 5.3.2 create_db()

5.3.3. LoadRatings()

Mô tả: Hàm LoadRatings dùng để nạp dữ liệu từ tệp ratings.dat vào một bảng mới trong PostgreSQL. Dữ liệu được xử lý và làm sạch trong quá trình nạp để chỉ giữ lại các cột cần thiết: userid, movieid, và rating.

Tham số đầu vào:

- ratingstable: Tên bảng trong cơ sở dữ liệu mà dữ liệu sẽ được nạp vào.
- ratingsfilepath: Đường dẫn tới file ratings.dat.
- connection: Đối tượng kết nối đến cơ sở dữ liệu PostgreSQL.

Mã nguồn:

```
def LoadRatings(ratingstable, ratingsfilepath,connection):
    cur = connection.cursor()
    cur.execute("DROP TABLE IF EXISTS " + ratingstable + ";")
    cur.execute("CREATE TABLE " + ratingstable + "(userid integer, extra1 char, movieid integer, extra2 char, rating float, extra3 char, timestamp bigint);")
    cur.execute("TRUNCATE TABLE " + ratingstable + ";")
    connection.commit()
    try:
        cur.copy_from(open(ratingsfilepath), ratingstable, sep=':')
        cur.execute("ALTER TABLE " + ratingstable + " DROP COLUMN extra1, DROP COLUMN extra2, DROP COLUMN extra3, DROP COLUMN timestamp;")
    except IOError as e:
        print("Lỗi khi đọc file:", e)
        cur.close()
        return
    cur.close()
    connection.commit()
```

Hình 5.3.3 LoadRatings()

5.3.4. Range_Partition()

Mô tả: Hàm Range_Partition được sử dụng để phân mảnh bảng ratingstable thành N bảng con dựa trên giá trị rating, theo phương pháp phân mảnh theo khoảng (Range Partitioning). Các bản ghi được chia đều thành các phân vùng tương ứng với khoảng giá trị rating từ 0 đến 5.

Tham số đầu vào:

- ratingstable: Tên bảng lưu dữ liệu gốc cần phân mảnh.
- N: Số phân vùng mong muốn (phải > 0).
- connection: Đối tượng kết nối PostgreSQL (psycopg2.connect).

Mã nguồn:

```python
def Range_Partition(ratingstable, N,connection):

    if N <= 0:
        print("Số phân vùng phải lớn hơn 0")
        return
    init_metadata_table(connection)

    delta = 5.0 / N
    PREFIX = 'range_part'
    cur = connection.cursor()
    cur.execute("""
        SELECT tablename
        FROM pg_tables
        WHERE tablename LIKE %s;
    """, (PREFIX + '%',))
    old_tables = cur.fetchall()

    for (table_name,) in old_tables:
        cur.execute(f"DROP TABLE IF EXISTS {table_name} CASCADE;")

    for i in range(N):
        minRange = i * delta
        maxRange = minRange + delta
        table_name = PREFIX + str(i)

        cur.execute(f"""
            CREATE TABLE {table_name} (
                userid integer,
                movieid integer,
                rating float
            );
        """)

        if i == 0:
            sql = f"""
                INSERT INTO {table_name} (userid, movieid, rating)
                SELECT userid, movieid, rating FROM {ratingstable}
                WHERE rating >= %s AND rating <= %s;
            """
            params = (minRange, maxRange)
        else:
            sql = f"""
                INSERT INTO {table_name} (userid, movieid, rating)
                SELECT userid, movieid, rating FROM {ratingstable}
                WHERE rating > %s AND rating <= %s;
            """
            params = (minRange, maxRange)

        cur.execute(sql, params)
    cur.close()
    update_metadata(connection, 'range', N)
    connection.commit()
```

Hình 5.3.4 Range_Partition()

5.3.5. RoundRobin_Partition()

Mô tả: Hàm RoundRobin_Partition dùng để phân mảnh bảng ratingstable thành N phân vùng theo phương pháp **Round-Robin**, nghĩa là các bản ghi được phân phối đều và luân phiên vào các bảng con theo thứ tự tuần tự. Điều này giúp đảm bảo phân phối dữ liệu gần như đồng đều, không phụ thuộc vào giá trị rating.

Tham số đầu vào:

- ratingstable: Tên bảng gốc cần phân mảnh.
- N: Số phân vùng cần tạo (phải lớn hơn 0).
- connection: Đối tượng kết nối cơ sở dữ liệu PostgreSQL (psycopg2.connect).

Mã nguồn:

```
def RoundRobin_Partition(ratingstable, N ,connection):

    if N <= 0:
        print("Số phân vùng phải lớn hơn 0")
        return
    init_metadata_table(connection)

    cur = connection.cursor()

    PREFIX = 'rrobin_part'
    cur.execute("""
        SELECT tablename
        FROM pg_tables
        WHERE tablename LIKE %s;
    """, (PREFIX + '%',))
    old_tables = cur.fetchall()

    for (table_name,) in old_tables:
        cur.execute(f"DROP TABLE IF EXISTS {table_name} CASCADE;")
    for i in range(N):
        table_name = PREFIX + str(i)
        cur.execute("CREATE TABLE " + table_name + " (userid integer, movieid integer, rating float);")
        cur.execute(
            "INSERT INTO " + table_name + " (userid, movieid, rating) "
            "SELECT userid, movieid, rating FROM (SELECT userid, movieid, rating, ROW_NUMBER() OVER () as rnum FROM " + ratingstable + ") as temp "
            "WHERE mod(temp.rnum - 1, %s) = %s;", (N, i))
    cur.close()
    update_metadata(connection, 'rrobin', N)

    connection.commit()
```

Hình 5.3.5 RoundRobin_Partition()

5.3.6. RoundRobin_Insert()

Mô tả: Hàm RoundRobin_Insert dùng để chèn một bản ghi mới vào bảng ratingstable và đồng thời phân phối bản ghi đó vào bảng phân mảnh tương ứng theo phương pháp **Round-Robin**. Vị trí phân mảnh được xác định dựa trên số thứ tự dòng hiện tại của bảng gốc (ratingstable) chia cho số phân vùng N.

Tham số đầu vào:

- ratingstable: Tên bảng gốc đang lưu dữ liệu rating.
- userid: ID của người dùng.
- itemid: ID của phim.
- rating: Giá trị đánh giá.
- connection: Kết nối đến cơ sở dữ liệu PostgreSQL (psycopg2.connect).

Mã nguồn:

```
def RoundRobin_Insert(ratingstable, userid, itemid, rating,connection):
    cur = connection.cursor()
    PREFIX = 'rrobin_part'
    cur.execute("INSERT INTO " + ratingstable + "(userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
    cur.execute("SELECT partitioncount, totalinserts FROM partitionmetadata WHERE partitiontype = 'rrobin';")

    result = cur.fetchone()
    if not result:
        print("Không tìm thấy metadata cho round-robin.")
        cur.close()
        return
    N, totalinserts = result
    index = totalinserts % N

    table_name = PREFIX + str(index)
    cur.execute("INSERT INTO " + table_name + "(userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
    cur.close()
    update_total_inserts(connection, 'rrobin')
    connection.commit()
```

Hình 5.3.6 RoundRobin_Insert()

5.3.7. Range_Insert()

Mô tả: Hàm Range_Insert thực hiện việc chèn một bản ghi mới vào bảng gốc ratingstable, sau đó phân bổ bản ghi này vào đúng bảng phân vùng theo phương pháp **Range Partitioning**. Phân vùng được xác định dựa trên giá trị rating và số lượng phân vùng N.

Tham số đầu vào:

- ratingstable: Tên bảng chính chứa toàn bộ dữ liệu đánh giá.
- userid: ID người dùng.
- itemid: ID bộ phim.
- rating: Giá trị đánh giá (float).
- connection: Kết nối đến cơ sở dữ liệu PostgreSQL.

Mã nguồn:

```
def Range_Insert(ratingstable, userid, itemid, rating,connection):
    cur = connection.cursor()
    PREFIX = 'range_part'
    # N = count_partitions(connection,PREFIX)
    N = get_partition_count_from_metadata(connection, 'range')
    if N == 0:
        print("Không có bảng phân vùng range nào tồn tại")
        cur.close()
        return
    delta = 5.0 / N
    if rating == 5.0:
        index = N - 1
    elif rating == 0.0:
        index = 0
    else:
        index = int(rating / delta)
        if rating <= (index * delta):
            index -= 1

    table_name = PREFIX + str(index)
    cur.execute("INSERT INTO "+ratingstable+" (userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
    cur.execute("INSERT INTO " + table_name + "(userid, movieid, rating) VALUES (%s, %s, %s);", (userid, itemid, rating))
    update_total_inserts(connection, 'range')
    cur.close()
    connection.commit()
```

Hình 5.3.7 Range_Insert()

5.3.8. init_metadata_table()

Mô tả: Hàm init_metadata_table tạo bảng partitionmetadata nếu bảng này chưa tồn tại trong cơ sở dữ liệu. Bảng này dùng để lưu thông tin về số lượng phân vùng của từng loại phân mảnh (range hoặc round-robin), hỗ trợ truy xuất nhanh trong các thao tác chèn, cập nhật sau này.

Tham số đầu vào: connection: Đối tượng kết nối với cơ sở dữ liệu PostgreSQL.

Mã nguồn:

```
def init_metadata_table(connection):
    with connection.cursor() as cur:

        cur.execute("""
            CREATE TABLE IF NOT EXISTS partitionmetadata (
                partitiontype TEXT PRIMARY KEY,
                partitioncount INTEGER NOT NULL,
                totalinserts INTEGER DEFAULT 0
            );
        """)
        connection.commit()
        cur.execute("SELECT COUNT(*) FROM ratings;")
        total_rows = cur.fetchone()[0]
        # print("total_rows:", total_rows)
        for partition_type in ['range', 'rrobin']:
            cur.execute("""
                INSERT INTO partitionmetadata (partitiontype, partitioncount, totalinserts)
                VALUES (%s, %s, %s)
                ON CONFLICT (partitiontype) DO UPDATE
                SET totalinserts = EXCLUDED.totalinserts;
            """, (partition_type, 0, total_rows))

        connection.commit()
```

Hình 5.3.8 init_metadata_table()

5.3.9. update_metadata()

Mô tả: Hàm update_metadata có nhiệm vụ cập nhật thông tin số lượng phân vùng (partition count) vào bảng partitionmetadata. Nếu kiểu phân vùng (partitiontype) đã tồn tại, nó sẽ cập nhật giá trị partitioncount. Nếu chưa tồn tại, nó sẽ thêm mới bản ghi vào bảng.

Tham số đầu vào:

- connection: Đối tượng kết nối với cơ sở dữ liệu PostgreSQL.
- partition_type: Chuỗi văn bản biểu thị loại phân vùng, ví dụ 'range' hoặc 'rrobin'.
- count: Số lượng phân vùng tương ứng với loại phân vùng.

Mã nguồn:

```
def update_metadata(connection, partition_type, count):
    with connection.cursor() as cur:
        cur.execute("""
            INSERT INTO partitionmetadata (partitiontype, partitioncount)
            VALUES (%s, %s)
            ON CONFLICT (partitiontype) DO UPDATE
            SET partitioncount = EXCLUDED.partitioncount;
        """, (partition_type, count))
        connection.commit()
```

Hình 5.3.9 update_metadata()

5.3.10. update_total_inserts()

Mô tả: Hàm update_total_inserts có nhiệm vụ cập nhật thông tin số lần chèn dữ liệu (insert) vào các phân vùng trong bảng partitionmetadata. Mỗi khi có một bản ghi được chèn vào hệ thống (qua phân mảnh range hoặc round-robin), hàm sẽ tăng biến đếm totalinserts tương ứng với loại phân vùng đó.

Tham số đầu vào:

- connection: Đối tượng kết nối với cơ sở dữ liệu PostgreSQL.
- partition_type: Chuỗi văn bản biểu thị loại phân vùng, ví dụ 'range' hoặc 'roundrobin'.

Mã nguồn:

```
def update_total_inserts(connection, partition_type):
    with connection.cursor() as cur:
        cur.execute("""
            UPDATE partitionmetadata
            SET totalinserts = totalinserts + 1
            WHERE partitiontype = %s;
        """, (partition_type,))
        connection.commit()
```

Hình 5.3.10 update_total_inserts()

5.3.11. get_partition_count_from_metadata()

Mô tả: Hàm get_partition_count_from_metadata có nhiệm vụ truy vấn bảng partitionmetadata để lấy số lượng phân vùng tương ứng với loại phân vùng (partition_type) được truyền vào. Nếu không tìm thấy, hàm trả về 0.

Tham số đầu vào:

- connection: Đối tượng kết nối với cơ sở dữ liệu PostgreSQL.
- partition_type: Chuỗi văn bản biểu thị loại phân vùng cần truy vấn, ví dụ 'range' hoặc 'rrobin'.

Mã nguồn:

```
def get_partition_count_from_metadata(connection, partition_type):
    with connection.cursor() as cur:
        cur.execute("SELECT partitioncount FROM partitionmetadata WHERE partitiontype = %s;", (partition_type,))
        result = cur.fetchone()
        return result[0] if result else 0
```

Hình 5.3.11 get_partition_count_from_metadata()

5.3.12. main()

Mô tả: Hàm main() là điểm bắt đầu chính của chương trình. Nó thực hiện các thao tác chính như tạo cơ sở dữ liệu, nạp dữ liệu, phân mảnh dữ liệu bằng cả hai phương pháp (range và round-robin), sau đó kiểm tra việc chèn bản ghi mới vào từng loại phân vùng.

Mã nguồn:

```
def main():
    create_db('csdlpt')
    con = getopenconnection()
    if con is None:
        print("Không thể kết nối đến database. Thoát chương trình.")
        return

    try:
        LoadRatings('ratings', 'data/ratings.dat',con)
        Range_Partition('ratings', 5,con)
        RoundRobin_Partition('ratings', 5,con)
        RoundRobin_Insert('ratings', 1, 1, 4.5,con)
        Range_Insert('ratings', 2, 2, 3.0,con)
    except Exception as e:
        print("Lỗi trong quá trình xử lý:", e)
    finally:
        con.close()
        print("Đã đóng kết nối đến database.")

if __name__ == '__main__':
    main()
```

Hình 5.3.12 main()

5.4 Tối ưu hoá

Trong quá trình xây dựng chương trình, nhóm đã áp dụng một số cơ chế tối ưu nhằm đảm bảo hiệu suất xử lý, khả năng mở rộng và độ tin cậy:

- **Tối ưu kết nối cơ sở dữ liệu**
  - Hàm getopenconnection() được thiết kế hợp lý để tự động tạo database nếu chưa tồn tại, giúp chương trình khởi chạy mượt mà.
  - Sử dụng cơ chế try-except để xử lý lỗi kết nối và đảm bảo không làm gián đoạn toàn bộ quy trình.
- **Sử dụng COPY_FROM để tải dữ liệu nhanh**
  - Dữ liệu từ file ratings.dat được nạp vào bảng chính thông qua lệnh copy_from, giúp tăng tốc độ đọc file lớn so với các lệnh INSERT thông thường.
- **Xử lý phân mảnh hiệu quả bằng SQL thuần**

    - Hàm rangepartition() và roundrobinpartition() sử dụng các truy vấn SQL phân mảnh trực tiếp trên cơ sở dữ liệu, tận dụng tối đa khả năng xử lý của PostgreSQL.
    - Với RoundRobin, sử dụng ROW_NUMBER() OVER() để đánh số thứ tự dòng giúp phân phối đều dữ liệu mà không cần vòng lặp trong Python.
- **Hàm chèn dữ liệu (insert) tự động phân tích và đưa vào phân vùng thích hợp**
    - Hàm Range_Insert() xác định phân vùng bằng cách so sánh giá trị rating với khoảng range đã định sẵn.

    - Hàm RoundRobin_Insert() sử dụng bảng partitionmetadata để lưu trạng thái, tránh phải đếm lại tổng số bản ghi mỗi lần insert, giúp giảm chi phí xử lý.
- **Kiểm soát phân vùng bằng tiền tố tên bảng**
    - Các bảng phân mảnh được đặt tên theo quy tắc (range_partX, rrobin_partX) để dễ kiểm tra, đếm, truy vấn và mở rộng trong tương lai.

6. Kết quả thực nghiệm,đánh giá

6.1. Với test_data.dat

Số dòng 20 dòng.
Khi chạy LoadRatings ta sẽ có:

```
dds_assgn1=# \dt
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | ratings | table | postgres
(1 row)

dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
(20 rows)
```

Bảng ratings được load với 20 dòng như trên

Range Partition

Sau khi chạy phân mảnh range ta có kết quả:

```
dds_assgn1=# \dt
               List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | partitionmetadata | table | postgres
 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
 public | ratings           | table | postgres
```

5 phân mảnh range đã được tạo ra

Khi test insert dữ liệu có dạng (100,2,3) vào cơ sở dữ liệu

```
dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
    100 |       2 |      3
(21 rows)

dds_assgn1=# select * from range_part2;
 userid | movieid | rating
--------+---------+--------
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     520 |    2.5
    100 |       2 |      3
(4 rows)
```

Dòng dữ liệu đã được thêm vào đúng range_part2 vì có giá trị rating nằm trong khoảng từ (2,3].

Round Robin Partition

Khi thực hiện phân mảnh Round Robin theo 5 phân mảnh thì 5 bảng đã được tạo ra

```
dds_assgn1=# \dt
               List of relations
 Schema |       Name        | Type  |  Owner
--------+-------------------+-------+----------
 public | partitionmetadata | table | postgres
 public | ratings           | table | postgres
 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
(7 rows)
```

Thực hiện insert dữ liệu (100,1,3)

Dữ liệu đã được thêm vào bảng, ta có bảng dữ liệu sau khi được thêm như sau:

```
dds_assgn1=# select * from ratings;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     185 |    4.5
      1 |     231 |      4
      1 |     292 |    3.5
      1 |     316 |      3
      1 |     329 |    2.5
      1 |     355 |      2
      1 |     356 |    1.5
      1 |     362 |      1
      1 |     364 |    0.5
      1 |     370 |      0
      1 |     377 |    3.5
      1 |     420 |      5
      1 |     466 |      4
      1 |     480 |      5
      1 |     520 |    2.5
      1 |     539 |      5
      1 |     586 |    3.5
      1 |     588 |      5
      1 |     589 |    1.5
    100 |       1 |      3
(21 rows)
```

```
dds_assgn1=# select * from rrobin_part0;
 userid | movieid | rating
--------+---------+--------
      1 |     122 |      5
      1 |     329 |    2.5
      1 |     370 |      0
      1 |     520 |    2.5
    100 |       1 |      3
(5 rows)
```

Dữ liệu có 21 dòng và 5 phân mảnh ta có 21%5 =1. Vì vậy khi thực hiện insert ta có dữ liệu sẽ được chèn vào rrobin_part0 (phân mảnh đầu tiên).

6.2. Với ratings.dat

Số dòng dữ liệu: Khoảng 10 triệu dòng.

Với dữ liệu lớn này, hệ thống vẫn thực hiện phân mảnh thành công.

- **Range Partition**:

  Đọc file ratings.dat

  ```
  dds_assgn1=# select count(*) from ratings;
    count
  ----------
   10000054
  (1 row)
  ```

  Khi thực hiện phân mảnh và chèn thêm 1 dòng ta có

  range_part0: 479168 rows

  range_part1: 908584 rows

  range_part2: 2726854 rows

  range_part3: 3755614 rows

  range_part4: 2129834 rows

```
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
---------
 2726854
(1 row)

dds_assgn1=# select count(*) from range_part3;
  count
---------
 3755614
(1 row)

dds_assgn1=# select count(*) from range_part4;
  count
---------
 2129834
```

Sau khi thêm dữ liệu rows (100,1,3) ta có dữ liệu của bảng ratings và range_part2 được cập nhật

```
dds_assgn1=# select count(*) from range_part2;
  count
---------
 2726855
(1 row)
```

```
dds_assgn1=# select count(*) from ratings;
  count
----------
 10000055
(1 row)
```

dữ liệu đã được tăng thêm 1 dòng.

Nhận xét: số lượng dữ liệu giữa các phân mảnh range không đều

- **Round-Robin Partition**: Dữ liệu được phân phối đều đặn giữa các phân mảnh, nhờ cơ chế phân phối tuần tự theo chỉ số dòng.

  Khi thực hiện phân mảnh ta có số dòng trên lý thuyết của round robin sẽ là 10000054//5 = 2000010 rows và 10000054%5 = 4 rows

  -> rrobin_part(i) với i từ 0 đến 3 sẽ có độ dài là 2000011 dòng và rrobin_part4 sẽ có độ dài là 2000010.

  Thực tế

```
dds_assgn1=# select count(*) from rrobin_part0;
  count
---------
 2000011
(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)
```

Kết quả sau khi test:

```
loadratings function pass!
Press enter to continue...
rangepartition function pass!
rangeinsert function pass!
Press enter to continue...
roundrobinpartition function pass!
roundrobininsert function pass!
```

6.3. Đánh giá hiệu năng hệ thống

LoadRatings:

- Việc sử dụng lệnh copy_from() để nạp hàng loạt bản ghi từ file vào cơ sở dữ liệu giúp tối ưu hóa hiệu năng đáng kể, nhanh hơn nhiều so với việc INSERT từng dòng một.

Phân mảnh và chèn dữ liệu:

- Các hàm phân mảnh (Range_Partition, RoundRobin_Partition) và hàm chèn (Range_Insert, RoundRobin_Insert) thực thi chính xác, ổn định.

- Việc sử dụng metadata để lưu trữ chỉ số vòng lặp giúp đảm bảo tính chính xác và khả năng tiếp tục phân phối dữ liệu theo đúng thứ tự, ngay cả sau khi hệ thống khởi động lại.

6.4. So sánh phương pháp phân mảnh

Tiêu chí	Range	Round-robin
Phân phối	Theo khoảng rating	Tuần tự (modulo N) dựa vào số thứ tự dòng
Ưu điểm	Khi biết giá trị ratings truy vấn select sẽ thực hiện nhanh hơn.	Phân phối đều, dễ triển khai.
Nhược điểm	Dễ gặp phải vấn đề data skew (số lượng dữ liệu giữa các phân mảnh không đều)	Không tối ưu cho truy vấn cụ thể
Ứng dụng	Phân tích rating	Cân bằng tải

7. Kết Luận

Trong quá trình thực hiện bài tập lớn môn Cơ sở dữ liệu phân tán, nhóm chúng em đã nghiên cứu, thiết kế và lập trình một hệ thống mô phỏng quá trình phân mảnh dữ liệu ngang – một trong những kỹ thuật quan trọng của hệ quản trị cơ sở dữ liệu phân tán. Hệ thống được xây dựng bằng ngôn ngữ Python kết hợp với PostgreSQL, áp dụng hai phương pháp phân mảnh phổ biến là Range Partitioning và Round-Robin Partitioning.

Cụ thể, nhóm đã xây dựng các hàm:

- **LoadRatings**: nạp dữ liệu từ file ratings.dat vào bảng ratings, xử lý định dạng dữ liệu chuẩn.
- **Range_Partition** và **RoundRobin_Partition**: chia bảng chính thành nhiều bảng nhỏ (phân mảnh) theo phương pháp tương ứng.
- **Range_Insert** và **RoundRobin_Insert**: đảm bảo dữ liệu mới được chèn đúng vào bảng chính và phân phối chính xác đến các phân mảnh tương ứng.

Điểm nổi bật trong thiết kế là việc sử dụng bảng metadata để lưu thông tin số lượng phân mảnh của từng kiểu, từ đó giúp cho các hàm insert trở nên linh hoạt, không phụ thuộc vào hard-code và có khả năng mở rộng trong tương lai.

Kết quả kiểm thử cho thấy:

- Các phân mảnh được tạo đúng theo logic đã đề ra.
- Dữ liệu sau khi chèn được phân phối chính xác, không trùng lặp và không mất mát.
- Hệ thống hoạt động ổn định và hiệu quả ngay cả với dữ liệu lớn.

Thông qua bài tập lớn này, nhóm không chỉ củng cố các kiến thức lý thuyết đã học trên lớp về cơ sở dữ liệu phân tán, mà còn nâng cao kỹ năng lập trình, tư duy hệ thống, và phối hợp làm việc nhóm trong dự án thực tế. Đây là nền tảng quan trọng giúp nhóm có thể tiếp cận các bài toán lớn hơn trong lĩnh vực xử lý dữ liệu phân tán, hệ thống thông tin quy mô lớn, hoặc các ứng dụng thực tiễn trong doanh nghiệp và công nghiệp 4.0.

Nhóm xin chân thành cảm ơn thầy/cô đã tạo điều kiện và hướng dẫn để nhóm hoàn thành tốt bài tập lớn này.

Phân chia công việc

- **Nguyễn Phúc Hưng**:
  - Triển khai hàm LoadRatings và Range_Partition.
  - Kiểm tra dữ liệu đầu vào và đảm bảo schema đúng yêu cầu.
  - Viết báo cáo
- **Trần Văn Đức**:
  - Triển khai hàm RoundRobin_Partition và RoundRobin_Insert.
  - Kiểm tra code trên máy ảo Ubuntu và xử lý lỗi runtime.
  - Đóng góp ý tưởng tối ưu hiệu suất.
- **Nguyễn Đức Mạnh**:
  - Triển khai hàm Range_Insert và thiết lập bảng partitionmetadata.
  - Viết báo cáo.

Mỗi thành viên đều tham gia kiểm tra chéo code, chạy thử trên dữ liệu mẫu, và thảo luận để thống nhất cách giải quyết.