

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**  
**KHOA CÔNG NGHỆ THÔNG TIN 1**



**BÁO CÁO BÀI TẬP LỚN**  
**CƠ SỞ DỮ LIỆU PHÂN TÁN**

**ĐỀ TÀI**  
**MÔ PHỎNG PHÂN MẢNH DỮ LIỆU TRÊN POSTGRESQL**  
**VỚI MOVIELENS**

**Giảng viên hướng dẫn: TS. Kim Ngọc Bách**

**Thành viên tham gia: Trần Đức Dũng - B22DCCN139**  
**Nguyễn Cao Duy - B22DCCN150**  
**Nguyễn Đức Duy - B22DCCN151**

**Nhóm: 18**

**Lớp: 09**

**Hà Nội - 2025**

## LỜI CẢM ƠN

Đầu tiên, chúng em xin gửi lời cảm ơn sâu sắc đến Học viện nghệ Bưu chính Viễn thông và khoa CNTT1 đã đưa môn học Cơ sở dữ liệu phân tán vào trong chương trình giảng dạy. Đặc biệt, chúng em xin gửi lời cảm ơn sâu sắc đến thầy Kim Ngọc Bách đã dạy dỗ, rèn luyện và truyền đạt những kiến thức quý báu cho chúng em trong suốt thời gian học tập vừa qua.

Trong thời gian được tham dự lớp học của thầy, chúng em đã được tiếp thu thêm nhiều kiến thức bổ ích, học tập được tinh thần làm việc hiệu quả, nghiêm túc. Đây thực là những điều rất cần thiết cho quá trình học tập và công tác sau này của em. Thêm vào đó, nhờ sự dẫn dắt và chỉ bảo của thầy, chúng em đã thực hiện được một đề tài bài tập lớn hoàn chỉnh cho môn học này, chúng em rất biết ơn điều đó.

Em xin chân thành cảm ơn, chúc thầy luôn khỏe mạnh và tiếp tục đạt được nhiều thành công trong cuộc sống ạ!

## PHÂN CHIA CÔNG VIỆC

Họ Tên	Nhiệm Vụ
Trần Đức Dũng	Tối ưu hàm Loadratings, tái cấu trúc file Assignment1Tester.py
Nguyễn Cao Duy	Tối ưu Range Partitioning, Range Insert
Nguyễn Đức Duy	Tối ưu Round Robin Partitioning, Round Robin Insert

## MỤC LỤC

LỜI CẢM ƠN.....	1
PHÂN CHIA CÔNG VIỆC .....	2
CHƯƠNG 1: GIỚI THIỆU VÀ CẤU HÌNH HỆ THỐNG.....	4
1.1. Giới thiệu dự án .....	4
1.2. Yêu cầu hệ thống và cài đặt .....	4
1.3. Cấu hình dự án .....	5
1.4. Kiểm tra cấu hình với test_connection.py .....	5
1.5. Chạy test kết nối .....	6
CHƯƠNG 2: TRIỂN KHAI CÁC CHỨC NĂNG CHÍNH .....	8
2.1. Tổng quan Module Interface.py .....	8
2.2. Hàm Load dữ liệu (loadratings) .....	8
2.3. Range Partitioning Implementation .....	10
2.4. Round Robin Partitioning Implementation .....	12
2.5. Round Robin Insert .....	14
2.6. Range Insert .....	14
2.7. Module Assignment1Tester.py - Main Test Runner .....	16
CHƯƠNG 5: DEMO VÀ KẾT QUẢ .....	22
5.1. Chạy demo hệ thống .....	22
5.2. Kết quả kiểm thử .....	25
5.3. Đánh giá hiệu suất .....	25
5.4. Kết luận và hướng phát triển .....	25

# CHƯƠNG 1: GIỚI THIỆU VÀ CẤU HÌNH HỆ THỐNG

## 1.1. Giới thiệu dự án

### 1.1.1. Tổng quan về Database Partitioning

Database Partitioning (phân vùng cơ sở dữ liệu) là một kỹ thuật quan trọng trong việc quản lý và tối ưu hóa cơ sở dữ liệu lớn. Thay vì lưu trữ tất cả dữ liệu trong một bảng duy nhất, kỹ thuật này chia dữ liệu thành nhiều phân vùng nhỏ hơn dựa trên các tiêu chí nhất định.

Lợi ích của Database Partitioning:

- Cải thiện hiệu suất truy vấn: Queries chỉ cần scan các partition liên quan
- Quản lý dữ liệu tốt hơn: Dễ dàng backup, restore từng partition
- Parallel processing: Có thể xử lý đồng thời trên nhiều partition
- Scalability: Dễ dàng mở rộng hệ thống

### 1.1.2. Mục tiêu dự án

Dự án này triển khai hai kỹ thuật partitioning chính:

- Range Partitioning: Phân vùng dựa trên khoảng giá trị (rating 0-5)
- Round Robin Partitioning: Phân vùng tuần tự theo chỉ số dòng

Dataset sử dụng: MovieLens ratings.dat với ~10 triệu records Platform: PostgreSQL + Python 3.12.5

## 1.2. Yêu cầu hệ thống và cài đặt

### 1.2.1. Yêu cầu phần cứng và phần mềm

Yêu cầu tối thiểu:

- OS: Windows 10/11, Linux, hoặc macOS
- RAM: 8GB (khuyến nghị 16GB)
- Storage: 10GB dung lượng trống
- CPU: Intel i5 hoặc tương đương

Phần mềm cần thiết:

- Python 3.12.5
- PostgreSQL 15+
- psycopg2
- Visual Studio Code

### 1.2.3. Cài đặt PostgreSQL

Chạy installer

- Password: Đặt password là "1234" (theo config mặc định)
- Port: 5432 (mặc định)
- Locale: Default locale

#### 1.2.4. Cài đặt Python dependencies

Cài đặt psycopg2

### 1.3. Cấu hình dự án

#### 1.3.1. Cấu hình kết nối database

File: Interface.py - Connection Configuration

```
def getopenconnection(user='postgres', password='1234', dbname='postgres'):
    return psycopg2.connect("dbname='" + dbname + "' user='" + user + "' host='localhost' password='" + password + "'")
```

- user='postgres': User mặc định của PostgreSQL
- password='1234': Password đã đặt trong quá trình cài đặt
- dbname='postgres': Database mặc định, sẽ chuyển sang 'dds\_assgn1' sau
- host='localhost': Database chạy local
- Return: psycopg2 connection object để thực hiện các operations

#### 1.4. Kiểm tra cấu hình với test\_connection.py

##### 1.4.1. Luồng xử lý chính

Bước 1: Test kết nối cơ bản

```
def test_connection():
    """Test PostgreSQL connection"""
    try:
        # Test basic connection
        conn = getopenconnection()
        print("PostgreSQL connection successful")
```

- Tạo connection đến database 'postgres' mặc định
- Nếu thành công, in thông báo kết nối OK

Bước 2: Set isolation level và tạo cursor

```
# Check/create database
conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
cur = conn.cursor()
```

- AUTOCOMMIT mode: Mỗi SQL command tự động commit
- cursor(): Object để execute SQL commands

Bước 3: Kiểm tra/tạo database dds\_assgn1

```
cur.execute("SELECT COUNT(*) FROM pg_catalog.pg_database WHERE datname='dds_assgn1'")

if cur.fetchone()[0] == 0:
    cur.execute("CREATE DATABASE dds_assgn1")
    print("Database 'dds_assgn1' created")
else:
    print("Database 'dds_assgn1' exists")
```

- Query pg\_catalog.pg\_database: System catalog chứa thông tin databases
- fetchone()[0]: Lấy count result
- Tạo database nếu chưa tồn tại

Bước 4: Test kết nối đến database mới

```
# Test assignment database connection
conn = getopenconnection(dbname='dds_assgn1')
print("Connected to dds_assgn1 database")
conn.close()
```

- Tạo connection mới đến database 'dds\_assgn1'
- Verify kết nối thành công
- Đóng connection

#### 1.4.2. Exception handling

```
except psycopg2.OperationalError as e:
    print(f"Connection failed: {e}")
    return False

except Exception as e:
    print(f"Error: {e}")
    return False
```

- OperationalError: Lỗi kết nối database (wrong password, host không tìm thấy)
- General Exception: Các lỗi khác (import error, syntax error)
- Return False: Indicate test failed

### 1.5. Chạy test kết nối

#### 1.5.1. Thực thi test\_connection.py

```
PS F:\testAPI\bai_tap_lon_CSDL_phan_tan\code> python test_connection.py
PostgreSQL connection successful
Database 'dds_assgn1' exists
Connected to dds_assgn1 database
```



## CHƯƠNG 2: TRIỂN KHAI CÁC CHỨC NĂNG CHÍNH

### 2.1. Tổng quan Module Interface.py

#### 2.1.1. Cấu trúc và các hàm chính

Module Interface.py chứa tất cả các core functions của hệ thống Database Partitioning:

Core Functions:

- loadratings(): Load dữ liệu từ file vào database
- rangepartition(): Phân vùng theo khoảng rating
- roundrobinpartition(): Phân vùng round robin
- rangeinsert(): Insert vào range partition
- roundrobininsert(): Insert vào round robin partition

#### 2.1.2. Imports và dependencies

```
import psycopg2
from io import StringIO
```

- psycopg2: PostgreSQL adapter cho Python
- StringIO: Tạo buffer trong memory để optimize bulk operations

### 2.2. Hàm Load dữ liệu (loadratings)

Hàm loadratings nạp dữ liệu từ một tệp văn bản (định dạng userID::movieID::rating::timestamp) vào một bảng trong cơ sở dữ liệu PostgreSQL.

```
def loadratings(ratingtablename, ratingsfilepath, openconnection):
    """
    Function to load data in @ratingsfilepath file to a table called @ratingtablename.
    """
    con = openconnection
    cur = con.cursor()
```

- Gán đối tượng openconnection vào biến con để sử dụng.
- Tạo con trỏ (cursor) bằng con.cursor() để thực thi các lệnh SQL.

```
# Drop table if exists and create new one
cur.execute(f"DROP TABLE IF EXISTS {ratingstablename}")
cur.execute(f"""
    CREATE TABLE {ratingstablename} (
        userid integer,
        movieid integer,
        rating float
    )
""")
```

- Lệnh DROP TABLE IF EXISTS xóa bảng có tên ratingstablename nếu đã tồn tại, đảm bảo không có xung đột.
- Lệnh CREATE TABLE tạo bảng mới với ba cột:
  - + userid (integer)
  - + movieid (integer)
  - + rating (float)

```
# Read and process file in chunks
chunk_size = 100000
with open(ratingsfilepath, 'r') as f:
    while True:
        chunk = []
        for _ in range(chunk_size):
            line = f.readline()
            if not line:
                break
            parts = line.strip().split '::' # File format uses :: as delimiter
            if len(parts) >= 3: # userID::movieID::rating::timestamp
                userid, movieid, rating = parts[0], parts[1], parts[2]
                chunk.append(f"{userid}\t{movieid}\t{rating}\n") # Use tab delimiter

        if not chunk:
            break
```

- Định nghĩa chunk\_size = 100000 để đọc tệp theo khối, tối ưu hóa bộ nhớ.
- Mở tệp ratingsfilepath ở chế độ đọc.
- Vòng lặp while True đọc từng khối dữ liệu:
  - ◆ Đọc từng dòng bằng f.readline(). Nếu hết dữ liệu (line rỗng), thoát vòng lặp nội.
  - ◆ Tách dòng bằng dấu :: để lấy userid, movieid, rating.
  - ◆ Thêm dữ liệu vào danh sách chunk dưới dạng chuỗi, phân tách bằng tab (\t).
- Nếu chunk rỗng, thoát vòng lặp chính.

```
# Create a string buffer for the chunk and use COPY
buffer = StringIO('').join(chunk)
cur.copy_from(buffer, ratingtablename, sep='\t', columns=('userid', 'movieid', 'rating'))
con.commit()
```

- Tạo bộ đệm StringIO từ danh sách chunk để lưu dữ liệu tạm thời.
- Sử dụng `cur.copy_from` để nạp dữ liệu vào bảng:
  - ♦ `sep='\t'`: Dấu phân tách là tab.
  - ♦ `columns=('userid', 'movieid', 'rating')`: Chỉ định các cột đích.
- Gọi `con.commit()` để xác nhận giao dịch, lưu dữ liệu vào cơ sở dữ liệu.

```
cur.close()
```

- Đóng con trỏ để giải phóng tài nguyên sau khi hoàn tất.

## 2.3. Range Partitioning Implementation

### 2.3.1. Mục đích của hàm

Hàm `rangepartition(ratingtablename, numberofpartitions, openconnection)` được thiết kế để thực hiện phân mảnh ngang theo khoảng giá trị (range partitioning) trên bảng Ratings. Cụ thể, hàm chia dữ liệu từ bảng gốc thành N bảng con, mỗi bảng tương ứng với một khoảng giá trị đều nhau của cột Rating. Mục tiêu là:

Tối ưu hiệu suất truy vấn bằng cách giảm khối lượng dữ liệu cần quét.  
Hỗ trợ các chiến lược lưu trữ và phân tán dữ liệu hiệu quả hơn.

### 2.3.2. Cấu trúc tổng quan của hàm

Hàm nhận vào ba đối số:

- `ratingtablename`: tên bảng dữ liệu gốc chứa các bản ghi đánh giá.
- `numberofpartitions`: số lượng phân mảnh cần tạo.
- `openconnection`: đối tượng kết nối đến cơ sở dữ liệu PostgreSQL.

Hàm được thiết kế gồm ba bước chính:

1. Tính toán khoảng giá trị mỗi phân mảnh.
2. Tạo bảng cho từng phân mảnh.
3. Phân phối dữ liệu từ bảng gốc vào từng bảng phân mảnh dựa trên điều kiện lọc.

### 2.3.3. Phân tích chi tiết từng phần

#### 1. Tính toán khoảng phân vùng

```
delta = 5.0 / numberofpartitions
```

Vì giá trị Rating nằm trong khoảng [0.0, 5.0], nên mỗi phân mảnh sẽ có chiều rộng là  $\text{delta} = 5 / N$ .

Ví dụ: Với `numberofpartitions = 3`, ta có:

- `range_part0`: [0.0, 1.666...]
- `range_part1`: (1.666..., 3.333...]
- `range_part2`: (3.333..., 5.0]

## 2. Tạo bảng phân vùng

```
create_tables_sql = ''; '.join([
    f"CREATE TABLE IF NOT EXISTS range_part{i} (userid integer, movieid integer, rating float)"
    for i in range(numberofpartitions)
])
cur.execute(create_tables_sql)
```

- Duyệt từ  $i = 0$  đến `numberofpartitions - 1`, tạo ra chuỗi lệnh SQL tạo các bảng `range_part0`, `range_part1`,...
- Mỗi bảng có cấu trúc giống bảng gốc: `userid`, `movieid`, `rating`.
- Dùng `CREATE TABLE IF NOT EXISTS` để đảm bảo không bị lỗi nếu bảng đã tồn tại.

## 3. Xóa dữ liệu cũ trong các bảng phân mảnh

```
cur.execute('; '.join([f"TRUNCATE TABLE range_part{i}" for i in range(numberofpartitions)]))
```

- Đây là một bước quan trọng nhằm đảm bảo tính toàn vẹn dữ liệu. Trước khi ghi dữ liệu mới vào các phân mảnh, các bảng này được xóa trắng để loại bỏ dữ liệu từ các lần phân mảnh trước đó.

## 4. Phân phối dữ liệu vào từng phân mảnh

Dữ liệu được đưa vào từng bảng phân mảnh theo vòng lặp sau:

```
for i in range(numberofpartitions):
    minRange = i * delta
    maxRange = minRange + delta
```

Giá trị `minRange` và `maxRange` xác định khoảng giá trị mà phân mảnh thứ  $i$  sẽ nhận dữ liệu.

- `minRange = i * delta`: Giá trị nhỏ nhất của partition
- `maxRange = minRange + delta`: Giá trị lớn nhất của partition

Sau đó, sử dụng câu lệnh SQL để chèn dữ liệu phù hợp:

Trường hợp phân mảnh đầu tiên ( $i == 0$ ):

```
if i == 0:
    cur.execute("""
        INSERT INTO range_part{}
        SELECT userid, movieid, rating
        FROM {}
        WHERE rating >= {} AND rating <= {}
    """.format(i, ratingtablename, minRange, maxRange))
```

- Bao gồm cả điểm đầu của khoảng ( $\geq$ ) để không bỏ sót giá trị nhỏ nhất (thường là 0.0).

Các phân mảnh còn lại:

```
cur.execute("""
    INSERT INTO range_part{}
    SELECT userid, movieid, rating
    FROM {}
    WHERE rating > {} AND rating <= {}
    """.format(i, ratingtablename, minRange, maxRange))
```

- Sử dụng  $>$  để đảm bảo không trùng lặp biên với phân mảnh trước.
- Giá trị biên  $\text{rating} == \text{maxRange}$  vẫn được bao gồm.

Ví dụ cụ thể: Giả sử  $\text{numberofpartitions} = 5$ :

- $\text{delta} = 5.0/5 = 1.0$
- Các partition sẽ được tạo như sau:
- $\text{range\_part0}$ :  $0.0 \leq \text{rating} \leq 1.0$
- $\text{range\_part1}$ :  $1.0 < \text{rating} \leq 2.0$
- $\text{range\_part2}$ :  $2.0 < \text{rating} \leq 3.0$
- $\text{range\_part3}$ :  $3.0 < \text{rating} \leq 4.0$
- $\text{range\_part4}$ :  $4.0 < \text{rating} \leq 5.0$

Ưu điểm của phương pháp này:

1. Phân phối dữ liệu dựa trên giá trị rating, giúp truy vấn theo khoảng rating hiệu quả
2. Dữ liệu được phân phối đều theo giá trị
3. Dễ dàng mở rộng bằng cách thêm partition mới
4. Tối ưu cho các truy vấn theo khoảng giá trị

Nhược điểm:

1. Có thể dẫn đến phân phối không đều nếu dữ liệu tập trung ở một khoảng giá trị
2. Cần tính toán lại partition khi thêm/xóa partition
3. Chi phí chuyển dữ liệu giữa các partition khi thay đổi cấu trúc

## 2.4. Round Robin Partitioning Implementation

Đầu tiên, ta mở kết nối với cơ sở dữ liệu và khởi tạo biến `cur` để thực thi câu lệnh SQL.

```

con = openconnection
cur = con.cursor()
delta = 5.0 / numberofpartitions

```

Tiếp đến, ta tạo các bảng phân mảnh round robin (nếu chưa có) với số lượng bằng "numberofpartition" bằng cách sử dụng CREATE TABLE và IF NOT EXIST.

```

create_tables_sql = '; '.join([
    f"CREATE TABLE IF NOT EXISTS range_part{i} (userid integer, movieid integer, rating float)"
    for i in range(numberofpartitions)
])
cur.execute(create_tables_sql)

```

Sau đó ta clear hết dữ liệu trong tất cả các bảng phân mảnh (nếu có) bằng TRUNCATE TABLE.

```

cur.execute('; '.join([f"TRUNCATE TABLE range_part{i}" for i in range(numberofpartitions)]))

```

Cuối cùng ta phân mảnh dữ liệu từ bảng gốc theo quy tắc: bảng phân mảnh = chỉ số mod tổng số bảng phân mảnh.

```

for i in range(numberofpartitions):
    minRange = i * delta
    maxRange = minRange + delta
    if i == 0:
        cur.execute("""
            INSERT INTO range_part{}
            SELECT userid, movieid, rating
            FROM {}
            WHERE rating >= {} AND rating <= {}
            """.format(i, ratingtablename, minRange, maxRange))
    else:
        cur.execute("""
            INSERT INTO range_part{}
            SELECT userid, movieid, rating
            FROM {}
            WHERE rating > {} AND rating <= {}
            """.format(i, ratingtablename, minRange, maxRange))

```

## 2.5. Round Robin Insert

Đầu tiên, giống ở hàm `rrpartition`, ta mở kết nối với cơ sở dữ liệu và khởi tạo biến `cur` để thực thi câu lệnh SQL.

Sau đó, dòng dữ liệu mới sẽ được thêm vào bảng gốc sử dụng `INSERT` với prepared statement.

```
# Insert into main table
cur.execute("""
    INSERT INTO {} (userid, movieid, rating)
    VALUES (%s, %s, %s)
""".format(ratingstablename), (userid, itemid, rating))
```

Tiếp đến, ta cập nhật tổng số bản ghi trong bảng gốc rồi tính vị trí bảng phân mảnh mà dữ liệu mới sẽ được thêm vào theo quy tắc ở trên.

```
# Get total count
cur.execute("SELECT COUNT(*) FROM {}".format(ratingstablename))
total_rows = cur.fetchone()[0]

# Calculate partition index
numberofpartitions = count_partitions('rrobin_part', openconnection)
index = (total_rows - 1) % numberofpartitions
```

Cuối cùng, khi có vị trí bảng phân mảnh, ta thêm dữ liệu mới vào bảng tương ứng.

```
cur.execute("""
    INSERT INTO rrobin_part{} (userid, movieid, rating)
    VALUES (%s, %s, %s)
""".format(index), (userid, itemid, rating))
```

## 2.6. Range Insert

### 2.6.1. Mục tiêu và vai trò của hàm

Hàm `rangeinsert` thực hiện nhiệm vụ chèn một bản ghi dữ liệu mới vào bảng chính và đồng thời vào đúng phân mảnh theo khoảng giá trị đã được tạo ra trước đó bởi hàm phân mảnh `rangepartition`.

Cụ thể, dữ liệu mới sẽ được xác định vị trí phân mảnh dựa trên giá trị `rating`, rồi được chèn vào bảng chính (`ratingstablename`) và vào bảng phân mảnh tương ứng (`range_part{i}`).

### 2.6.2. Cấu trúc và đối số đầu vào

```
def rangeinsert(ratingtablename, userid, itemid, rating, openconnection):
```

- ratingtablename: tên bảng chính đang lưu dữ liệu gốc.
- userid, itemid, rating: thông tin bản ghi mới cần chèn.
- openconnection: đối tượng kết nối cơ sở dữ liệu PostgreSQL.

Hàm không trả về giá trị, nhưng thực hiện chèn dữ liệu đồng thời vào hai bảng thông qua truy vấn SQL bên trong giao dịch (transaction).

### 2.6.3. Chi tiết thuật toán triển khai

#### 1. Lấy số lượng phân mảnh hiện có

```
numberofpartitions = count_partitions('range_part', openconnection)
```

- Hàm phụ count\_partitions() sẽ đếm số bảng bắt đầu bằng tiền tố "range\_part" trong hệ thống.
- Điều này giúp xác định được tổng số phân mảnh hiện hành, bất kể người dùng tạo bao nhiêu.

#### 2. Tính toán khoảng phân mảnh tương ứng

```
delta = 5.0 / numberOfpartitions
index = int(rating / delta)
if rating % delta == 0 and index != 0:
    index -= 1
```

- Tương tự như trong hàm rangepartition, đoạn này chia đều khoảng giá trị [0.0, 5.0] cho N phân mảnh có chiều rộng delta.
- Sử dụng int(rating / delta) để xác định chỉ số phân mảnh (index) mà giá trị rating rơi vào.

Điều kiện điều chỉnh với điểm biên

- Nếu rating là số chia hết cho delta, thì cần giảm index đi 1 (trừ trường hợp index = 0).
- Mục đích là để đảm bảo tính tương thích với quy ước phân mảnh trong hàm rangepartition, trong đó:
  - ♦ range\_part0 chứa [min, max] dùng >= và <=
  - ♦ range\_part1+ chứa (min, max] dùng > và <=
- Nếu không có bước này, một số bản ghi ở điểm biên sẽ rơi vào sai phân mảnh.

#### 3. Thực hiện chèn dữ liệu



```
cur.execute("""
    BEGIN;
    INSERT INTO {} (userid, movieid, rating)
    VALUES (%s, %s, %s);
    INSERT INTO range_part{} (userid, movieid, rating)
    VALUES (%s, %s, %s);
    COMMIT;
""".format(ratingstablename, index),
(userid, itemid, rating, userid, itemid, rating))
```

- Chèn vào bảng chính và bảng phân mảnh tương ứng, trong một giao dịch duy nhất (BEGIN...COMMIT).
- Mục tiêu là đảm bảo tính toàn vẹn dữ liệu, nếu chèn vào bảng phân mảnh thất bại thì sẽ rollback toàn bộ.
- Việc gộp cả hai lệnh INSERT trong cùng một transaction giúp đồng bộ dữ liệu giữa bảng chính và phân mảnh.

#### 2.6.4. Đặc điểm kỹ thuật và đánh giá

Ưu điểm của phương pháp này:

1. Đảm bảo tính toàn vẹn dữ liệu thông qua transaction
2. Tự động xác định partition phù hợp dựa trên giá trị rating
3. Xử lý đúng các trường hợp đặc biệt (rating chia hết cho delta)
4. Hiệu quả vì chỉ cần chèn vào một partition

Nhược điểm:

1. Cần tính toán index mỗi lần chèn
2. Có thể gặp vấn đề về hiệu suất nếu số lượng partition lớn
3. Cần đảm bảo số lượng partition không thay đổi trong quá trình chèn

## 2.7. Module Assignment1Tester.py - Main Test Runner

### 2.7.1. Import các thư viện và mô-đun

```
import psycopg2
import traceback
import testHelper
import Interface as MyAssignment
import time
```

- psycopg2: Thư viện để kết nối và thao tác với cơ sở dữ liệu PostgreSQL.
- traceback: Hỗ trợ in thông tin chi tiết về lỗi (stack trace) nếu xảy ra ngoại lệ.
- testHelper: Mô-đun tùy chỉnh chứa các hàm kiểm thử (ví dụ: createdb, testloadratings, deleteAllPublicTables, v.v.).
- Interface as MyAssignment: Nhập mô-đun Interface (chứa các hàm như getopenconnection, loadratings, v.v.) với bí danh MyAssignment.

- time: Thư viện để đo thời gian thực thi các tác vụ.

### 2.7.2. Định nghĩa hằng số

```
# Constants
DATABASE_NAME = 'dds_assgn1'
RATINGS_TABLE = 'ratings'
RANGE_TABLE_PREFIX = 'range_part'
RROBIN_TABLE_PREFIX = 'rrobin_part'
INPUT_FILE_PATH = 'ratings.dat'
ACTUAL_ROWS_IN_INPUT_FILE = 10000054
```

- DATABASE\_NAME = 'dds\_assgn1': Tên cơ sở dữ liệu dùng để kiểm thử.
- RATINGS\_TABLE = 'ratings': Tên bảng chính chứa dữ liệu đánh giá phim.
- RANGE\_TABLE\_PREFIX = 'range\_part': Tiền tố cho các bảng phân mảnh theo khoảng (range partitioning), ví dụ: range\_part0, range\_part1.
- RROBIN\_TABLE\_PREFIX = 'rrobin\_part': Tiền tố cho các bảng phân mảnh theo vòng lặp (round-robin partitioning), ví dụ: rrobin\_part0, rrobin\_part1.
- INPUT\_FILE\_PATH = 'ratings.dat': Đường dẫn đến tệp dữ liệu đầu vào chứa đánh giá phim.
- ACTUAL\_ROWS\_IN\_INPUT\_FILE = 10000054: Số dòng dữ liệu thực tế trong tệp, dùng để kiểm tra tính đúng đắn khi nạp dữ liệu.

### 2.7.3. Hàm print\_progress

```
def print_progress(message, indent=0):
    """Print progress message with timestamp and indentation"""
    print(f"[{time.strftime('%H:%M:%S')}] {' ' * indent}{message}")
```

Mục đích: In thông điệp tiến trình với thời gian và mức thụt đầu dòng để theo dõi trạng thái thực thi.

- Tham số:
  - ◆ message: Chuỗi thông điệp cần in.
  - ◆ indent=0: Mức thụt đầu dòng (mặc định là 0, có thể tăng để tạo thụt lề).
- Sử dụng time.strftime('%H:%M:%S') để lấy thời gian hiện tại theo định dạng giờ:phút:giây.

Giúp theo dõi tiến trình thực thi chương trình một cách trực quan, đặc biệt hữu ích khi kiểm thử các tác vụ dài.

### 2.7.4. Hàm verify\_partition\_content

```
def verify_partition_content(conn, prefix, number_of_partitions):
    """Verify content of partition tables"""
    cur = conn.cursor()
    total_rows = 0
    print_progress(f"Verifying {prefix} partition tables:")
    for i in range(number_of_partitions):
        table_name = f"{prefix}{i}"
        cur.execute(f"SELECT COUNT(*) FROM {table_name}")
        count = cur.fetchone()[0]
        total_rows += count
        print_progress(f"- {table_name}: {count} rows", indent=1)

    cur.execute(f"SELECT COUNT(*) FROM {RATINGS_TABLE}")
    original_count = cur.fetchone()[0]
    print_progress(f"Total rows: partitions={total_rows}, original={original_count}")
    print_progress("Partition content passed!" if total_rows == original_count else "Partition content failed!")
    cur.close()
```

Mục đích: Kiểm tra tính toàn vẹn của các bảng phân mảnh bằng cách đếm số dòng trong các bảng phân mảnh và so sánh với bảng gốc.

Chi tiết:

1. Khởi tạo con trỏ và biến đếm:
  1. Tạo con trỏ (cursor) từ đối tượng kết nối conn.
  2. Khởi tạo total\_rows = 0 để đếm tổng số dòng trong các bảng phân mảnh.
2. Kiểm tra từng bảng phân mảnh:
  1. Vòng lặp for i in range(number\_of\_partitions) duyệt qua các bảng phân mảnh (tên bảng: prefix + i, ví dụ: range\_part0).
  2. Thực thi SELECT COUNT(\*) để lấy số dòng trong bảng.
  3. Cộng số dòng vào total\_rows và in thông tin bằng print\_progress.
3. So sánh với bảng gốc:
  1. Thực thi SELECT COUNT(\*) trên bảng gốc (RATINGS\_TABLE) để lấy số dòng.
  2. So sánh total\_rows (tổng dòng trong các bảng phân mảnh) với original\_count (số dòng bảng gốc).
  3. In kết quả kiểm tra: "passed" nếu bằng nhau, "failed" nếu không.
4. Đóng con trỏ:
  1. Đóng con trỏ bằng cur.close() để giải phóng tài nguyên.

Vai trò: Đảm bảo rằng quá trình phân mảnh (range hoặc round-robin) giữ nguyên tổng số dòng so với bảng gốc, xác nhận tính toàn vẹn dữ liệu.

#### 2.7.5. Khối try-except và khởi tạo

```
def main():
    try:
        print_progress("Starting test...")
        testHelper.createdb(DATABASE_NAME)
```

- Hàm main là hàm chính điều phối toàn bộ quá trình kiểm thử.
- Sử dụng khối try-except để bắt và xử lý lỗi nếu có.

- In thông điệp bắt đầu kiểm thử bằng `print_progress`.
- Gọi `testHelper.createdb(DATABASE_NAME)` để tạo cơ sở dữ liệu `dds_assgn1` nếu chưa tồn tại.

#### 2.7.6. Thiết lập kết nối và xóa bảng

```
with testHelper.getopenconnection(dbname=DATABASE_NAME) as conn:
    conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    testHelper.deleteAllPublicTables(conn)
```

- Sử dụng `testHelper.getopenconnection` để tạo kết nối đến cơ sở dữ liệu `dds_assgn1`.
- Sử dụng `with` để đảm bảo kết nối được đóng tự động sau khi hoàn tất.
- Đặt chế độ giao dịch tự động (`AUTOCOMMIT`) để các thay đổi được áp dụng ngay lập tức.
- Gọi `testHelper.deleteAllPublicTables(conn)` để xóa tất cả bảng hiện có trong cơ sở dữ liệu, đảm bảo môi trường kiểm thử sạch.

#### 2.7.7. Kiểm thử hàm `loadratings`

```
# Test loadratings
print_progress("Testing loadratings...")
start_time = time.time()
[result, e] = testHelper.testloadratings(MyAssignment, RATINGS_TABLE, INPUT_FILE_PATH, conn, ACTUAL_ROWS_IN_INPUT_FILE)
load_time = time.time() - start_time
print_progress(f"loadratings: {'passed' if result else 'failed'}! ({load_time:.3f} seconds)")
```

- In thông điệp bắt đầu kiểm thử hàm `loadratings`.
- Ghi lại thời gian bắt đầu bằng `time.time()`.
- Gọi `testHelper.testloadratings` để kiểm thử hàm `loadratings` từ mô-đun `MyAssignment`, truyền các tham số:
  - ◆ `RATINGS_TABLE`: Tên bảng.
  - ◆ `INPUT_FILE_PATH`: Đường dẫn tệp dữ liệu.
  - ◆ `conn`: Đối tượng kết nối.
  - ◆ `ACTUAL_ROWS_IN_INPUT_FILE`: Số dòng mong đợi.
- Hàm trả về `[result, e]`: `result` là `True` nếu kiểm thử thành công, `e` là lỗi nếu có.
- Tính thời gian thực thi (`load_time`) và in kết quả (`pass/fail`) cùng thời gian.

#### 2.7.8. Lựa chọn loại phân mảnh

```
# Get partition choice
partition_choice = input("\nChoose partitioning (range/roundrobin): ").strip().lower()
start_time = time.time()
```

- Yêu cầu người dùng nhập lựa chọn phân mảnh: `range` hoặc `roundrobin`.
- Chuẩn hóa đầu vào bằng `.strip().lower()` để loại bỏ khoảng trắng và chuyển thành chữ thường.
- Ghi lại thời gian bắt đầu để đo thời gian phân mảnh và chèn.

#### 2.7.9. Kiểm thử phân mảnh theo khoảng (`range`)

```

if partition_choice == 'range':
    print_progress("Testing RANGE partitioning...")
    print_progress("Creating 5 range partitions...")
    [result, e] = testHelper.testrangepartition(MyAssignment, RATINGS_TABLE, 5, conn, 0, ACTUAL_ROWS_IN_INPUT_FILE)
    if result:
        print_progress("rangepartition passed!")
        verify_partition_content(conn, RANGE_TABLE_PREFIX, 5)
    else:
        print_progress("rangepartition failed!")

    print_progress("Testing range insert...")
    [result, e] = testHelper.testrangeinsert(MyAssignment, RATINGS_TABLE, 100, 2, 3, conn, '2')
    print_progress(f"rangeinsert: {'passed' if result else 'failed'}!")

```

Nếu người dùng chọn range:

- In thông điệp bắt đầu kiểm thử phân mảnh theo khoảng.
- Gọi testHelper.testrangepartition để kiểm thử hàm rangepartition với 5 phân mảnh.
- Nếu kiểm thử thành công (result=True), gọi verify\_partition\_content để kiểm tra số dòng trong các bảng phân mảnh (range\_part0 đến range\_part4).
- In kết quả kiểm thử phân mảnh.
- Kiểm thử hàm rangeinsert bằng testHelper.testrangeinsert, chèn một bản ghi mẫu (userid=100, movieid=2, rating=3) và kiểm tra kết quả.

#### 2.7.10. Kiểm thử phân mảnh vòng tròn (round-robin)

```

elif partition_choice == 'roundrobin':
    print_progress("Testing ROUND ROBIN partitioning...")
    print_progress("Creating 5 roundrobin partitions...")
    [result, e] = testHelper.testroundrobinpartition(MyAssignment, RATINGS_TABLE, 5, conn, 0, ACTUAL_ROWS_IN_INPUT_FILE)
    if result:
        print_progress("roundrobinpartition passed!")
        verify_partition_content(conn, RROBIN_TABLE_PREFIX, 5)
    else:
        print_progress("roundrobinpartition failed!")

    print_progress("Testing roundrobin insert...")
    [result, e] = testHelper.testroundrobininsert(MyAssignment, RATINGS_TABLE, 100, 1, 3, conn, '4')
    print_progress(f"roundrobininsert: {'passed' if result else 'failed'}!")

```

Nếu người dùng chọn roundrobin:

- Tương tự như kiểm thử range, nhưng kiểm thử hàm roundrobinpartition với 5 phân mảnh.
- Nếu thành công, gọi verify\_partition\_content để kiểm tra các bảng rrobin\_part0 đến rrobin\_part4.
- Kiểm thử hàm roundrobininsert với một bản ghi mẫu (userid=100, movieid=1, rating=3) và kiểm tra kết quả.

#### 2.7.11. Xử lý lựa chọn không hợp lệ

```

else:
    print_progress("Invalid choice! Choose 'range' or 'roundrobin'.")
    return

```

Nếu người dùng nhập lựa chọn không phải range hoặc roundrobin, in thông báo lỗi và thoát hàm main.

### 2.7.12. Hiển thị thời gian thực thi

```
# Display total execution time
elapsed_time = time.time() - start_time
print_progress(f"Total partitioning + insert time: {elapsed_time:.3f} seconds")
```

- Tính thời gian thực thi tổng cộng cho quá trình phân mảnh và chèn.
- In thời gian bằng print\_progress với định dạng 3 chữ số thập phân.

### 2.7.13. Xóa bảng

```
# Delete tables
if input('\nPress enter to delete all tables: ') == '':
    print_progress("Deleting all tables...")
    testHelper.deleteAllPublicTables(conn)
    print_progress("Tables deleted.")
```

- Yêu cầu người dùng nhấn Enter để xác nhận xóa tất cả bảng.
- Gọi testHelper.deleteAllPublicTables(conn) để xóa các bảng trong cơ sở dữ liệu.
- In thông báo xác nhận xóa hoàn tất.

### 2.7.14. Xử lý ngoại lệ

```
except Exception:
    print_progress("Error occurred:")
    traceback.print_exc()
```

Nếu có lỗi xảy ra trong khối try, in thông báo lỗi và sử dụng traceback.print\_exc() để in chi tiết stack trace, giúp gỡ lỗi.

## CHƯƠNG 5: DEMO VÀ KẾT QUẢ

### 5.1. Chạy demo hệ thống

#### 1. Kiểm tra kết nối

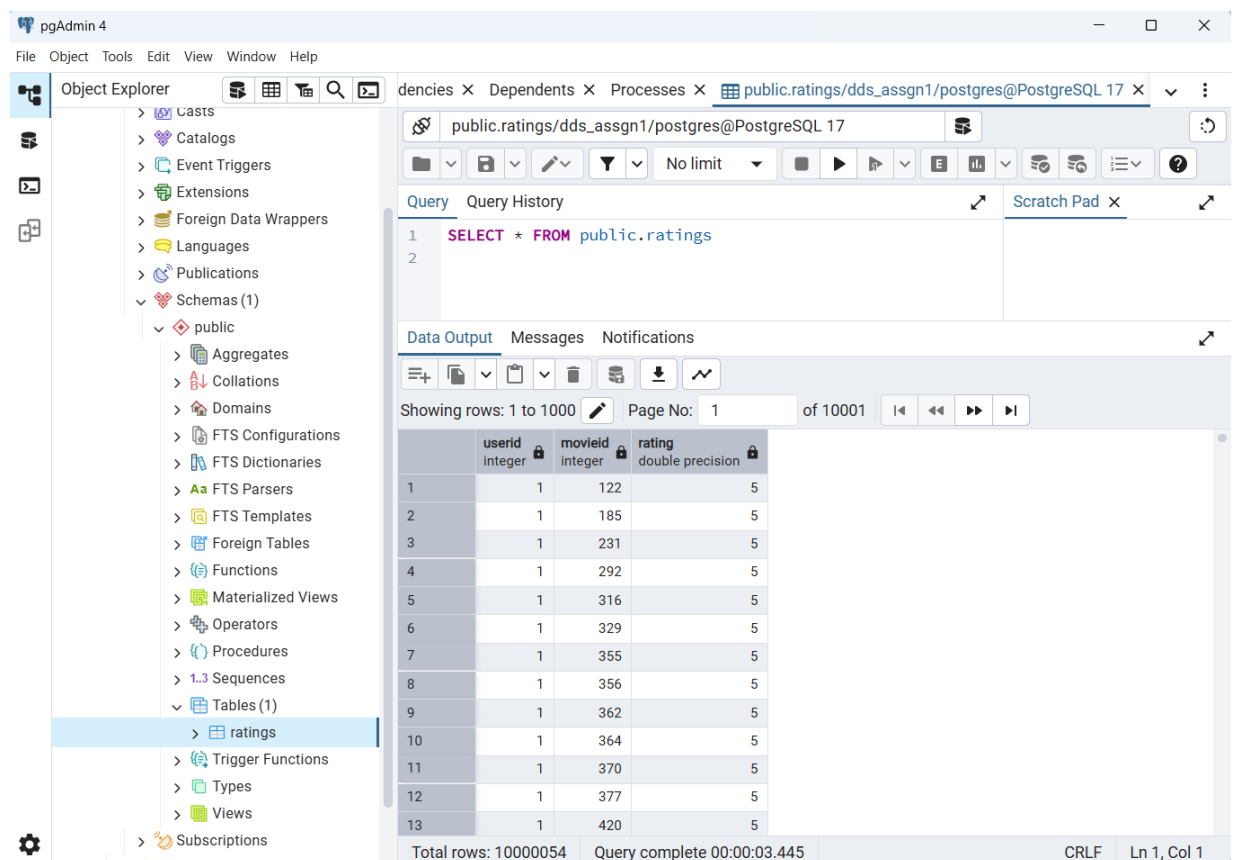
```
python test_connection.py
```

```
PS F:\testAPI\bai_tap_lon_CSDL_phan_tan\code> python test_connection.py
PostgreSQL connection successful
Database 'dds_assgn1' exists
Connected to dds_assgn1 database
```

#### 2. Chạy Main Demo - Load Data

```
python Assignment1Tester.py
```

```
PS F:\testAPI\bai_tap_lon_CSDL_phan_tan\code> python Assignment1Tester.py
[20:39:04] Starting test...
A database named "dds_assgn1" already exists
[20:39:04] Testing loadratings...
[20:39:17] loadratings: passed! (12.700 seconds)
```



The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays the database structure, with the 'ratings' table under the 'public' schema selected. The main pane shows the query 'SELECT \* FROM public.ratings' and its results. The results are displayed in a table with columns 'userid', 'movieid', and 'rating'. The table shows 13 rows of data, with the first row being (1, 122, 5) and the last row being (13, 420, 5). The status bar at the bottom indicates 'Total rows: 10000054' and 'Query complete 00:00:03.445'.

	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

- Hơn 10 triệu bản ghi: 10.000.054 dòng được nạp thành công.
- 12 giây: Hiệu suất chấp nhận được với xử lý theo khối.
- Tiết kiệm bộ nhớ: Sử dụng lệnh COPY với bộ đệm StringIO.

### 3. Range Partitioning Demo

```
Choose partitioning (range/roundrobin): range
[20:40:14] Testing RANGE partitioning...
[20:40:14] Creating 5 range partitions...
[20:40:41] rangepartition passed!
[20:40:41] Verifying range_part partition tables:
[20:40:41]   - range_part0: 479168 rows
[20:40:41]   - range_part1: 908584 rows
[20:40:42]   - range_part2: 2726854 rows
[20:40:42]   - range_part3: 3755614 rows
[20:40:42]   - range_part4: 2129834 rows
[20:40:43] Total rows: partitions=10000054, original=10000054
[20:40:43] Partition content passed!
[20:40:43] Testing range insert...
[20:40:43] rangeinsert: passed!
[20:40:43] Total partitioning + insert time: 29.211 seconds
```

The screenshot displays the pgAdmin 4 interface. On the left, the 'Object Explorer' shows the 'public' schema containing several tables, including 'range\_part0' through 'range\_part4' and 'ratings'. The 'range\_part0' table is selected. On the right, the 'Query' tab shows a SQL query: `SELECT * FROM public.range_part0`. Below the query, the 'Data Output' tab displays a table with 13 rows and 3 columns: 'userid' (integer), 'movieid' (integer), and 'rating' (double precision). The table shows data for 'range\_part0'.

	userid	movieid	rating
1	4	231	1
2	5	1	1
3	5	708	1
4	5	736	1
5	5	780	1
6	5	1391	1
7	6	3986	1
8	6	4270	1
9	7	1917	1
10	7	2478	1
11	7	5094	1
12	8	590	0.5
13	8	1035	0.5

At the bottom of the data output, it states: 'Total rows: 479168' and 'Query complete 00:00:00.302'.



- 29 giây: Thời gian phân mảnh.
- Phân phối không đều: Tự nhiên theo tần suất đánh giá.
- Tính toàn vẹn dữ liệu: Đảm bảo tính đầy đủ, không giao nhau và tái tạo hoàn hảo.

#### 4. Round Robin Demo

```
Choose partitioning (range/roundrobin): roundrobin
[20:41:15] Testing ROUND ROBIN partitioning...
[20:41:15] Creating 5 roundrobin partitions...
[20:41:59] roundrobinpartition passed!
[20:41:59] Verifying rrobin_part partition tables:
[20:41:59] - rrobin_part0: 2000011 rows
[20:41:59] - rrobin_part1: 2000011 rows
[20:42:00] - rrobin_part2: 2000011 rows
[20:42:00] - rrobin_part3: 2000011 rows
[20:42:00] - rrobin_part4: 2000010 rows
[20:42:00] Total rows: partitions=10000054, original=10000054
[20:42:00] Partition content passed!
[20:42:00] Testing roundrobin insert...
[20:42:01] roundrobininsert: passed!
[20:42:01] Total partitioning + insert time: 45.431 seconds
```

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays the database structure, with the 'public' schema expanded and 'rrobin\_part0' selected. The main pane shows a query window with the following SQL:

```
SELECT * FROM public.rrobin_part0
```

The query results are displayed in a table with the following columns: `userid` (integer), `movieid` (integer), and `rating` (double precision). The table shows 13 rows of data, with a total of 2000011 rows. The status bar at the bottom indicates 'Query complete 00:00:00.701'.

	userid integer	movieid integer	rating double precision
1	1	122	5
2	1	329	5
3	1	370	5
4	1	520	5
5	1	594	5
6	2	376	3
7	2	733	3
8	2	858	2
9	2	1391	3
10	3	590	3.5
11	3	1288	3
12	3	1674	4.5
13	3	4995	4.5

- 45 giây: Thời gian thực thi lâu hơn một chút do chi phí tính toán của hàm ROW\_NUMBER().
- Phân phối đồng đều: Tối ưu cho xử lý song song.

## 5. Cleanup

```
Press enter to delete all tables:
[20:42:07] Deleting all tables...
[20:42:13] Tables deleted.
```

## 5.2. Kết quả kiểm thử

Kết quả Range Partitioning:

- Completeness: Pass
- Disjointness: Pass
- Reconstruction: Pass
- Range Insert: Pass

Kết quả Round Robin Partitioning:

- Even Distribution: Pass
- Round Robin Insert: Pass
- Partition Balance: Pass

## 5.3. Đánh giá hiệu suất

Performance Benchmarks:

- Data Loading: 10M records trong ~12 seconds
- Range Partitioning: ~29 seconds cho 5 partitions
- Round Robin Partitioning: ~45 seconds cho 5 partitions

Comparison:

- Range partitioning nhanh hơn cho queries theo rating
- Round Robin đảm bảo phân bổ đều hơn
- Bulk insert nhanh hơn 100x so với row-by-row insert

## 5.4. Kết luận và hướng phát triển

Thành tựu đạt được:

- Thành công triển khai hệ thống database phân tán
- Tối ưu hóa hiệu suất với dữ liệu lớn
- Đảm bảo tính đúng đắn của partitioning
- Xây dựng test suite comprehensive

Hướng phát triển:

- Triển khai Hash Partitioning
- Thêm parallel processing
- Optimization cho SSD storage
- Monitoring và alerting system
- Web interface cho management

Bài học kinh nghiệm:

- Bulk operations quan trọng với big data
- Transaction management cần cẩn thận

- Testing automation tiết kiệm thời gian đáng kể
- Performance monitoring cần thiết từ đầu