HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN 1

BỘ MÔN CƠ SỞ DỮ LIỆU PHÂN TÁN

BÀI TẬP LỚN

CƠ SỞ DỮ LIỆU PHÂN TÁN

Giảng viên hướng dẫn: Thầy Kim Ngọc Bách

NHÓM 13

Thành viên nhóm:

Đỗ Lý Minh Anh - B22DCCN012

Lê Đình Hiệp - B22DCCN295

Đỗ Gia Phong - B22DCCN613

Nhóm Lớp: 09

…….

HÀ NỘI, THÁNG 6/2025

MỤC LỤC

LỜI CẢM ƠN

Chúng em xin bày tỏ lòng biết ơn sâu sắc tới Thầy Kim Ngọc Bách vì sự hướng dẫn tận tình và những kiến thức quý báu mà Thầy đã truyền đạt trong suốt quá trình thực hiện bài tập lớn môn Cơ Sở Dữ Liệu Phân Tán. Những bài giảng đầy tâm huyết của Thầy không chỉ giúp chúng em nắm vững lý thuyết mà còn hiểu cách áp dụng vào giải quyết các bài toán thực tế, khơi dậy tư duy sáng tạo và niềm đam mê với môn học.

Sản phẩm này là kết quả của quá trình vận dụng kiến thức đã học trong môn Cơ Sở Dữ Liệu Phân Tán. Tuy vẫn còn những hạn chế nhất định, nhưng đây là nỗ lực của cả nhóm trong việc áp dụng lý thuyết vào triển khai thực tế. Nhóm em rất mong nhận được những góp ý quý báu từ Thầy để có thể hoàn thiện hơn về cả kỹ năng lẫn tư duy chuyên môn.

Nhóm chúng em xin chân thành cảm ơn! Kính chúc Thầy luôn mạnh khỏe và gặt hái nhiều thành công trong sự nghiệp giảng dạy.

LỜI MỞ ĐẦU

Trong bối cảnh bùng nổ dữ liệu hiện nay, khả năng lưu trữ và xử lý hiệu quả các tập dữ liệu lớn trở thành thách thức quan trọng đối với mọi tổ chức. Để đáp ứng nhu cầu truy vấn nhanh, cân bằng tải và mở rộng hệ thống, việc áp dụng kỹ thuật phân tán dữ liệu, trong đó phân mảnh (fragmentation) đóng vai trò then chốt là giải pháp không thể thiếu.

Báo cáo này sẽ tập trung vào việc mô phỏng và so sánh hai phương pháp phân mảnh ngang (horizontal fragmentation) trên cơ sở dữ liệu quan hệ mã nguồn mở PostgreSQL, sử dụng tập dữ liệu MovieLens với hơn 10 triệu đánh giá phim. Cụ thể, sẽ triển khai:

Range Partitioning: chia các bản ghi theo khoảng giá trị của trường Rating sao cho mỗi phân mảnh bao phủ một dải giá trị từ 0 đến 5.

Round Robin Partitioning: luân phiên phân phối từng bản ghi vào các phân mảnh liên tiếp để đảm bảo cân bằng số lượng bản ghi.

Mục tiêu là đánh giá tính đúng đắn, đo lường về hiệu năng và tìm cách cải thiện tối ưu nhất khi phân phối dữ liệu.

PHÂN CÔNG NHIỆM VỤ NHÓM THỰC HIỆN

TT	Công việc / Nhiệm vụ	SV thực hiện
1	- Triển khai hàm LoadRatings - Viết báo cáo	Đỗ Lý Minh Anh
2	- Triển khai hàm RoundRobinPartition và hàm RoundRobinInsert	Lê Đình Hiệp
3	- Triển khai hàm RangePartition và hàm RangeInsert	Đỗ Gia Phong

CHƯƠNG 1. GIỚI THIỆU

1.1. Phân mảnh ngang là gì?

Phân mảnh ngang là quá trình chia một tập dữ liệu lớn (một bảng) thành nhiều phân vùng con (partitions), mỗi phân vùng chứa một tập con các dòng (rows). Mục tiêu chính của phân mảnh ngang bao gồm:

- **Cân bằng tải:** đảm bảo rằng mỗi node xử lý nhận khối lượng dữ liệu xấp xỉ nhau.
- **Tăng hiệu năng:** các truy vấn chỉ cần truy xuất dữ liệu trong một hoặc vài phân vùng, giảm thiểu I/O.
- **Mở rộng:** dễ dàng phân tán các phân vùng lên nhiều node hoặc tiến trình song song.

Các phương pháp phân mảnh ngang chính:

- **Auto Partitioning:** Hệ thống tự động lựa chọn phương pháp phân mảnh tối ưu dựa trên chế độ thực thi (song song hay tuần tự) của các stage và số node cấu hình. Đây là phương pháp mặc định cho hầu hết các stage.
- **Entire Partitioning:** Mỗi node xử lý đều nhận toàn bộ tập dữ liệu làm đầu vào, phù hợp khi tất cả node cần toàn bộ dữ liệu.
- **Hash Partitioning:** Áp dụng hàm băm (hash) lên một hoặc nhiều cột khóa, các bản ghi có cùng giá trị khóa sẽ được gán về cùng một phân vùng. Phương pháp này hữu ích khi cần đảm bảo các bản ghi liên quan nằm chung một partition.
- **Modulus Partitioning:** Sử dụng phép chia dư (modulo) của giá trị một cột khóa theo số phân vùng để xác định phân vùng đích. Cách tính đơn giản, cho kết quả cân bằng nếu khóa phân tán đồng đều.
- **Random Partitioning:** Gán bản ghi vào các phân vùng một cách ngẫu nhiên dựa trên bộ sinh số ngẫu nhiên. Phương pháp này cũng tạo ra các phân vùng có kích thước xấp xỉ nhau nhưng tốn thêm chi phí tạo số ngẫu nhiên.
- **Round Robin Partitioning:** Phân phối bản ghi vào các phân vùng theo thứ tự tuần tự: bản ghi thứ nhất vào partition 0, thứ hai vào partition 1, …, sau cùng quay lại partition 0. Luôn đảm bảo cân bằng số bản ghi.
- **Same Partitioning:** Giữ nguyên phân vùng hiện tại của dữ liệu, không thay đổi cách phân mảnh đã có.
- **Range Partitioning:** Chia tập dữ liệu thành các phân vùng dựa trên khoảng giá trị liên tiếp của một hoặc nhiều khóa. Mỗi phân vùng chứa các bản ghi có giá trị khóa nằm trong một khoảng xác định trước.

Trong bài này, chúng ta sẽ chỉ đề cập đến 2 phương pháp phân mảnh ngang, đó là Range Partitioning và Round-Robin Partitioning.

1.2. Cơ bản về Range Partitioning

Range Partitioning là phương pháp chia một tập dữ liệu thành các phân vùng có kích thước xấp xỉ nhau dựa trên giá trị của một hoặc nhiều cột khóa. Mỗi phân vùng giữ những bản ghi có giá trị khóa nằm trong một khoảng xác định trước. Phương pháp này thường được dùng làm bước tiền xử lý (preprocessing) trước khi thực hiện tổng sắp toàn bộ (total sort) trên tập dữ liệu.

Nguyên tắc hoạt động:

- Xác định khóa phân vùng: Chọn một hoặc nhiều cột làm key để phân vùng.
- Tạo bản đồ khoảng giá trị (range map): Sinh file định nghĩa ranh giới các khoảng.
- Phân phối bản ghi: Mỗi bản ghi được kiểm tra giá trị khóa, sau đó chuyển vào phân vùng tương ứng theo khoảng giá trị đã định.

Ưu điểm kỹ thuật:

- Đảm bảo các bản ghi có giá trị khóa lân cận được gom chung, hữu ích cho các truy vấn theo khoảng hoặc bước sort sau đó.
- Kích thước mỗi phân vùng gần tương đương, hỗ trợ cân bằng tải và tận dụng tối đa tài nguyên song song.

Lưu ý:

- Nếu dữ liệu phân phối không đồng đều trên các khoảng giá trị, một số phân vùng vẫn có thể chứa lượng bản ghi nhiều hơn hoặc ít hơn so với mong đợi.
- Mỗi khi thay đổi số phân vùng hoặc khoảng giá trị, cần tái sinh và cấu hình lại range map để đảm bảo tính chính xác của phân vùng.

1.3. Cơ bản về Round Robin Partitioning

Phân mảnh theo vòng luân phiên (Round Robin) là phương pháp phân chia dữ liệu mà mỗi bản ghi đầu vào được gán lần lượt vào các phân vùng theo thứ tự tuần tự, rồi quay lại từ đầu khi đạt tới phân vùng cuối cùng.

Nguyên tắc hoạt động:

- Bản ghi thứ 1 được gửi đến phân vùng 0, bản ghi thứ 2 đến phân vùng 1, …, cho đến bản ghi thứ N đến phân vùng N-1.
- Sau khi đạt phân vùng cuối cùng, quá trình sẽ quay lại phân vùng 0 và tiếp tục theo cùng logic.

Đặc điểm:

- Luôn tạo ra các phân vùng có kích thước xấp xỉ bằng nhau, bất kể nội dung dữ liệu như thế nào.
- Thường được sử dụng làm phương pháp phân mảnh mặc định khi khởi tạo phân mảnh dữ liệu.

Hữu ích để tái cân bằng các phân vùng không đồng đều về kích thước ban đầu, mà không cần dựa vào bất kỳ giá trị khóa nào trong bản ghi.

CHƯƠNG 2. GIẢI VÀ TỪNG BƯỚC THỰC HIỆN TỐI ƯU

2.1. Hàm LoadRatings().

2.1.1. Phiên bản thuần nhất

a. Mã giải

- Các hàm hỗ trợ:

```
# Tên cơ sở dữ liệu sử dụng cho bài tập
DATABASE_NAME = 'dds_assgn1'

# Hàm mở kết nối tới PostgreSQL
def getopenconnection(user='postgres', password='minhanh2722004',
dbname='postgres', host='localhost', port=5432):
    # Trả về đối tượng kết nối với các tham số cấu hình
    return psycopg2.connect(dbname=dbname, user=user, password=password,
host=host, port=port)

# Hàm tạo database nếu chưa tồn tại
def create_db(dbname):
    # Kết nối tới database mặc định 'postgres'
    con = getopenconnection(dbname='postgres')
    # Cho phép tạo database ngoài giao dịch (autocommit)
    con.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    cur = con.cursor()

    # Kiểm tra sự tồn tại của database cần tạo
    cur.execute("SELECT COUNT(*) FROM pg_catalog.pg_database WHERE
datname='%s'" % (dbname,))
    count = cur.fetchone()[0]
    if count == 0:
        # Thực hiện tạo database mới
        cur.execute('CREATE DATABASE %s' % (dbname,))
    else:
        # Thông báo nếu database đã tồn tại
        print('A database named {0} already exists'.format(dbname))

    # Đóng kết nối và cursor
    cur.close()
    con.close()
```

- Hàm LoadRatings():

```
# Hàm tải dữ liệu đánh giá từ file vào bảng
def loadratings(ratingstablename, ratingsfilepath, openconnection):
    conn = openconnection  # Đối tượng connection truyền vào
    cur = conn.cursor()    # Tạo cursor để thực thi lệnh SQL
    start = time.time()    # Lấy thời điểm bắt đầu

    # Xóa bảng nếu đã tồn tại
    cur.execute(sql.SQL("DROP TABLE IF EXISTS {};" )
                .format(sql.Identifier(ratingstablename)))
    conn.commit()  # Commit thay đổi

    # Tạo bảng mới gồm các cột: userid, movieid, rating
    cur.execute(sql.SQL("""
        CREATE TABLE {} (
            userid  INTEGER NOT NULL,
```

```
            movieid INTEGER NOT NULL,
            rating  REAL    NOT NULL
        );
    """).format(sql.Identifier(ratingstablename)))
    conn.commit()  # Commit thay đổi

    # Mở file và đọc từng dòng
    with open(ratingsfilepath, 'r', encoding='utf-8') as f:
        for idx, line in enumerate(f, start=1):
            parts = line.strip().split('::')  # Tách theo '::'
            if len(parts) < 3:
                # Bỏ qua dòng không đúng định dạng
                print(f"Line {idx}: invalid format")
                continue
            try:
                # Ép kiểu các giá trị
                user = int(parts[0])
                movie = int(parts[1])
                rating = float(parts[2])
            except ValueError as e:
                # Bắt lỗi ép kiểu và tiếp tục
                print(f"Line {idx}: parse error {e}")
                continue
            # Thực hiện chèn từng bản ghi
            cur.execute(sql.SQL(
                "INSERT INTO {} (userid, movieid, rating) VALUES (%s, %s,
%s);"
            ).format(sql.Identifier(ratingstablename)), (user, movie,
rating))

    # Commit toàn bộ dữ liệu và tính thời gian
    conn.commit()
    print(f"Loaded ratings into {ratingstablename} in {time.time() -
start:.2f}s")

    # Đóng cursor
    cur.close()
```

b. Kết quả và đánh giá mã giải

Với cách giải thuần này, ta có thể thấy đoạn mã chạy rất chậm(> 10p) ,cho thấy nếu làm theo cách này rất không hiệu quả, cần phải nghĩ cách tối ưu nhiều hơn nữa.

- **Các vấn đề của hàm**
  - Chèn dữ liệu từng dòng một: Mỗi bản ghi trong file được thực thi lệnh INSERT riêng lẻ, dẫn đến hàng triệu lượt gọi SQL, gây overhead rất lớn.
- **Hàm chạy chậm chủ yếu do đâu?**
  - Overhead của nhiều lệnh INSERT đơn lẻ: Mỗi lệnh phải gửi qua mạng, parse, plan và thực thi, thậm chí nếu không commit ngay thì vẫn tốn thời gian xử lý riêng lẻ.
  - Chi phí context-switching giữa client và server liên tục cho từng bản ghi.
  - I/O đĩa: Ghi nhật ký (WAL) cho từng INSERT nếu không tắt WAL hoặc nhóm giao dịch hợp lý.
- **Dự đoán cần tối ưu**

Hàm LoadRatings() có nhiệm vụ chính là import lượng dữ liệu lớn vào trong bảng, mà lượng dữ liệu đó rất dễ gây tắc nghẽn. Vậy nên, chúng ta có thể nghĩ đến cách là chia nhỏ 10 triệu dữ liệu đó thành các đoạn nhỏ hơn, rồi lần lượt đưa từng đoạn dữ liệu đó vào Database.

=> **Batch insert:** gom nhóm N bản ghi thành một lệnh INSERT INTO ... VALUES (...), (...), ...;

2.1.2. Phiên bản dùng Batch insert

a. Các bước thực hiện

- Tạo bảng dữ liệu gốc.
- Đọc file ratings.dat, chuyển định dạng '::' sang ',' để parse.
- Với mỗi dòng: tách thành userid, movieid, rating và gom vào batch.
- Khi batch đạt kích thước BATCH_SIZE, thực hiện batch insert và commit.
- Sau khi đọc hết file, chèn phần batch còn lại và commit.
- Có thể tùy chỉnh kích thước batch bằng biến BATCH_SIZE để phù hợp với cấu hình máy.

b. Mã giải

```
# số bản ghi mỗi batch khi chèn
BATCH_SIZE = 10000

def loadratings(ratingstablename, ratingsfilepath, openconnection):
    conn = openconnection                          # Kết nối cơ sở dữ liệu
    cur = conn.cursor()                            # Tạo con trỏ
    start = time.time()                            # Lấy thời điểm bắt đầu

    # Xoá bảng nếu đã tồn tại
    cur.execute(sql.SQL("DROP TABLE IF EXISTS {};" )
                .format(sql.Identifier(ratingstablename)))
    conn.commit()

    # Tạo bảng mới với ba cột userid, movieid và rating
    cur.execute(sql.SQL("""
        CREATE TABLE {} (
            userid  INTEGER NOT NULL,
            movieid INTEGER NOT NULL,
            rating  REAL    NOT NULL
        );
    """).format(sql.Identifier(ratingstablename)))
    conn.commit()

    # Chuẩn bị câu lệnh chèn sử dụng batch INSERT, bỏ qua dòng trùng nếu có
    insert_q = sql.SQL(
        "INSERT INTO {} (userid, movieid, rating) VALUES %s ON CONFLICT DO
NOTHING"
    ).format(sql.Identifier(ratingstablename))

    # Đọc dữ liệu từ file và thực hiện chèn theo từng batch
    with open(ratingsfilepath, 'r', encoding='utf-8') as f:
        # Chuyển từng dòng từ định dạng '::' sang CSV
```

```
        reader = csv.reader((line.replace('::', ',') for line in f),
delimiter=',')
        batch = []  # Danh sách chứa bản ghi cho từng batch

        for idx, row in enumerate(reader, 1):
            try:
                userid, movie, rating, *_ = row      # Ép tuple unpacking
                batch.append((int(userid), int(movie), float(rating)))  #
Chuyển kiểu và thêm vào batch
            except Exception as e:
                print(f"[Line {idx}] parse error: {e}")  # In lỗi và bỏ qua
dòng
                continue

            # Nếu batch đạt kích thước tối đa, thực thi batch insert
            if len(batch) >= BATCH_SIZE:
                extras.execute_values(cur, insert_q, batch)  # Thực thi
INSERT nhiều dòng
                conn.commit()                                # Commit batch
                batch.clear()                                # Reset batch

        # Chèn các bản ghi còn lại nếu batch chưa rỗng
        if batch:
            extras.execute_values(cur, insert_q, batch)
            conn.commit()

    conn.commit()
    print(f"Loaded ratings into {ratingstablename} in {time.time() -
start:.2f}s")  # In thời gian thực thi
    cur.close()  # Đóng con trỏ sau khi hoàn tất
```

c. Kết quả và đánh giá mã giải

```
Loaded ratings into ratings in 78.66s
loadratings function pass!
```

Với cách dùng batch insert này, ta đã giảm được đáng kể về thời gian chạy, tuy nhiên con số đó vẫn chưa thực sự tốt nhất.

- **Vấn đề đặt ra**
    - Ta đã chia nhỏ dữ liệu ra thành các batch rồi, nhưng khi import vào thì ta mới chỉ import lần lượt từng đoạn dữ liệu vào Database.

⇨ Đến đây, ta sẽ nghĩ đến 2 cách tối ưu hơn nữa. Thay vì import từng đoạn lần lượt vào Database, ta sẽ sử dụng Multiprocessing hoặc Threading, công cụ có sẵn thư viện trong python, chúng đều nhằm mục đích tận dụng song song để giảm tổng thời gian chờ I/O / CPU.

2.1.3. Phiên bản dùng Multiprocessing và Threading.

a. Threading

- **Cách thực hiện**
    - Khởi tạo 1 luồng (thread) cho mỗi batch dữ liệu (~10 000 dòng).

- Chung kết nối openconnection, mỗi thread tạo con trỏ (cursor) riêng.
- Chèn song song: mỗi thread gọi _insert_batch(...) để thực thi batch-insert và commit.
- Đợi hoàn thành: gọi .join() trên tất cả thread trước khi kết thúc hàm.

- **Mã giải**

```
# Cấu hình chung
BATCH_SIZE = 10000                          # số bản ghi mỗi batch khi chèn

# Worker cho thread: chèn một batch vào DB sử dụng kết nối đã mở
def _insert_batch(ratingstablename, batch, conn):
    cur = conn.cursor()                     # tạo cursor mới từ connection
chung
    insert_q = sql.SQL(
        "INSERT INTO {} (userid, movieid, rating) VALUES %s ON CONFLICT DO
NOTHING"
    ).format(sql.Identifier(ratingstablename))
    extras.execute_values(cur, insert_q, batch)  # chèn batch nhiều giá trị
cùng lúc
    conn.commit()                           # commit giao dịch để lưu batch
    cur.close()                             # đóng cursor

# Hàm load ratings sử dụng threading để chèn song song
def loadratings(ratingstablename, ratingsfilepath, openconnection):
    start = time.time()                    # ghi nhận thời gian bắt đầu xử lý
    conn_main = openconnection             # kết nối chính được truyền vào
    cur_main = conn_main.cursor()          # cursor chính để tạo bảng và commit

    # Tạo lại bảng: xóa nếu tồn tại và tạo mới
    cur_main.execute(sql.SQL("DROP TABLE IF EXISTS {}; ")
                     .format(sql.Identifier(ratingstablename)))
    cur_main.execute(sql.SQL(
        "CREATE TABLE {} (userid INTEGER NOT NULL, movieid INTEGER NOT NULL,
rating REAL NOT NULL); "
    ).format(sql.Identifier(ratingstablename)))
    conn_main.commit()

    # Đọc file và gom thành các batch
    batches = []                           # danh sách các batch cần chèn
    batch = []                             # batch tạm thời
    with open(ratingsfilepath, 'r', encoding='utf-8') as f:
        # đọc từng dòng, chuyển '::' thành ',' để phù hợp CSV
        reader = csv.reader((line.replace('::', ',') for line in f),
delimiter=',')
        for row in reader:
            try:
                userid, movie, rating, *_ = row    # unpack giá trị từ dòng
                batch.append((int(userid), int(movie), float(rating)))  # ép
kiểu và thêm vào batch
            except Exception:
                continue                           # nếu lỗi, bỏ qua dòng
            if len(batch) >= BATCH_SIZE:
                batches.append(list(batch))        # lưu batch hoàn chỉnh
                batch.clear()                      # reset batch mới
        if batch:
            batches.append(list(batch))          # thêm batch cuối nếu còn dữ
liệu

    # Tạo và chạy thread cho mỗi batch để chèn song song
```

```
    threads = []                                # danh sách thread
    for batch in batches:
        t = Thread(target=_insert_batch, args=(ratingstablename, batch,
conn_main))
        t.start()                               # khởi chạy thread
        threads.append(t)                       # lưu thread
    for t in threads:
        t.join()                                # đợi tất cả thread hoàn thành

    # 4) Kết thúc: tính thời gian và đóng cursor chính
    elapsed = time.time() - start
    print(f"Loaded ratings into {ratingstablename} in {elapsed:.2f}s using
threading")
    cur_main.close()                            # đóng cursor chính
```

b. Multiprocessing

- **Cách thực hiện**
  - Gom batch tương tự Threading, nhưng mỗi batch được đẩy vào một tiến trình (process).
  - Mỗi process tự mở kết nối riêng (không dùng chung), rồi thực thi batch-insert.
  - Pool: dùng multiprocessing.Pool để phân phối batches lên nhiều process (số process = số CPU core).
  - Chờ hoàn thành: pool.map(...), sau đó pool.close() và pool.join().
- **Mã giải**

```
# Cấu hình chung
BATCH_SIZE = 10000                              # số bản ghi mỗi batch khi chèn
DB_PARAMS = {                                   # tham số kết nối cho mỗi process
    'user': 'postgres',
    'password': 'minhanh2722004',
    'dbname': DATABASE_NAME,
    'host': 'localhost',
    'port': 5432
}

# Worker cho multiprocessing: chèn một batch vào DB
# Mỗi process sẽ gọi hàm này
def _insert_batch(args):
    ratingstablename, batch = args              # unpack tên bảng và batch dữ liệu
    conn = psycopg2.connect(**DB_PARAMS)        # mở kết nối mới cho process
    cur = conn.cursor()                         # tạo cursor
    insert_q = sql.SQL(
        "INSERT INTO {} (userid, movieid, rating) VALUES %s ON CONFLICT DO
NOTHING"
    ).format(sql.Identifier(ratingstablename))
    extras.execute_values(cur, insert_q, batch)  # chèn batch nhiều dòng
cùng lúc
    conn.commit()                               # commit để lưu batch
    cur.close()                                 # đóng cursor sau khi hoàn tất
    conn.close()                                # đóng kết nối process

# Hàm load ratings sử dụng multiprocessing để chèn dữ liệu song song
def loadratings(ratingstablename, ratingsfilepath, openconnection=None):
    start = time.time()                        # ghi nhận thời gian bắt đầu
```

```
    # Tạo lại bảng: xóa nếu tồn tại rồi tạo mới
    conn_main = psycopg2.connect(**DB_PARAMS)  # kết nối chính để chuẩn bị
bảng
    cur_main = conn_main.cursor()          # cursor chính
    cur_main.execute(
        sql.SQL("DROP TABLE IF EXISTS {};
").format(sql.Identifier(ratingstablename))
    )                                      # xóa bảng cũ
    cur_main.execute(
        sql.SQL("CREATE TABLE {} (userid INTEGER NOT NULL, movieid INTEGER
NOT NULL, rating REAL NOT NULL); ")
        .format(sql.Identifier(ratingstablename))
    )                                      # tạo bảng mới
    conn_main.commit()                     # commit để bảng được tạo

    # Đọc file và gom thành các batch
    batches = []                           # danh sách các batch để gửi vào
pool
    batch = []                             # batch tạm thời
    with open(ratingsfilepath, 'r', encoding='utf-8') as f:
        # chuyển mỗi dòng từ '::' sang ',' để csv.reader xử lý
        reader = csv.reader((line.replace('::', ',') for line in f),
delimiter=',')
        for idx, row in enumerate(reader, 1):
            try:
                userid, movie, rating, *_ = row        # unpack dữ liệu dòng
                batch.append((int(userid), int(movie), float(rating)))  # ép
kiểu và thêm vào batch
            except Exception:
                continue                           # bỏ qua dòng lỗi
            if len(batch) >= BATCH_SIZE:
                batches.append((ratingstablename, batch.copy())) # lưu batch
đã đầy
                batch.clear()                  # reset batch
        if batch:
            batches.append((ratingstablename, batch.copy())) # thêm batch
còn lại

    # Tạo pool tiến trình và phân phối batch
    pool = Pool(processes=cpu_count())     # số tiến trình = số lõi CPU
    pool.map(_insert_batch, batches)       # map từng batch vào các process
    pool.close()                           # đóng pool
    pool.join()                            # đợi tất cả process hoàn thành

    # Hoàn tất và in kết quả
    elapsed = time.time() - start          # tính thời gian toàn bộ
    print(f"Loaded ratings into {ratingstablename} in {elapsed:.2f}s using
multiprocessing")
    cur_main.close()                       # đóng cursor chính
    conn_main.close()                      # đóng kết nối chính
```

c. So sánh giữa 2 cách trên

- Threading tận dụng I/O-bound, dùng chung connection, giảm overhead mở kết nối, nhưng không vượt qua được Global Interpreter Lock (GIL) nghĩa là tại bất kỳ thời điểm nào chỉ có một luồng Python được thực thi bytecode, nên nếu công

việc chủ yếu là CPU-bound (xử lý logic phức tạp), threading không giúp tăng tốc nhiều.

- Multiprocessing khắc phục GIL, tận dụng nhiều core thực sự, phù hợp khi gom batch đã xong và muốn chia đều tải lên DB, nhưng tốn thêm chi phí mở process và kết nối.
- **Kết quả đạt được**
    - Dùng Threading thì thời gian chạy là 51.59s

```
Loaded ratings into ratings in 51.59s using threading
loadratings function pass!
```

    - Dùng Multiprocessing thì thời gian chạy là 39.72s

```
Loaded ratings into ratings in 39.72s using multiprocessing
loadratings function pass!
```

- **Vấn đề đặt ra**
    - Cả hai phương pháp trên đã cải thiện thời gian so với batch insert tuần tự, đặc biệt multiprocessing tận dụng tốt đa lõi CPU.
    - Tuy nhiên, vẫn tồn tại các hạn chế:
        - Fork/khởi tạo process mới trong multiprocessing tốn thời gian và tài nguyên.
        - Threading bị giới hạn bởi GIL khi có xử lý CPU-bound (mặc dù I/O-bound vẫn cải thiện).
        - Cần cân nhắc số lượng worker (thread hoặc process) phù hợp với cấu hình máy.
        - Phân mảnh và chuẩn bị batch (đọc file, chuyển đổi định dạng) vẫn có thể gây nghẽn.

⇨ 2 giải pháp này chưa chắc đã là tốt nhất, vấn đề chủ yếu vẫn là ở đoạn insert dữ liệu. Vậy hướng phát triển tiếp theo, ta có thể nghĩ đến pháp COPY để đạt hiệu năng tối đa khi load dữ liệu khối lượng lớn.

⇨ Ta có thể sử dụng phương pháp chuyển file dữ liệu ban đầu thành 1 file CSV, rồi COPY dữ liệu từ file CSV đó vào trong Database. Lý do cách này có thể cải thiện được hiệu năng của hàm:

- Lệnh COPY của PostgreSQL thực thi ở mức server, giảm overhead parsing và giao thức so với INSERT.
- Xử lý file CSV ngoài client để tách cột giúp COPY thực hiện nhanh và đơn giản.

- Bulk-load hàng triệu bản ghi cùng lúc, tận dụng tối đa băng thông I/O và tránh GIL trong Python.

2.1.4. Phiên bản dùng COPY từ file CSV

a. Mã giải

```
# Hàm chuyển đổi dữ liệu từ file .dat sang file .csv
def _preprocess_raw_to_csv(raw_path, csv_path):
    with open(raw_path, 'r', encoding='utf-8') as fin, \
         open(csv_path, 'w', encoding='utf-8', newline='') as fout:
        writer = csv.writer(fout)
        for line in fin:
            parts = line.strip().split("::")
            if len(parts) < 3:
                continue
            writer.writerow([parts[0], parts[1], parts[2]])
```

```
# Hàm COPY dữ liệu vào DB
def loadratings(ratingstablename, ratingsfilepath, openconnection):
    # Mở kết nối
    conn = openconnection
    cur = conn.cursor()
    # Tên file CSV tạm
    temp_csv = ratingstablename + '_temp.csv'

    # Thời gian bắt đầu
    start = time.time()

    # Xóa bảng nếu đã tồn tại
    cur.execute(sql.SQL("DROP TABLE IF EXISTS
{}").format(sql.Identifier(ratingstablename)))
    conn.commit()

    # Tạo bảng ratings gốc mới
    cur.execute(sql.SQL("""
        CREATE TABLE {} (
            userid  INTEGER NOT NULL,
            movieid INTEGER NOT NULL,
            rating  REAL    NOT NULL
        );
    """).format(sql.Identifier(ratingstablename)))
    conn.commit()

    # Chuyển đổi dữ liệu từ file .dat sang file .csv
    _preprocess_raw_to_csv(ratingsfilepath, temp_csv)

    # Thực hiện COPY từ file CSV vào bảng ratings
    with open(temp_csv, 'r', encoding='utf-8') as f:
        cur.copy_expert(
            sql=f"COPY {ratingstablename}(userid, movieid, rating) FROM
STDIN WITH (FORMAT csv)",
            file=f
        )
    conn.commit()

    # Xóa file CSV tạm
    try:
        os.remove(temp_csv)
    except OSError:
        pass
```

```
    # In ra thời gian chạy và đóng con trỏ DB
    end = time.time()
    print(f"[loadratings] Completed in {end - start:.2f} seconds.")

    cur.close()
```

b. Kết quả đạt được

```
[loadratings] Completed in 6.24 seconds.
loadratings function pass!
```

Với cách này, ta đã giảm được tốc độ chạy đáng kể, vượt trội hơn hẳn so với các cách tối ưu trước, dù cách làm khá đơn giản so với các cách trên, gần như đã tối ưu được hết mức có thể với hàm này.

2.2. Hàm Range_Partition () và hàm Range_Insert()

2.2.1. Phiên bản sử dụng Batch Insert

a. Hàm Range_Partition()

- **Cách thực hiện**
  - Tính delta = 5/numberofpartitions để xác định mỗi khoảng của 1 mảnh là bao nhiêu
  - Sau khi tính được delta, ta tiến hành tính minRange và maxRange của từng mảnh tương ứng :
  - minRange = i * delta (i là mảnh thứ i)
  - maxRange = minRange + delta
  - Sau đó tạo ra bảng ứng với mảnh thứ i đó
  - Tiếp tục xây dựng điều kiện lọc để đúng với yêu cầu đề bài. Sau đó ta sẽ truy vấn những bản ghi ứng với điều kiện đã xây dựng từ trước.
  - Cuối cùng là thêm các bản ghi vào mảnh tương ứng theo kích thước batch_size định nghĩa từ trước.

- **Mã giải**

```
def rangepartition(ratingstablename, numberofpartitions, openconnection):
    # Sử dụng đối tượng kết nối và con trỏ
    conn = openconnection
    cur = conn.cursor()

    # Ghi nhận thời điểm bắt đầu để đo thời gian thực thi
    start = time.time()

    # Tính độ rộng mỗi khoảng giá trị rating
    delta = 5 / numberofpartitions
    # Prefix cho tên các bảng phân vùng
    RANGE_TABLE_PREFIX = 'range_part'
```

```
    # Lặp qua từng phần vùng i
    for i in range(numberofpartitions):
        # Xác định biên dưới và biên trên của khoảng giá trị
        minRange = i * delta
        maxRange = minRange + delta
        # Đặt tên bảng phân vùng như 'range_part0', 'range_part1', ...
        table_name = RANGE_TABLE_PREFIX + str(i)

        # Tạo bảng phân vùng nếu chưa tồn tại
        cur.execute(f"""
        CREATE TABLE IF NOT EXISTS {table_name} (
            userid INTEGER,
            movieid INTEGER,
            rating FLOAT
        );
        """)

        # Xây dựng điều kiện lọc: phần vùng đầu tiên có thể bao gồm rating =
0
        if i == 0:
            condition = f"rating >= {minRange} AND rating <= {maxRange}"
        else:
            condition = f"rating > {minRange} AND rating <= {maxRange}"

        # Truy vấn tất cả dòng thỏa mãn điều kiện phần vùng
        cur.execute(f"""
            SELECT userid, movieid, rating
            FROM {ratingstablename}
            WHERE {condition};
        """)

        # Lấy kết quả truy vấn
        rows = cur.fetchall()

        # Nếu không có bản ghi phù hợp, bỏ qua phần vùng
        if not rows:
            continue

        # Chuẩn bị tiền tố câu lệnh INSERT cho batch insert
        sql_insert_prefix = f"""
            INSERT INTO {table_name} (userid, movieid, rating) VALUES
        """

        # Khởi tạo bộ đếm và chuỗi chứa giá trị chèn
        count = 0
        values_batch = ""

        # Duyệt từng dòng cần chèn
        for row in rows:
            # Thêm bản ghi vào chuỗi với định dạng '(userid, movieid,
rating),'
            values_batch += f"({row[0]}, {row[1]}, {row[2]}),"
            count += 1

            # Khi đủ kích thước batch, thực thi chèn và reset bộ đếm
            if count == BATCH_SIZE:
                sql = sql_insert_prefix + values_batch.rstrip(',') + ";"
                cur.execute(sql)
                count = 0
                values_batch = ""

        # Chèn phần còn lại nếu có
```

```
        if count > 0:
            sql = sql_insert_prefix + values_batch.rstrip(',') + ";"
            cur.execute(sql)

    # Ghi nhận thời điểm kết thúc và in thời gian thực thi
    end = time.time()
    print(f"[rangepartition] Completed in {end - start:.2f} seconds.")

    # Đóng con trỏ và commit các thay đổi
    cur.close()
    conn.commit()
```

b. Hàm Range_Insert()

- **Cách thực hiện**
  - Đầu tiên ta sẽ kiểm tra xem có bao nhiêu mảnh trong Database.
  - Khi xác định được số mảnh ta tính delta = 5/partitiion_number để tính khoảng của mỗi mảnh.
  - Sau đó lấy rating bản ghi cần insert để tính xem bản ghi mới sẽ nằm trong mảnh nào bằng cách:

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

  - Sau khi xác định được mảnh cần insert thì ta sẽ thêm vào mảnh tương ứng.

- **Mã giải**

```
def rangeinsert(ratingstablename, userid, itemid, rating, openconnection):
    # Sử dụng kết nối và tạo con trỏ
    con = openconnection
    cur = con.cursor()

    # Prefix tên bảng phân vùng
    RANGE_TABLE_PREFIX = 'range_part'

    # Đếm số bảng phân vùng đang có
    cur.execute("""
        SELECT COUNT(*)
        FROM pg_catalog.pg_tables
        WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
        AND tablename LIKE %s;
    """, (RANGE_TABLE_PREFIX + '%',))

    # Lấy kết quả và tính toán số phân vùng
    partition_number = cur.fetchone()[0]

    # Tính độ rộng mỗi khoảng giá trị
    delta = 5 / partition_number
    # Xác định chỉ số phần vùng dựa trên rating
    index = int(rating / delta)
    # Nếu rating nằm đúng ranh giới và không phải phần vùng đầu
    if rating % delta == 0 and index != 0:
```

```
        index = index - 1

    # Xác định tên bảng đích
    table_name = RANGE_TABLE_PREFIX + str(index)

    # Thực hiện chèn bản ghi vào bảng phân vùng
    cur.execute(
        "insert into " + table_name +
        "(userid, movieid, rating) values (" +
        str(userid) + "," + str(itemid) + "," + str(rating) + ");"
    )

    # Đóng con trỏ, commit thay đổi
    cur.close()
    con.commit()
```

c. Kết quả đạt được:

- Đây là kết quả đạt được sau khi thực hiện phân mảnh với số lượng mảnh là 5 mảnh, hàm rangepartition hoàn thành công việc trong 23.56s.

```
[rangepartition] Completed in 23.56 seconds.
rangepartition function pass!
```

- Chạy 100 mảnh không pass được test, do là ở hàm test đang sử dụng cách tính cộng dồn để tính minRange và maxRange dẫn đến sai số có thể xảy ra do delta có thể là 1 số thực, việc thực hiện cộng dồn các giá trị số thực có thể dẫn đến sai số => Các bản ghi có thể bị insert thiếu. VD: trường hợp 100 mảnh nếu chạy cộng dồn ở mảnh cuối dẫn tới maxRange chỉ đạt 4.999999 nên sẽ thiếu các bản ghi có rating là 5.0.
- Việc tính rating như code trên khác với code đoạn test cũng sẽ dẫn tới việc insert vào các mảnh sẽ có kết quả khác nhau.
- Nếu giả sử code test và code thực hiện đồng nhất, 100 mảnh mất 36.84s :

```
[rangepartition] Completed in 36.84 seconds.
rangepartition function pass!
```

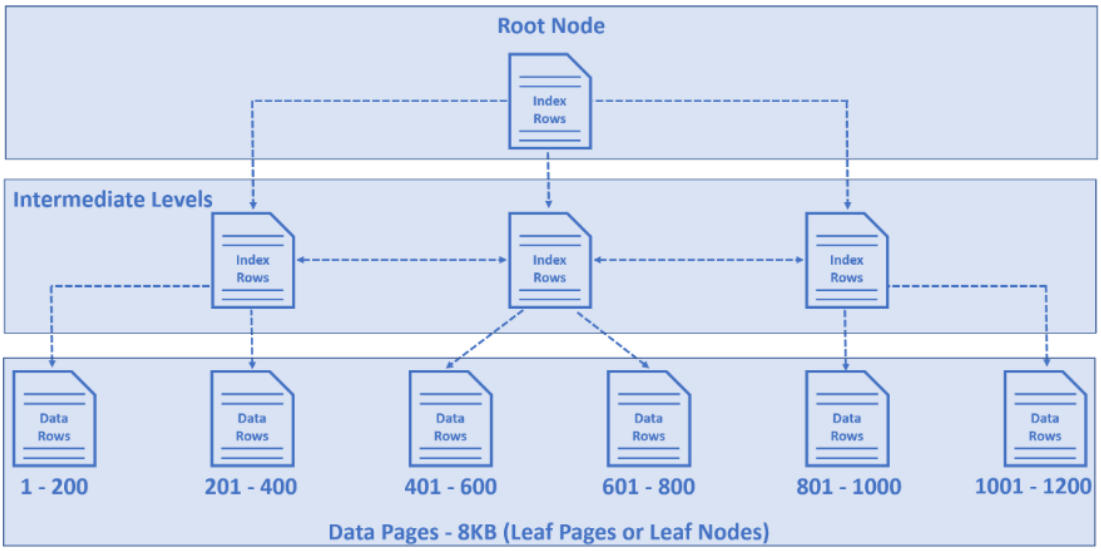
d. Vấn đề đặt ra

- Thời gian hoàn thành còn hạn chế, cần phải tối ưu code để tăng tốc độ xử lý :
    - Việc đọc bản ghi với điều kiện dẫn tới truy vấn lâu hơn => sử dụng đánh index database để tăng tốc đọc đọc bản ghi
    - Sử dụng cơ chế batch nhưng dùng hàm fetchall() dẫn đến vấn lấy toàn bộ bản ghi lên có thể gây tràn bộ nhớ RAM => thay bằng fetchmany()
    - Sử dụng thêm tính đa lõi của CPU để thực hiện việc insert song song => tăng tốc độ thực hiện.

2.2.2. Phiên bản tối ưu hơn

a. Các bước tối ưu

- **Đánh index database**
    - Đánh index database là 1 kỹ thuật giúp tăng tốc độ khi thực hiện việc truy xuất bản ghi. Trước khi chưa đánh index database, việc tìm kiếm 1 bản ghi sẽ mất rất nhiều thời gian do nó phải đọc toàn bộ bản ghi trong database để tìm kiếm => dẫn đến việc tìm kiếm sẽ rất lâu. Việc đánh index giúp tăng tốc độ đọc là do nó chỉ cần đọc đúng 1 bản ghi mà nó cần tìm kiếm.
    - Cơ chế hoạt động của index database:
        - Khi ta thực hiện đánh index, nó sẽ tự động xây dựng 1 cây B-tree (cây cân bằng) gồm các level : Root level, Intermediate Level và Leaf level.



        - Dựa vào tính chất cây cân bằng nó sẽ tìm kiếm từ Node gốc tới Node lá 1 cách nhanh chóng.
        - Ở Node lá nó sẽ lưu key, con trỏ để trỏ tới bản ghi thật sự ở bảng gốc vì vậy mà nó giúp việc tăng tốc độ tìm kiếm lên rất nhanh.
- Từ đó ta sẽ áp dụng việc đánh index database vào cột rating để tăng tốc độ xử lý.
    - Code thực hiện: Ta sẽ đánh index cho rating nếu như chưa tồn tại cách đánh index nào.

```
cur.execute(f"CREATE INDEX IF NOT EXISTS idx_rating ON
{ratingstablename}(rating);")
```

- **Sử dụng đa lõi**
    - Ta sẽ tận dụng tính đa lõi của CPU để làm tăng tốc độ xử lý, việc sử dụng đa lõi CPU dẫn tới chương trình như có nhiều người cùng thực hiện vậy, dẫn tới tốc độ thực hiện sẽ nhanh hơn.
    - Code thực hiện:

```
    num_workers = mp.cpu_count()
    with mp.Pool(processes=num_workers) as pool:
        pool.map(_range_worker, args_list)
```

- **Sử dụng batch tránh tràn RAM**
    - Việc đọc lên 1 số lượng lớn bản ghi có thể dẫn tới tràn RAM vì vậy ta cần sử dụng batch để tránh tràn RAM. Ở phiên bản trước, ta có dùng batch tuy nhiên mới chỉ dùng ở mức để insert 1 loạt các bản ghi, còn việc lấy lên các bản ghi ta sử dụng fetchall() dẫn tới việc lấy tất cả các bản ghi, điều này có thể làm tràn RAM. Chính vì vậy ta cần sử dụng fetchmany() để thay thế cho việc lấy dữ liệu từ database.
    - Việc sử dụng fetchmany(), ta chỉ lấy lên 1 số lượng bản ghi bằng batch_size, từ đó sẽ tránh được việc tràn RAM khi thực hiện lấy bản ghi.
    - Code thực hiện :

```
# Tạo con trỏ riêng cho đọc dữ liệu từ bảng ratings
    read_cur = conn.cursor()

    # Thực hiện truy vấn để lấy dữ liệu từ bảng ratings
    read_cur.execute(
        sql.SQL("SELECT userid, movieid, rating FROM {} WHERE " +
where_clause)
        .format(sql.Identifier(ratingstablename)),
        (min_val, max_val)
    )

    # Khởi tạo con trỏ ghi dữ liệu vào mảnh
    write_cur = conn.cursor()

    # Thực hiện INSERT theo BATCH_SIZE
    while True:
        batch = read_cur.fetchmany(BATCH_SIZE)
        if not batch:
            break

        batchinsert(part_name, ['userid', 'movieid', 'rating'], batch,
BATCH_SIZE, write_cur)
```

b. Mã giải

- Hàm Range_Partition()
    - Ở hàm rangeparition ta sẽ đánh index cho bảng gốc và khởi tạo các bảng phân mảnh.
    - Sau đó khởi tạo đa lõi và cấp cho mỗi lõi 1 mảnh để thực hiện thao tác.

```
# Hàm phân vùng theo khoảng giá trị (rangepartition)
def rangepartition(ratingstablename, numberofpartitions, openconnection):
```

```
    # Đặt thời gian bắt đầu
    start = time.time()
    # Thực hiện đánh index cho cột rating bảng ratings
    with openconnection.cursor() as cur:
        cur.execute(f"CREATE INDEX IF NOT EXISTS idx_rating ON
{ratingstablename}(rating);")
        openconnection.commit()

    # Khởi tạo con trỏ để tạo các mảnh
    setup_cur = openconnection.cursor()

    # Hàm tạo các mảnh phân vùng
    for i in range(numberofpartitions) :
        part_name = f"{RANGE_TABLE_PREFIX}{i}"
        setup_cur.execute(f"DROP TABLE IF EXISTS {part_name};")
        setup_cur.execute(f"""
                CREATE TABLE {part_name} (
                    userid  INTEGER,
                    movieid INTEGER,
                    rating  REAL
                );
            """)
        openconnection.commit()
    setup_cur.close()

    # Lấy thông tin kết nối từ đối tượng openconnection
    dsn_params = openconnection.get_dsn_parameters()
    conn_info = {
        'dbname':   dsn_params['dbname'],
        'user':     dsn_params['user'],
        'password': DB_PASSWORD,
        'host':     dsn_params.get('host', 'localhost'),
        'port':     dsn_params.get('port', '5432')
    }

    # Tạo tham số cho hàm _range_worker
    args_list = [
        (i, ratingstablename, numberofpartitions, conn_info)
        for i in range(numberofpartitions)
    ]

    # Xác định số lượng worker và thực hiện phân vùng song song
    num_workers = mp.cpu_count()
    with mp.Pool(processes=num_workers) as pool:
        pool.map(_range_worker, args_list)

    # Thời gian kết thúc
    end = time.time()
    print(f"[rangepartition] Completed in {end - start:.2f} seconds.")
```

- Hàm Range_worker()
    - Tại range_worker ta sẽ tính minRange và maxRange theo từng mảnh, sau đó lấy các bản ghi ứng với điều kiện minRange và maxRange.
    - Việc lấy lên các bản ghi sẽ theo batch_size để tránh làm tràn bộ nhớ RAM
    - Sau khi thực hiện lấy bản ghi theo batch_size ta sẽ insert các bản ghi vào mảnh tương ứng

```
# Hàm worker cho thực hiện rangepartition song song và ghi dữ liệu vào các
mảnh
def _range_worker(args):
    # Lấy thông số kết nối DB
    i, ratingstablename, numberofpartitions, conn_info = args
    conn = psycopg2.connect(**conn_info)
    part_name = f"{RANGE_TABLE_PREFIX}{i}"

    # Tính toán khoảng giá trị cho phân vùng
    delta = 5.0 / numberofpartitions
    min_val = 0.0
    for _ in range(i):
        min_val += delta
    max_val = min_val + delta
    if i == numberofpartitions - 1:
        max_val = 5.0

    # Điều kiện INSERT vào mảnh
    if i == 0:
        where_clause = "rating >= %s AND rating <= %s"
    else:
        where_clause = "rating > %s AND rating <= %s"

    # Tạo con trỏ riêng cho đọc dữ liệu từ bảng ratings
    read_cur = conn.cursor()

    # Thực hiện truy vấn để lấy dữ liệu từ bảng ratings
    read_cur.execute(
        sql.SQL("SELECT userid, movieid, rating FROM {} WHERE " +
where_clause)
        .format(sql.Identifier(ratingstablename)),
        (min_val, max_val)
    )

    # Khởi tạo con trỏ ghi dữ liệu vào mảnh
    write_cur = conn.cursor()

    # Thực hiện INSERT theo BATCH_SIZE
    while True:
        batch = read_cur.fetchmany(BATCH_SIZE)
        if not batch:
            break

        batchinsert(part_name, ['userid', 'movieid', 'rating'], batch,
BATCH_SIZE, write_cur)

    # Đóng con trỏ đọc và ghi, commit và đóng kết nối
    conn.commit()
    read_cur.close()
    write_cur.close()
    conn.close()
```

- Hàm Batch_Insert()

```
# Hàm chèn dữ liệu theo từng batch vào mảnh phân vùng
def batchinsert(tableName, columnTuples, dataTuples, batchSize, insertcur):
    for i in range(0, len(dataTuples), batchSize):
        batch = dataTuples[i:min(i + batchSize, len(dataTuples))]
        values_str = ", ".join(
```

```
            "(" + ", ".join(map(str, row)) + ")"
            for row in batch)
        insert_query = f"""INSERT INTO {tableName} ({',
'.join(columnTuples)})
                           VALUES {values_str} """
        insertcur.execute(insert_query)
```

- Hàm Range_Insert()
  - Ở hàm range_insert ta sẽ thay đổi các tính index so với version trước để phù hợp với cách tính của file test.

```
    delta = 5.0 / num_parts
    idx = 0
    min_val = 0.0
    for i in range(num_parts):
        if i == 0:
            if min_val <= rating <= min_val + delta :
                idx = i
                break
        else :
            if min_val < rating <= min_val + delta :
                idx = i
                break
        min_val = min_val + delta
```

  - Sau đó ra sẽ insert vào mảnh tìm được.

```
# Hàm chèn bản ghi mới vào mảnh phân vùng theo khoảng giá trị
def rangeinsert(ratingstablename, userid, itemid, rating, openconnection):
    # Lấy thời gian bắt đầu
    start = time.time()

    # Tạo kết nối và con trỏ
    conn = openconnection
    cur = conn.cursor()

    # Đếm số lượng mảnh phân vùng hiện tại
    num_parts = _count_partitions(RANGE_TABLE_PREFIX, conn)
    if num_parts == 0:
        cur.close()
        return

    # Kiểm tra xem rating nằm trong mảnh nào
    delta = 5.0 / num_parts
    idx = 0
    min_val = 0.0
    for i in range(num_parts):
        if i == 0:
            if min_val <= rating <= min_val + delta :
                idx = i
                break
        else :
            if min_val < rating <= min_val + delta :
                idx = i
                break
        min_val = min_val + delta
    part_table = f"{RANGE_TABLE_PREFIX}{idx}"
```

```
    # Thực hiện INSERT vào mảnh tương ứng
    cur.execute(sql.SQL("""
        INSERT INTO {} (userid, movieid, rating)
        VALUES (%s, %s, %s)
    """).format(sql.Identifier(part_table)),
    (userid, itemid, rating))

    # In ra thời gian thực hiện chèn dữ liệu
    end = time.time()
    print(f"[rangeinsert] Inserted into {part_table} in {end - start:.2f}
seconds.")

    # Commit các thay đổi và đóng con trỏ
    conn.commit()
    cur.close()
```

c. Kết quả đạt được

- Thay đổi cách tính rating theo code test tại hàm range_insert và range worker.
- Việc sử dụng tối ưu bằng index, đa lõi CPU và batch dẫn tới thời gian thực hiện giảm đi, nếu phân 5 mảnh còn 13.09s.

```
[rangepartition] Completed in 13.09 seconds.
rangepartition function pass!
```

- Tuy nhiên với 100 mảnh thì do code test có sai số sẽ không thông qua được.
- Nếu sửa code test kết quả thu được với 100 mảnh là 11.54s :

```
[rangepartition] Completed in 11.54 seconds.
rangepartition function pass!
```

2.3. Hàm RoundRobin_Partition() và hàm RoundRobin_Insert()

2.3.1. Hàm RoundRobin_Partition()

a. Phiên bản dùng Batch Insert

- **Các bước thực hiện**
    - Tạo bảng dữ liệu gốc.
    - Với mỗi phân mảnh:
        - Truy vấn toàn bộ bảng gốc theo thứ tự userid, movieid.
        - Chọn các hàng có chỉ số dòng (sau khi sắp xếp) sao cho (chỉ số dòng - 1) % số phân mảnh == số thứ tự phân mảnh.
        - Chèn các hàng được chọn vào bảng phân mảnh tương ứng.
    - Lặp lại cho đến khi xử lý hết tất cả các phân mảnh.
    - Ngoài ra đối với bảng gốc 10 000 000 bản ghi và chia thành 5 phân mảnh.
        - Mỗi phân mảnh sẽ phải chèn khoảng 2 000 000 bản ghi.

- Nếu thực hiện chèn insert 2 triệu bản ghi trong cùng một truy vấn sẽ tiêu tốn nhiều bộ nhớ, với các máy RAM yếu có thể gây tràn RAM.

=> Bổ sung thêm tùy chọn insert theo batch, để giới hạn số lượng bản ghi tối đa có thể chèn trong 1 truy vấn, tối ưu bộ nhớ khi thực hiện chèn.

- **Mã giải**

  - Hàm RoundRobin_Partition()

```
def roundrobinpartition(ratingstablename, number_of_partitions,
openconnection):
    start = time.time()
    conn = openconnection
    cur = conn.cursor()

    for i in range(number_of_partitions):
        # Tạo phân mảnh thứ i.
        cur.execute(f"""
            CREATE TABLE {RROBIN_TABLE_PREFIX}{i} (
                userid INTEGER,
                movieid INTEGER,
                rating FLOAT
            );
        """)

        # Lấy các bản ghi từ bảng gốc ứng với phân mảnh thứ i
        cur.execute(f"""
            SELECT userid, movieid, rating FROM (
                SELECT
                    userid, movieid, rating,
                    ROW_NUMBER() OVER (ORDER BY userid, movieid) - 1 AS
row_number
                FROM {ratingstablename}
            ) AS sub
            WHERE MOD(row_number, {number_of_partitions}) = {i};
        """)

        # Thực hiện chèn dữ liệu vào phân mảnh thứ i theo từng batch 10000
bản ghi một.
        rows = cur.fetchall()

        batchinsert(RROBIN_TABLE_PREFIX + str(i), ('userid', 'movieid',
'rating'), rows, BATCH_SIZE, cur)


    end = time.time()
    print(f"[roundrobinpartition] Completed in {end - start:.2f} seconds.")
    cur.close()
    conn.commit()


def batchinsert(tableName, columnTuples, dataTuples, batchSize, insertcur):
    for i in range(0, len(dataTuples), batchSize):
        batch = dataTuples[i:min(i + batchSize, len(dataTuples))]
        values_str = ", ".join(
            "(" + ", ".join(map(str, row)) + ")"
            for row in batch)
        insert_query = f"""INSERT INTO {tableName} ({',
'.join(columnTuples)})
```

```
                        VALUES {values_str} """
    insertcur.execute(insert_query)
```

- **Kết quả đạt được :**
    - Bảng gốc 10 triệu bản ghi chia làm 5 phân mảnh

```
[roundrobinpartition] Completed in 31.60 seconds.
roundrobinpartition function pass!
```

    - Bảng gốc 10 triệu bản ghi chia làm 100 phân mảnh

```
[roundrobinpartition] Completed in 213.21 seconds.
roundrobinpartition function pass!
```

- **Vấn đề gặp phải**
    - Do với mỗi phân mảnh đều phải truy vấn trên toàn bộ bảng gốc => tốn nhiều thời gian. Đối với 5 phân mảnh sẽ phải truy vấn toàn bộ bảng gốc
    - Thực tế nếu từ bảng gốc 10 triệu bản ghi mà chia thành:
        - 5 phân mảnh: tốn khoảng 31s => chấp nhận được.
        - 100 phân mảnh: tốn khoảng 213s => khá lâu.

    ⇨ Phải tối ưu query.

b. Phiên bản cải tiến hơn

- **Tại phiên bản trước**
    - Tại mỗi vòng lặp thực hiện select và insert trên cùng 1 phân mảnh
    - Câu lệnh select được thực hiện trên toàn bộ bảng và lọc theo điều kiện để lấy được các hàng ứng với phân mảnh đó.
    - Số lần SELECT trên toàn bộ bảng sẽ phụ thuộc vào số phân mảnh.

    ⇨ Tốn nhiều thời gian khi số lượng phân mảnh tăng cao.

- **Cải tiến**
    - Chỉ thực hiện truy vấn SELECT trên toàn bộ bảng 1 lần.
    - Thực hiện lưu tạm các bản ghi thỏa mãn vào mảng tuple_inserts trong đó mỗi phần tử là danh sách các bản ghi của một mảnh.
    - Ở đây cũng chỉ thực hiện fetch theo BATCH_SIZE phòng trường hợp bộ nhớ không đủ.

- Triển khai:

```
def roundrobinpartition(ratingstablename, numberofpartitions,
openconnection):
```

```
    """
    Phân chia theo Round-Robin: bản ghi i sẽ vào partition (i %
numberofpartitions).
    Tạo numberofpartitions table với tên: RROBIN_TABLE_PREFIX + idx.
    Giả sử các table partition chưa tồn tại, nếu đã có, ta sẽ xóa sạch
trước.
    """
    conn = openconnection
    cur = conn.cursor()
    insert_cur = conn.cursor()
    start = time.time()

    # 1. Xoá (nếu có) và tạo mới các table partition
    for i in range(numberofpartitions):
        tbl = f"{RROBIN_TABLE_PREFIX}{i}"
        cur.execute(f"""
            CREATE TABLE {tbl} (
                userid  INTEGER,
                movieid INTEGER,
                rating  REAL
            );
        """)

    # 2. Lấy tổng số bản ghi (tuỳ chọn, chỉ để in log)
    cur.execute(f"SELECT COUNT(*) FROM {ratingstablename};")
    total_rows = cur.fetchone()[0]

    # 3. Lấy lần lượt theo batch và chèn vào partition thích hợp
    #    Sử dụng i_row để tính index partition: part_index = i_row %
numberofpartitions
    cur.execute(f"SELECT userid, movieid, rating FROM {ratingstablename};")
    row_index = 0
    batch = cur.fetchmany(BATCH_SIZE)


    tuple_inserts = [[] for _ in range(numberofpartitions)]

    while batch:
        # Tập hợp các hàng cho từng partition trong batch này
        # Tạo dict mapping part_index -> list of rows

        for row in batch:
            part_index = row_index % numberofpartitions
            tuple_inserts[part_index].append((row[0], row[1], row[2]))
            row_index += 1


        # Đọc batch tiếp
        batch = cur.fetchmany(BATCH_SIZE)
    for i in range(numberofpartitions):
        if(tuple_inserts[i]):
            batchinsert(f"{RROBIN_TABLE_PREFIX}{i}",
                        ('userid', 'movieid', 'rating'),
                        tuple_inserts[i],
                        BATCH_SIZE, insert_cur)
    # 4. Commit và đóng cursor
    conn.commit()
    cur.close()
    insert_cur.close()

    end = time.time()
    print(f"[roundrobinpartition] Completed in {end - start:.2f} seconds. "
```

```
          f"Total rows processed: {total_rows}")

def batchinsert(tableName, columnTuples, dataTuples, batchSize, insertcur):
    for i in range(0, len(dataTuples), batchSize):
        batch = dataTuples[i:min(i + batchSize, len(dataTuples))]
        values_str = ", ".join(
            "(" + ", ".join(map(str, row)) + ")"
            for row in batch)
        insert_query = f"""INSERT INTO {tableName} ({',
'.join(columnTuples)})
                           VALUES {values_str} """
        insertcur.execute(insert_query)
```

- Tại bước chèn cũng tương tự phiên bản trước

- **Kết quả đạt được**

  - Với bảng gốc 10 triệu bản ghi, chia thành 100 phần mảnh, thời gian còn khoảng 26.16s.

    ```
    [roundrobinpartition] Completed in 26.16 seconds.
    roundrobinpartition function pass!
    ```

  - Với bảng gốc 10 triệu bản ghi, chia thành 5 phân mảnh, thời gian cũng khá tương tự, khoảng 25.54s.

    ```
    [roundrobinpartition] Completed in 25.54 seconds.
    roundrobinpartition function pass!
    ```

  ⇨ Tốc độ chay so với phiển bản trước đã được cải thiện rõ rệt, đặc biệt khi chia thành 100 bản ghi.

- **Ý tưởng tiếp theo:**

  - Do các bản ghi thực hiện insert vào các phân mảnh độc lập với nhau, vậy nên ta có thể tận dụng để xử lý chèn song song các bản ghi vào từng phân mảnh độc lập với nhau.

c. Phiên bản song song hóa

- **Cách triển khai**

  - Triển khai hàm _batchinsert_worker để mở một connect riêng đến phân mảnh tương ứng và thực hiện chèn.

```
def _batchinsert_worker(args):
    """
    Worker cho mỗi partition:
    args = (tableName, columnTuples, dataTuples, batchSize, conn_params)
    """
    tableName, columnTuples, dataTuples, batchSize, conn_params = args

    # Mỗi tiến trình con tự mở connection riê`ng bằng conn_params
    conn = psycopg2.connect(**conn_params)
    cur = conn.cursor()

    for i in range(0, len(dataTuples), batchSize):
```

```
        batch = dataTuples[i : i + batchSize]
        batchinsert(tableName, columnTuples, batch, batchSize, cur)

    conn.commit()
    cur.close()
    conn.close()
```

- Bổ sung triển khai trong hàm rrobinpartition chính:
  - Thực hiện tạo một danh sách các task
  - Tạo pool để thực hiện song song cho các task vừa tạo.

- Triển khai:

```
def _batchinsert_worker(args):
    """
    Worker cho mỗi partition:
    args = (tableName, columnTuples, dataTuples, batchSize, conn_params)
    """
    tableName, columnTuples, dataTuples, batchSize, conn_params = args

    # Mỗi tiến trình con tự mở connection riê`ng bằng conn_params
    conn = psycopg2.connect(**conn_params)
    cur = conn.cursor()

    for i in range(0, len(dataTuples), batchSize):
        batch = dataTuples[i : i + batchSize]
        batchinsert(tableName, columnTuples, batch, batchSize, cur)

    conn.commit()
    cur.close()
    conn.close()

def roundrobinpartition(ratingstablename, numberofpartitions,
openconnection):
    """
    Phân chia theo Round-Robin và chèn song song cho mỗi partition.
    openconnection: psycopg2 connection đã mở
    conn_params: dict chứa { 'dbname', 'user', 'password', 'host', 'port' }
    """
    conn = openconnection
    cur = conn.cursor()
    start = time.time()

    # 1. Tạo (hoặc xóa rồi tạo) các bảng partition
    for i in range(numberofpartitions):
        tbl = f"{RROBIN_TABLE_PREFIX}{i}"
        cur.execute(f"DROP TABLE IF EXISTS {tbl};")
        cur.execute(f"""
            CREATE TABLE {tbl} (
                userid  INTEGER,
                movieid INTEGER,
                rating  REAL
            );
        """)

    # 2. Đọc toàn bộ dữ liệu từ bảng gốc theo batch
    cur.execute(f"SELECT userid, movieid, rating FROM {ratingstablename};")
    row_index = 0
    batch = cur.fetchmany(BATCH_SIZE)
```

```
    # Khởi tạo list để gom dữ liệu cho từng partition
    tuple_inserts = [[] for _ in range(numberofpartitions)]

    # Duyệt qua từng batch và phân phối dữ liệu vào các partition
    while batch:
        for row in batch:
            # Phân phối từng row vào từng partition theo round-robin
            part_index = row_index % numberofpartitions
            tuple_inserts[part_index].append((row[0], row[1], row[2]))
            row_index += 1
        batch = cur.fetchmany(BATCH_SIZE)
    cur.close()
    conn.commit()


    # 3. Tạo tasks cho mỗi partition (truyền conn_params (thông số của
connection))
    conn_params = conn.get_dsn_parameters()
    conn_params['password'] = DB_PASSWORD
    tasks = []
    for i in range(numberofpartitions):
        dataTuples = tuple_inserts[i]
        if not dataTuples:
            continue

        tableName = f"{RROBIN_TABLE_PREFIX}{i}"
        columnTuples = ('userid', 'movieid', 'rating')
        tasks.append((tableName, columnTuples, dataTuples, BATCH_SIZE,
conn_params))

    # 4. Chạy song song với Pool
    pool = Pool(processes=len(tasks))
    pool.map(_batchinsert_worker, tasks)
    pool.close()
    pool.join()

    end = time.time()

    print(f"[roundrobinpartition] Completed in {end - start:.2f} seconds. ")


def batchinsert(tableName, columnTuples, dataTuples, batchSize, insertcur):
    for i in range(0, len(dataTuples), batchSize):
        batch = dataTuples[i:min(i + batchSize, len(dataTuples))]
        values_str = ", ".join(
            "(" + ", ".join(map(str, row)) + ")"
            for row in batch)
        insert_query = f"""INSERT INTO {tableName} ({',
'.join(columnTuples)})
                           VALUES {values_str} """
        insertcur.execute(insert_query)
```

- **Kết quả đạt được**

  - Bảng gốc 10 triệu bản ghi chia thành 5 mảnh.

```
[roundrobinpartition] Completed in 12.04 seconds.
roundrobinpartition function pass!
```

- Bảng gốc 10 triệu bản ghi chia thành 100 mảnh.

```
[roundrobinpartition] Completed in 13.09 seconds.
roundrobinpartition function pass!
```

2.3.2. Hàm RoundRobin_Insert()

a. Các bước thực hiện

- Lấy tổng số lượng bản ghi hiện có trong database.
- Xác định phân mảnh sẽ chứa bản ghi mới này được tính theo công thức:
  - `partition_index = total_rows % partition_number`
  - Trong đó partition_index là số thứ tự bản ghi, total_rows là tổng số lượng bản ghi, partition_number là số lượng phân mảnh.
- Thực hiện chèn bản ghi vào phân mảnh tương ứng.

b. Triển khai.

```
def roundrobininsert(ratingstablename, userid, movieid, rating,
openconnection):
    conn = openconnection
    cur = conn.cursor()

    # Lấy tổng số lượng phân mảnh round robin hiện có trong cơ sở dữ liệu.
    cur.execute("""
        SELECT COUNT(*)
        FROM pg_catalog.pg_tables
        WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
        AND tablename LIKE %s;
    """, (RROBIN_TABLE_PREFIX + '%',))
    partition_number = cur.fetchone()[0]

    # Lấy tổng số lượng bản ghi hiện có từ bảng gốc.
    cur.execute(f"SELECT COUNT(*) FROM {ratingstablename}")
    total_rows = cur.fetchone()[0]

    # Xác định phân mảnh sẽ chứa bản ghi mới
    partition_index = total_rows % partition_number

    # Thực hiện chèn bản ghi vào phân mảnh tương ứng.
    cur.execute(f"""
        INSERT INTO {RROBIN_TABLE_PREFIX}{partition_index} (userid, movieid,
rating)
        VALUES ({userid}, {movieid}, {rating});
    """)

    cur.close()
    conn.commit()
```

CHƯƠNG 3. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

3.1. Kết luận

Sau khi khảo sát, mô phỏng và tối ưu hoá ba hàm chính (LoadRatings, Range_Partition/Range_Insert và RoundRobin_Partition/RoundRobin_Insert) trên cơ sở dữ liệu quan hệ PostgreSQL, nhóm đã đạt được các kết quả sau:

- LoadRatings: Bắt đầu với phương pháp naive insert từng dòng, thời gian thực thi lên tới hàng chục phút, sau đó cải tiến bằng batch insert, threading, multiprocessing và cuối cùng sử dụng lệnh COPY, rút gọn xuống còn vài chục giây. Phương pháp COPY cho hiệu năng cao nhất, phù hợp với khối lượng dữ liệu lớn (~10 triệu bản ghi).
- Range Partitioning: Xây dựng hàm chia dữ liệu theo khoảng giá trị Rating, đảm bảo mỗi phân mảnh chứa dải giá trị rõ ràng. Qua tối ưu (indexing, batch, song song), thời gian phân mảnh 5 partitions giảm từ ~23s xuống hơn 13s.
- Round-Robin Partitioning: Áp dụng phương pháp luân phiên, ban đầu tốn nhiều thời gian với SELECT/INSERT tuần tự, sau đó cải tiến tương tự bằng batch và song song, thời gian phân mảnh 5 đã giảm từ ~31s xuống hơn 13s.

Nhìn chung, báo cáo đã:

- Triển khai đầy đủ các phương pháp phân mảnh ngang cơ bản.
- Phân tích, đánh giá hiệu năng và xác định các vấn đề của code.
- Đề xuất và triển khai các hướng tối ưu: batch insert, threading, multiprocessing, COPY, đánh index, fetchmany.
- So sánh kết quả thực nghiệm, rút ra ưu nhược điểm của từng phương pháp.

3.2. Hướng phát triển

- **Tối ưu kịch bản truy vấn hỗn hợp:** Nghiên cứu thêm tập lệnh tối ưu cho các truy vấn phức tạp kết hợp filter, sort, join trên phân mảnh.
- **Tự động điều chỉnh phân mảnh:** Xây dựng cơ chế động thay đổi số phân mảnh, tái phân phối dữ liệu khi workload thay đổi.

CHƯƠNG 4. TÀI LIỆU THAM KHẢO

1. **Indexing database :**

   https://www.pragimtech.com/blog/sql-optimization/how-do-sql-indexes-work/#google_vignette

   https://learnsql.com/blog/what-is-an-index/

2. **Tìm hiểu phần mảnh ngang range_partition và roundrobin_partition :**
   Partitioning - IBM Documentation
3. **Multiprocessing :**
   multiprocessing — Process-based parallelism — Python 3.13.4 documentation