

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH



HỆ QUẢN TRỊ CƠ SỞ DỮ LIỆU (CO2017)
Final Assignment

PostgreSQL - Neo4j

Lớp L01 - Nhóm 8:

GVHD: Võ Thị Ngọc Châu
SV thực hiện: Nguyễn Kế Văn – 1915887
 Nguyễn Hồng Thiện – 1915290
 Trần Lê Viết Khánh – 1913758
 Đương Huỳnh Anh Đức – 2010226

Tp. Hồ Chí Minh, Tháng 05/2023



Mục lục

1 PostgreSQL	5
1.1 Introduction	5
1.2 Architectural Fundamentals	6
1.3 Indexing	6
1.3.1 Index Types (các loại chỉ mục)	7
1.3.1.a B-Tree	8
1.3.1.b Hash	8
1.3.1.c GiST	8
1.3.1.d SP-GiST	8
1.3.1.e GIN	8
1.3.1.f BRIN	9
1.3.2 Multicolumn Indexes (chỉ mục đa cột)	9
1.4 Query processing	9
1.5 Optimization	11
1.5.1 Genetic Algorithms	12
1.5.1.a Genetic Query Optimization (GEQO) in PostgreSQL	14
1.5.1.b Generating Possible Plans with GEQO (Sinh kế hoạch khả thực thi với GEQO)	14
1.5.2 EXPLAIN in PostgreSQL	15
2 Neo4j	21
2.1 Giới thiệu	21
2.1.1 Ưu điểm	21
2.1.2 Nhuược điểm	21
2.2 Indexing	22
2.2.1 Range index	22
2.2.2 Lookup index	22
2.2.3 Text index	23
2.2.4 Point index	23
2.2.5 Full-text index	25
2.3 Query processing:	27
2.4 Query optimization:	28
2.4.1 General Architecture	28
2.4.2 Operator Ordering	28
2.4.3 Profile a query	31
2.4.4 Example	31
2.4.4.a Example 1	31
2.4.4.b Example 2	34
2.4.4.c Example 3	36
2.4.4.d Example 4	37
2.4.4.e Example 5	38
3 Comparison between PostgreSQL and Neo4j	41
4 Application	46
4.1 Nguồn dữ liệu	46
4.2 Mô tả dữ liệu	46
4.3 Xử lý dữ liệu	50
4.4 Demo ứng dụng	52
4.5 Bonus	53



Danh sách hình vẽ

1	Kiến trúc dựa trên mô hình Client-Server	6
2	Quá trình xử lý truy vấn	10
3	Ví dụ về bước parser	10
4	Cây truy vấn được tạo ra sau khi hoàn thành 2 bước đầu	11
5	Plan tree là công cụ để giao tiếp với Executor	11
6	Structure of a Genetic Algorithm	13
7	join tree	14
8	Neo4j	21
9	Ví dụ point index	24
10	Analysis	25
11	Inverted index	26
12	Cách tính hàm TF.IDF	26
13	Neo4j Processing	27
14	Ví dụ về biểu diễn đồ thị cho câu truy vấn	28
15	Thuật toán tham lam cho việc xếp thứ tự toán tử trong Cypher	29
16	Ví dụ minh họa cho thuật toán Greedy	29
17	Áp dụng thuật toán Greedy cho ví dụ minh họa	30
18	Kết quả cho câu truy vấn tìm "Terry Notary"	31
19	Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary"	32
20	Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary" có chỉ định nhãn là People	33
21	Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary" có index và chỉ định nhãn là People	34
22	Kết quả cho câu truy vấn người có tên bắt đầu là "Terry"	35
23	Kế hoạch thực thi cho câu truy vấn người có tên bắt đầu là "Terry"	36
24	Kế hoạch thực thi cho câu truy vấn số lượng tên người riêng biệt trong CSDL	37
25	Kế hoạch thực thi cho câu truy vấn người có tên bắt đầu là "Terry" và xếp thứ tự theo tên	38
26	Kế hoạch thực thi cho câu truy vấn diễn viên đầu tiên xuất hiện theo thứ tự bảng chữ cái	39
27	Kết quả thực thi movie thể loại comedy ở PostgreSQL	41
28	Kết quả thực thi movie thể loại comedy ở Neo4j	42
29	Kết quả thực thi movie thể loại comedy và năm xuất bản 1993 ở PostgreSQL	42
30	Kết quả thực thi movie thể loại comedy và năm xuất bản 1993 ở Neo4j	42
31	Kết quả thực thi chọn diễn viên trong các phim mà họ đóng có phim có trung bình đánh giá lớn hơn 4 ở PostgreSQL	43
32	Quá trình thực thi câu lệnh	43
33	Kết quả thực thi chọn diễn viên trong các phim mà họ đóng có phim có trung bình đánh giá lớn hơn 4 ở Neo4j	44
34	Kết quả thực thi những người vừa làm diễn viên vừa làm đạo diễn ở PostgreSQL	44
35	Quá trình thực thi câu lệnh	44
36	Kết quả thực thi những người vừa làm diễn viên vừa làm đạo diễn ở Neo4j	45
37	File links.csv	46
38	File movies.csv	47
39	File ratings.csv	47
40	File tags.csv	48
41	File tmdb_5000_credits.csv	49
42	File tags.csv	50
43	File directors.csv	51
44	File actors.csv	52
45	Kết quả trả về thể loại, diễn viên và đạo diễn của một bộ phim tên là "Harry Potter and the Sorcerer's Stone"	53
46	Kết quả trả về những người đã xếp hạng hoặc gắn nhãn 1 bộ phim "Harry Potter and the Sorcerer's Stone"	54
47	Kết quả trả về đề xuất phim cho người dùng có id là "220"	55
48	Kết quả trả về đề xuất phim cho người dùng có id là "220" (improved version)	56
49	Danh sách bộ phim đề xuất cho người dùng có id là 220	57
50	Danh sách bộ phim đề xuất cho người dùng có id là 300 dựa trên nhãn Tags	58



51	Danh sách các diễn viên mà người dùng có id 312 thích theo thứ tự giảm dần	59
52	Danh sách các đạo diễn mà người dùng có id 312 thích theo thứ tự giảm dần	59
53	Danh sách các đạo diễn mà người dùng có id 312 thích theo thứ tự giảm dần	60
54	Danh sách bộ phim đề xuất cho người dùng id 312 theo thứ tự trọng số yêu thích giảm dần	61
55	Jaccard	62
56	Danh sách bộ phim đề xuất cho người dùng id 220 theo hàm độ đo tương tự jaccard . .	63
57	Pearsons formula	63
58	Danh sách người dùng tương đồng với người dùng có id là 220	64
59	Minh họa độ tương đồng giữa người dùng id 220 và người dùng id 494	65
60	Đề xuất phim cho người dùng id là 220 dựa trên 10 người dùng tương đồng nhất và xếp hạng theo cách tính toán trọng số thích score giảm dần	66
61	Minh họa cho kết quả trả về câu truy vấn trên	66
62	Kết quả danh sách xếp hạng những bộ phim mà người có id 220 không thích nhất	67



Bảng phân công công việc

Member	Work
Nguyễn Kế Văn	Neo4j - Point index, fulltext index, query optimization cho Neo4j, xử lý dữ liệu trong phần ứng dụng, Recommendation trong phần bonus, tính năng đăng nhập và đăng ký cho trang web.
Dương Huỳnh Anh Đức	- PostgreSQL: Indexing, Optimization. - Comparision: Thiết kế database, các câu lệnh query so sánh PostgreSQL. - Web: Movie Details, Listing Ratings, Listing People, Person Profile.
Nguyễn Hồng Thiện	PostgreSQL: Query processing, thiết kế ứng dụng: Find Movie by genres, Detail genres, Listing movie by genres and pagination.
Trần Lê Viết Khánh	Neo4j : Tổng quan và giới thiệu, query processing, indexing (range, lookup, text) Comparison: So sánh giữa 2 database, các câu lệnh query so sánh Application: Xác thực, Sửa lỗi các API về movie Trình bày và quay video



1 PostgreSQL

1.1 Introduction

PostgreSQL là một hệ thống quản trị cơ sở dữ liệu quan hệ - đối tượng (object-relational database management system), sử dụng Structured Query Language (SQL) - ngôn ngữ truy vấn có cấu trúc, dựa trên nền tảng POSTGRES, phiên bản 4.2. PostgreSQL được phát triển tại phòng Khoa Học Máy Tính Barkeley - Đại học California.

PostgreSQL hỗ trợ phần lớn các tiêu chuẩn SQL và cung cấp nhiều tính năng hiện đại:

- complex queries (các lệnh truy vấn phức tạp)
- foreign keys (khóa ngoại)
- Triggers
- Updatable views
- Transacional integrity (tính toàn vẹn của giao tác)
- Multiversion concurrency control (kiểm soát tính đồng thời)

Bên cạnh đó, PostgreSQL còn có thể được mở rộng bởi người dùng bằng nhiều cách, lấy ví dụ như thêm mới:

- Data types (kiểu dữ liệu)
- Functions (hàm)
- Operators (toán tử)
- Aggregate functions (hàm tổng hợp)
- index methods (phương pháp chỉ mục)
- procedural languages (ngôn ngữ thủ tục)

PostgreSQL là một trong những hệ cơ sở dữ liệu quan hệ mã nguồn mở chuyên nghiệp nhất, đã đạt được giải thưởng "Database System Of The Year" một vài năm. Một hệ thống có tính tin cậy cao, ổn định, có thể mở rộng và bảo mật cao.

Những lợi ích mà PostgreSQL mang lại:

- Open Source (mã nguồn mở): Cho phép tự do sử dụng, sửa đổi, triển khai theo nhu cầu cần thiết trong kinh doanh.
- Reduce costs (giảm chi phí): PostgreSQL hoàn toàn miễn phí để sử dụng! Không cần quan ngại về chi phí cấp phép, vấn đề hợp đồng.
- Reliability (tính tin cậy): Nhiều công ty và cá nhân đóng góp vào dự án và đã thúc đẩy sự đổi mới trong hơn 25 năm nay. Một cộng đồng vững mạnh đảm bảo rằng các lỗi sẽ được sửa nhanh chóng. Khi phát hiện lỗi trong PostgreSQL có thể gửi mail thông báo lỗi đến <pgsql-bugs@lists.postgresql.org>.
- Security (tính bảo mật): Có nhiều tính năng để tăng cường bảo mật, nhờ khả năng mở rộng dễ dàng. Nếu sử dụng đúng tính năng (TDE, Data Masking) sẽ có được một cơ sở dữ liệu rất an toàn.
- Scalability (khả năng mở rộng): Có nhiều tùy chọn kỹ thuật để vận hành PostgreSQL trên quy mô lớn.

Những hạn chế của PostgreSQL:

- Database structure (cơ sở dữ liệu có cấu trúc): Khi thực hiện truy vấn dữ liệu trong SQL, phải có những yêu cầu rất nghiêm ngặt đối với dữ liệu lưu trữ trong bảng cơ sở dữ liệu. Chúng ta không thể có thêm trường dữ liệu so với cấu trúc bảng dữ liệu mà chúng ta đã định nghĩa trước đó.

- Open source (mã nguồn mở): PostgreSQL là một ứng dụng cơ sở dữ liệu mã nguồn mở, do đó nó không được sở hữu bởi một tổ chức nhất định. Mặc dù có nhiều tính năng và khả năng vượt trội nhưng nó gặp khó khăn so với những phần mềm độc quyền có toàn quyền kiểm soát và bản quyền đối với sản phẩm. Do đó, PostgreSQL không đi kèm với bảo hành và không có trách nhiệm pháp lý hoặc bảo vệ bồi thường.
- Slower performance (Hiệu suất chậm): Có nhiều vấn đề về hiệu suất, sao lưu và khôi phục cần phải đổi mới khi sử dụng PostgreSQL. Khi thực hiện một lệnh truy vấn, PostgreSQL - vì lý do cấu trúc cơ sở dữ liệu quan hệ, nó bắt đầu tìm kiếm với hàng đầu tiên và duyệt qua toàn bộ bảng để tìm dữ liệu liên quan. Do đó hiệu suất chậm hơn, và đặc biệt là khi có một số lượng lớn dữ liệu được lưu trữ trong các hàng và cột của bảng chứa nhiều trường thông tin bổ sung để so sánh.

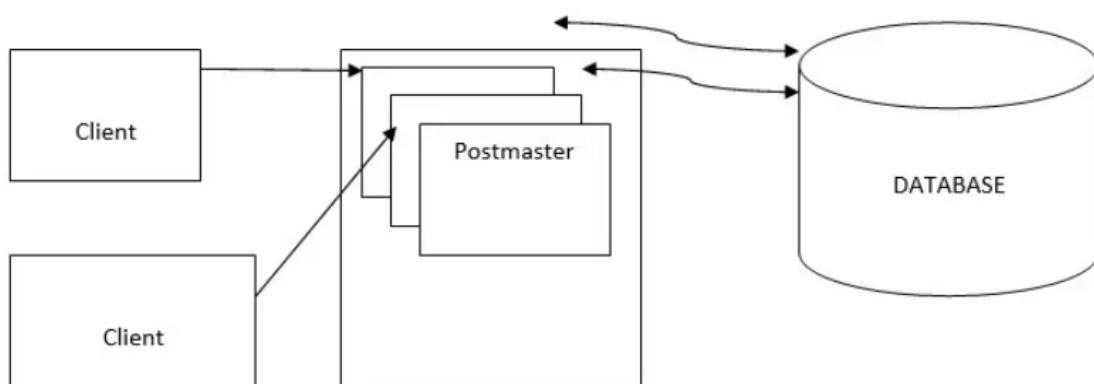
1.2 Architectural Fundamentals

PostgreSQL sử dụng mô hình client/server. Một PostgreSQL session bao gồm những processes phối hợp sau đây:

- Một server process quản lý tập tin cơ sở dữ liệu, cho phép kết nối đến cơ sở dữ liệu từ ứng dụng phía client, và thực hiện các thao tác trên cơ sở dữ liệu thay cho clients. Chương trình máy chủ cơ sở dữ liệu được gọi là postgres.
- Ứng dụng phía người dùng (frontend) muốn thực hiện các tác vụ trên cơ sở dữ liệu. Ứng dụng người dùng có thể rất đa dạng trong tự nhiên: client có thể là một công cụ hướng văn bản, một ứng dụng đồ họa, một máy chủ web truy cập cơ sở dữ liệu để hiển thị nội dung trang web, hoặc là một công cụ bảo trì cơ sở dữ liệu chuyên dụng. Một số ứng dụng khách được cung cấp với bản phân phối PostgreSQL, hầu hết được phát triển bởi người dùng.

Là một ứng dụng client/server, máy khách và máy chủ có thể nằm trên các hosts khác nhau. Trong trường hợp đó, chúng giao tiếp với nhau thông qua phương thức kết nối mạng TCP/IP.

Máy chủ PostgreSQL có thể thực hiện yêu cầu kết nối cho nhiều máy khách cùng lúc. Để làm được điều đó, nó sử dụng "forks" một tiến trình mới cho mỗi kết nối. Chính vì thế, máy khách và tiến trình mới trên máy chủ giao tiếp với nhau mà không có bất kỳ sự can thiệp nào bởi tiến trình postgres gốc.



Hình 1: Kiến trúc dựa trên mô hình Client-Server

1.3 Indexing

Chỉ mục là một cách thức phổ biến giúp tăng hiệu suất cơ sở dữ liệu. Một chỉ mục cho phép cơ sở dữ liệu máy chủ tìm kiếm và truy xuất những hàng trong bảng nhanh hơn so với khi không sử dụng chỉ mục. Tuy nhiên, chỉ mục cũng làm gia tăng chi phí trên hệ thống cơ sở dữ liệu, vì vậy chúng ta cần sử



dùng một cách hợp lý.

Ví dụ chúng ta có một bảng như sau:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

Và ứng dụng đưa ra nhiều truy vấn dạng:

```
SELECT content FROM test1 WHERE id = constant;
```

Nếu không có sự chuẩn bị trước, hệ thống phải duyệt qua toàn bộ bảng test1, từng dòng một, để tìm tất cả những hàng tương ứng với điều kiện truy vấn. Nếu có nhiều hàng trong test1 và chỉ có một vài hàng sẽ được trả về cho một lệnh truy vấn, nó rõ ràng là một phương thức không hiệu quả. Tuy nhiên nếu hệ thống được hướng dẫn để duy trì chỉ số trên cột id, nó có thể là một phương pháp hiệu quả hơn để tìm kiếm những dòng hợp với điều kiện. Chẳng hạn như, nó chỉ cần phải đi qua một số cấp bậc trong một cây tìm kiếm.

Một hướng tiếp cận tương tự được sử dụng trong phần lớn các sách: các thuật ngữ và khái niệm người đọc tìm kiếm được thu thập và sắp xếp theo chỉ mục bảng chữ cái ở phía cuối của quyển sách. Người đọc có thể quét chỉ mục tương đối nhanh chóng và lật đến trang thích hợp, hơn là phải đọc toàn bộ quyển sách để tìm nội dung quan tâm. Tương tự như là công việc của tác giả để dự đoán những mục, nội dung nào người đọc thường tìm kiếm, nhiệm vụ của người phát triển cơ sở dữ liệu là nhìn thấy trước được những chỉ mục nào sẽ hữu dụng cho cơ sở dữ liệu.

Câu lệnh sau đây có thể sử dụng để trao ra chỉ mục trên cột id của bảng trên:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Để xóa chỉ mục, sử dụng câu lệnh:

```
DROP INDEX
```

Khi chỉ mục được tạo, không cần bất kỳ sự can thiệp nào khác: hệ thống sẽ tự động cập nhật chỉ mục khi bảng có sự thay đổi, nó sẽ sử dụng chỉ mục trong các lệnh truy vấn khi xem xét việc dùng chỉ mục hiệu quả hơn so với duyệt tuần tự qua toàn bộ bảng. Tuy nhiên, người dùng cần phải chạy lệnh ANALYZE thường xuyên để cập nhật số liệu cho phép bộ lập kế hoạch truy vấn đưa ra quyết định phù hợp cho các câu lệnh truy vấn.

Chỉ mục cũng có lợi cho các lệnh UPDATE và DELETE với các điều kiện tìm kiếm. Hơn thế, chỉ mục cũng có thể được sử dụng cho các tìm kiếm trong phép kết. Do đó, một chỉ mục được định nghĩa trên một cột là một phần của điều kiện kết có thể tăng tốc đáng kể các truy vấn với phép kết.

Tạo một chỉ mục trên một bảng lớn có thể tốn nhiều thời gian. Một cách mặc định, PostgreSQL cho phép đọc (câu lệnh SELECT) diễn ra trên bảng song song với việc tạo chỉ mục, nhưng với việc ghi (INSERT, UPDATE, DELETE) thì bị khóa lại cho đến khi việc xây dựng chỉ mục hoàn thành.

Sau khi chỉ mục được tạo, hệ thống sẽ giữ cho nó đồng bộ với dữ liệu trong bảng. Việc này làm tăng chi phí. Do đó, các chỉ mục hiếm khi hoặc chưa bao giờ được sử dụng trong truy vấn nên được loại bỏ.

1.3.1 Index Types (các loại chỉ mục)

PostgreSQL cung cấp một vài loại chỉ mục như: B-tree, Hash, GiST, SP-GiST, GIN, BRIN. Mỗi loại sử dụng những giải thuật khác nhau để phù hợp với các kiểu truy vấn khác nhau. Theo mặc định, câu lệnh CREATE INDEX sẽ tạo ra chỉ mục B-tree, nó phù hợp với phần lớn các trường hợp. Những danh chỉ mục khác bằng cách viết từ khóa USING và phía sau là tên loại chỉ mục. Ví dụ, để tạo ra một chỉ mục dạng Hash:

```
CREATE INDEX name ON table USING HASH (column);
```



1.3.1.a B-Tree

B-trees có thể xử lý các truy vấn đẳng thức và phạm vi trên dữ liệu có thể được sắp xếp thứ tự. Trong thực tế, bộ lên kế hoạch truy vấn PostgreSQL sẽ xem xét sử dụng chỉ mục B-tree khi cột chỉ mục bao gồm một sự so sánh một trong các toán tử: $<$, \leq , $=$, \geq , $>$.

Các cấu trúc tương đương với sự kết hợp của các toán tử này, như BETWEEN and IN, cũng có thể được triển khai với tìm kiếm chỉ mục B-tree. Ngoài ra, điều kiều IS NULL hoặc IS NOT NULL trên cột chỉ mục cũng có thể được sử dụng với chỉ mục B-tree.

Trình tối ưu hóa cũng có thể sử dụng chỉ mục cây B cho các truy vấn liên quan đến toán tử khớp mẫu LIKE và nếu mẫu là hằng số và được neo vào đầu chuỗi, ví dụ: col LIKE 'foo%' hoặc col '^foo', nhưng không phải col LIKE '%bar'. Tuy nhiên, nếu cơ sở dữ liệu không sử dụng ngôn ngữ C, bạn sẽ phải cần tạo chỉ mục bằng một lớp toán tử đặc biệt để hỗ trợ chỉ mục cho các truy vấn khớp mẫu.

Chỉ mục B-tree có thể được sử dụng để truy xuất dữ liệu theo một thứ tự đã được sắp xếp. Nó không phải lúc nào cũng nhanh hơn cách duyệt thông thường và sắp xếp, nhưng nó thường hữu ích.

1.3.1.b Hash

Chỉ mục băm lưu trữ mã băm 32-bit lấy từ giá trị của cột được lập chỉ mục. Do đó, chỉ mục này chỉ có thể xử lý các truy vấn với điều kiện so sánh bằng. Bộ lên kế hoạch truy vấn sẽ xem xét sử dụng chỉ mục băm khi cột được chỉ mục liên quan đến phép so sánh với toán tử bằng: $=$

1.3.1.c GiST

Chỉ mục GiST không phải là một dạng chỉ mục duy nhất, mà là một cơ sở hạ tầng trong đó có thể triển khai nhiều chiến lược lập chỉ mục khác nhau. Theo đó, các toán tử cụ thể mà chỉ mục GiST có thể được sử dụng khác nhau tùy thuộc vào chiến lược lập chỉ mục (lớp toán tử). Ví dụ: phân phối tiêu chuẩn của PostgreSQL bao gồm các lớp toán tử GiST cho một số kiểu dữ liệu hình học hai chiều, hỗ trợ các truy vấn lập chỉ mục bằng cách sử dụng các loại toán tử này: \ll , $\&<$, $\&>$, \gg , $\ll|$, $\&<|$, $|&>$, $|>$, $<$, $\sim=$, $\&&$

Những lớp toán tử GiST khác săn có trong contrib hoặc các dự án riêng biệt.

Chỉ mục GiST cũng có thể tối ưu tìm kiếm "nearest-neighbor", như:

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

Câu lệnh trên sẽ tìm 10 nơi gần nhất với điểm được cho. Khả năng làm được điều đó phụ thuộc vào các lớp toán tử nhất định được sử dụng.

1.3.1.d SP-GiST

chỉ mục SP-GiST, tương tự như chỉ mục GiST, cung cấp kiến trúc hạ tầng hỗ trợ nhiều loại tìm kiếm khác nhau. SP-GiST cho phép triển khai một loạt các cấu trúc dữ liệu trên đĩa không cân bằng khác nhau, chẳng hạn như cây từ gốc, cây k-d và cây cơ sở (thứ). Ví dụ, phân phối tiêu chuẩn của PostgreSQL bao gồm các lớp toán tử SP-GiST cho các điểm hai chiều, hỗ trợ các truy vấn được lập chỉ mục sử dụng những toán tử: \ll , \gg , $\sim=$, $<$, $\ll|$, $|>$

Tương tự như GiST, SP-GiST hỗ trợ tìm kiếm "nearest-neighbor".

1.3.1.e GIN

Chỉ mục GIN là một chỉ mục đảo ngược, phù hợp với các giá trị dữ liệu chứa nhiều giá trị thành phần, chẳng hạn như mảng. Chỉ mục đảo ngược chứa một mục nhập riêng cho từng giá trị thành phần và có thể xử lý hiệu quả các truy vấn kiểm tra sự hiện diện của các giá trị thành phần cụ thể.

Tương tự như GiST và SP-GiST, GIN có thể hỗ trợ nhiều chiến lược lập chỉ mục do người dùng xác định khác nhau, và các toán tử cụ thể mà chỉ mục GIN sử dụng tùy thuộc vào chiến lược lập chỉ mục. Ví dụ, phân phối chuẩn của PostgreSQL bao gồm một lớp toán tử GIN cho các mảng, hỗ trợ các truy



vẫn được lập chỉ mục bằng cách sử dụng các toán tử sau: $<$, $>$, $=$, $\&\&$.

1.3.1.f BRIN

Chỉ mục BRIN (viết tắt của Block Range Indexes) lưu trữ tổng quát về các giá trị được lưu trữ trong phạm vi khối vật lý liên tiếp của bảng. Do đó, chúng có hiệu quả tốt nhất đối với các cột có giá trị tương quan tốt với thứ tự vật lý của các hàng trong bảng. Tương tự GiST, SP-GiST, SP-GiST và GIN, BRIN có thể hỗ trợ nhiều chiến lược lập chỉ mục khác nhau, và những toán tử cụ thể mà chỉ mục BRIN có thể sử dụng tùy thuộc vào chiến lược lập chỉ mục. Đối với các kiểu dữ liệu có thứ tự sắp xếp tuyến tính dữ liệu được lập chỉ mục tương ứng với giá trị nhỏ nhất và lớn nhất của các giá trị trong cột cho mỗi một khối. Điều này hỗ trợ các truy vấn được lập chỉ mục bằng cách sử dụng các toán tử sau: $<$, $<=$, $=$, $>=$, $>$.

1.3.2 Multicolumn Indexes (chỉ mục đa cột)

Một chỉ mục có thể được định nghĩa trên nhiều cột của một bảng. Lấy ví dụ, nếu chúng ta có một bảng như sau:

```
CREATE TABLE test2 (
    major int,
    minor int,
    name varchar
);
```

và bạn thường xuyên truy vấn như sau:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

Có thể thích hợp xác định một chỉ mục trên các cột chính và phụ cùng nhau, ví dụ:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

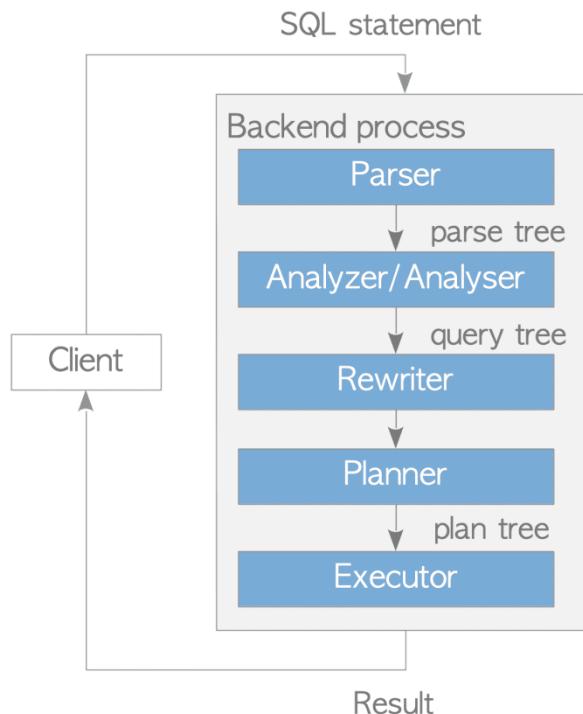
Một cách đồng thời, chỉ có kiểu chỉ mục B-tree, GiST, GIN, BRIN hỗ trợ chỉ mục đa cột. Việc có thể có nhiều cột chính hay không thì không phụ thuộc vào việc có thể INCLUDE cột vào chỉ mục hay không. Các chỉ mục có thể có tối đa 32 cột, bao gồm cả INCLUDE cột (giới hạn này có thể được thay đổi khi xây dựng PostgreSQL).

Một chỉ mục đa cột B-tree có thể được sử dụng với các điều kiện truy vấn bao gồm các tập con của các cột được chỉ mục, nhưng chỉ mục hoạt động hiệu quả nhất khi có các ràng buộc đối với các cột đầu (ngoài cùng bên trái). Quy tắc chính xác là các ràng buộc phải đặt trên các cột dẫn đầu, cộng với bất kỳ ràng buộc bắt đầu nào trên cột đầu tiên không có ràng buộc bắt đầu, sẽ được sử dụng để giới hạn phần của chỉ mục được quét. Các ràng buộc đối với các cột ở bên phải của các cột này được kiểm tra trong chỉ mục, vì vậy chúng lưu lượt truy cập vào bảng, nhưng không làm giảm phần chỉ mục phải được quét. Ví dụ: một chỉ mục trên (a, b, c) và một điều kiện truy vấn WHERE a = 5 AND b >= 42 AND c > 77, chỉ mục sẽ phải được quét từ mục nhập đầu tiên với a = 5 và b = 42 cho đến mục nhập cuối cùng có a = 5. Các mục nhập chỉ mục có c >= 77 sẽ được bỏ qua, nhưng chúng vẫn phải được quét qua. Về nguyên tắc, chỉ mục này có thể được sử dụng cho các truy vấn có ràng buộc đối với b và/hoặc c mà không có ràng buộc đối với a - nhưng toàn bộ chỉ mục sẽ phải được quét, vì vậy, trong hầu hết các trường hợp, bộ lập kế hoạch sẽ ưu tiên quét bảng tuần tự hơn là sử dụng chỉ mục.

Các chỉ mục đa cột nên được sử dụng một cách hợp lý. Trong hầu hết các trường hợp, chỉ mục trên một cột là đủ, tiết kiệm không gian lưu trữ và thời gian. Các chỉ mục có nhiều hơn ba cột ít có công dụng hơn trừ khi việc sử dụng bảng cực kỳ đặc biệt.

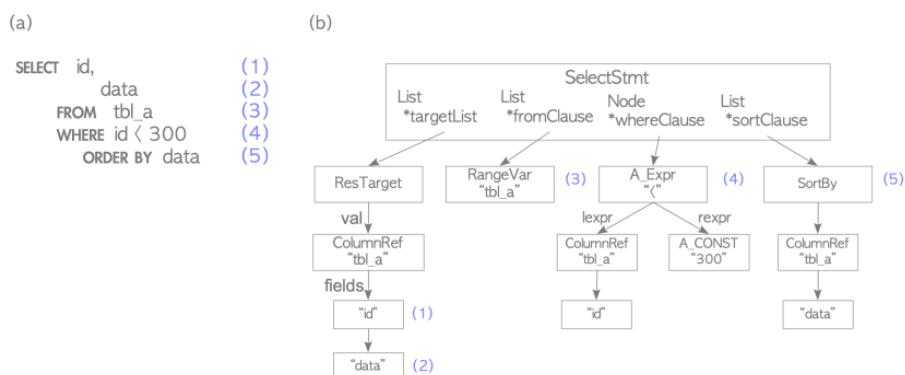
1.4 Query processing

Giống như hầu hết các DBMS khác, quá trình thực thi truy vấn (query processing) trong PostgreSQL cũng trải qua nhiều giai đoạn và mỗi giai đoạn đều có những chức năng cụ thể.



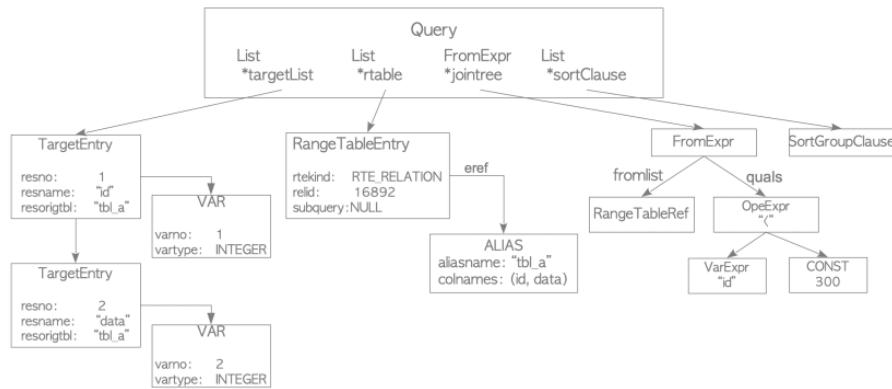
Hình 2: Quá trình xử lý truy vấn

- **Parser:** Bước đầu tiên là tạo ra một cây phân tích cú pháp, cho phép đọc cây phân tích cú pháp cho các hệ thống con, từ một câu lệnh SQL. Nhiệm vụ của bước này gần như là đầu tiên với mọi bộ phân tích ngôn ngữ, đó là phân tích câu lệnh đầu vào về cú pháp và ngữ nghĩa (ở đây là của ngôn ngữ SQL). Mỗi từ khóa hoặc định danh sẽ kích hoạt mã thông báo được tạo và chuyển đến bước tiếp theo.



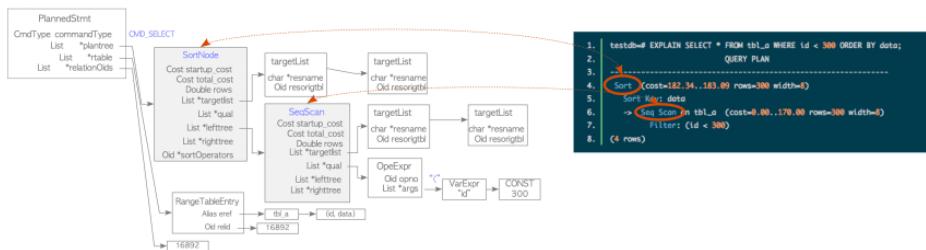
Hình 3: Ví dụ về bước parser

- **Analyser:** Bộ phân tích về cơ bản thực hiện đánh giá ngữ nghĩa của cây bộ phân tích cú pháp được tạo ở bước đầu tiên và nếu mọi thứ đều ổn, bộ phân tích sẽ tạo một cây truy vấn. Nhiệm vụ của bước này và bước đầu tiên quan hệ chặt chẽ với nhau và chúng gần như là đầu tiên với mọi bộ phân tích ngôn ngữ, đó là phân tích câu lệnh đầu vào về cú pháp và ngữ nghĩa (ở đây là của ngôn ngữ SQL).



Hình 4: Cây truy vấn được tạo ra sau khi hoàn thành 2 bước đầu

- Rewriter:** Trình viết lại truy vấn viết lại văn bản truy vấn bằng cách áp dụng các quy tắc cho nó. Trình viết lại xem xét các quy tắc, sau đó chuyển truy vấn đã sửa đổi tới trình lập kế hoạch truy vấn. Bảo mật cấp hàng được triển khai ở giai đoạn này. Ví dụ: các quy tắc trên SELECT luôn được áp dụng ở bước cuối cùng, bao gồm các truy vấn INSERT, UPDATE và DELETE.
- Planner:** Cây truy vấn được tạo bởi trình ghi lại được Planner nhận và nó tạo ra một cây truy vấn mà Executor có thể xử lý với hiệu suất tốt (còn gọi là plan tree).



Hình 5: Plan tree là công cụ để giao tiếp với Executor

- Executor:** Chức năng chính của bộ thực thi là đọc và ghi các bảng và đặt nó trong cụm cơ sở dữ liệu bởi trình quản lý bộ đệm. Và khi xử lý các truy vấn, người thực thi có thể sử dụng các vùng bộ nhớ được cấp phát trước và cũng có thể tạo các tệp tạm thời nếu cần.

1.5 Optimization

Nhiệm vụ của bộ lên kế hoạch thực thi và bộ tối ưu hóa là tạo ra một kế hoạch thực thi tối ưu. Một câu truy vấn SQL có thể được thực thi theo nhiều cách khác nhau, tuy nhiên mỗi cách đều sẽ tạo ra cùng một kết quả. Nếu có tính khả thi, trình tối ưu hóa truy vấn sẽ kiểm tra từng kế hoạch thực thi này, cuối cùng chọn kế hoạch thực thi dự kiến sẽ chạy nhanh nhất.

Trong một số trường hợp, việc kiểm tra từng kế hoạch thực thi có thể sẽ mất nhiều thời gian và bộ nhớ. Đặc biệt là khi thực hiện các truy vấn liên quan đến số lượng lớn các hoạt động kết. Để xác định kế hoạch truy vấn hợp lý (không nhất thiết phải tối ưu) trong một khoảng thời gian hợp lý, PostgreSQL sử dụng Trình tối ưu hóa truy vấn di truyền khi số lần kết vượt quá ngưỡng.

Quy trình tìm kiếm của bộ lập kế hoạch thực sự hoạt động với cấu trúc được gọi là đường dẫn, chỉ đơn giản là biểu diễn các cắt giảm của các kế hoạch chỉ chứa thông tin mà bộ kế hoạch cần để đưa ra quyết định. Sau khi xác định được con đường rẻ nhất, một cây kế hoạch chính thức được xây dựng để chuyển cho bộ thực thi thực hiện việc truy vấn. Việc này thể hiện kế hoạch thực thi mong muốn đủ chi tiết để bộ thực thi chạy nó.



Trình lập kế hoạch và trình tối ưu hóa bắt đầu bằng việc tạo các kế hoạch để quét từng mối quan hệ (bảng) riêng lẻ được sử dụng trong câu truy vấn. Các kế hoạch có thể được xác định bởi các chỉ mục có sẵn trên mỗi quan hệ. Như các DBMS khác PostgreSQL luôn có khả năng thực hiện quét tuần tự trên một quan hệ, do đó, kế hoạch quét tuần tự luôn được tạo. Giả sử một chỉ mục được định nghĩa trên một quan hệ (ví dụ: chỉ mục cây B) và một truy vấn chứa hàng số ràng buộc relation.attribute OPR. Nếu relation.attribute trùng khớp với khóa của chỉ mục cây B và OPR là một trong những toán tử được liệt kê trong lớp toán tử của chỉ mục, thì một kế hoạch khác được tạo bằng cách sử dụng chỉ mục cây B để quét quan hệ. Nếu như có thêm các chỉ mục và các ràng buộc trùng khớp với khóa của chỉ mục, các kế hoạch tiếp theo sẽ được xem xét. Các kế hoạch quét chỉ mục cũng được tạo cho các chỉ mục có thứ tự sắp xếp có thể khớp với mệnh đề ORDER BY của truy vấn (nếu có) hoặc thứ tự sắp xếp có thể hữu ích cho việc merge join (xem bên dưới).

Nếu truy vấn yêu cầu kết hai hoặc nhiều hơn các quan hệ, kế hoạch để kết các quan hệ sẽ được xem xét sau khi đã xem xét từng kế hoạch riêng lẻ cho từng quan hệ. Có ba phép kết như sau:

- nested loop join (kết lồng): Quan hệ bên phải sẽ được quét một lần cho mỗi hàng trong quan hệ bên trái. Phép kết này đơn giản để thực hiện nhưng có thể sẽ tốn nhiều thời gian. (Tuy nhiên, nếu quan hệ bên phải có thể được quét bởi chỉ mục, thì đây có thể là một phép kết tốt. Nó có thể sử dụng giá trị của hàng hiện tại trong quan hệ bên trái như là khóa để quét chỉ mục cho quan hệ bên trái.)
- merge join: Mỗi quan hệ được sắp xếp trên thuộc tính kết trước khi thực hiện kết. Sau đó, hai quan hệ được quét một cách song song với nhau, và những hàng thỏa điều kiện kết được kết với nhau và trả về dưới dạng là một bản ghi kết. Loại kết này đặc biệt ở chỗ mỗi quan hệ chỉ cần duyệt qua một lần. Yêu cầu sắp xếp có thể đạt được thông qua sắp xếp ngoại hoặc quét quan hệ theo thứ tự thích hợp bằng chỉ mục trên thuộc tính kết.
- hash join: Quan hệ bên phải sẽ được quét trước và tải vào bên trong bảng băm, sử dụng thuộc tính kết như khóa băm. Tiếp theo, quan hệ phải được quét và mỗi giá trị thích hợp của mỗi hàng được tìm thấy được sử dụng làm khóa băm để định vị các hàng phù hợp trong bảng.

Nếu truy vấn sử dụng ít hơn gepo_threshold, thì một tìm kiếm gần như toàn diện được tiến hành để tìm ra trình tự kết tốt nhất. Bộ lập kế hoạch ưu tiên xem xét các phép kết bất kỳ giữa hai quan hệ tồn tại điều kiện nối tương ứng trong câu lệnh Where. Các phép kết không có điều kiện kết chỉ được xem xét khi không còn lựa chọn nào khác. Tất cả các kế hoạch có thể được tạo cho mọi cặp kết được bộ lập kế hoạch xem xét và kế hoạch (ước tính) tốn ít chi phí nhất sẽ được chọn.

Ngược lại, nếu như gepothreshold vượt mức, trình tự kết sẽ được xác định bằng kinh nghiệm. Nếu không thì quy trình vẫn như cũ.

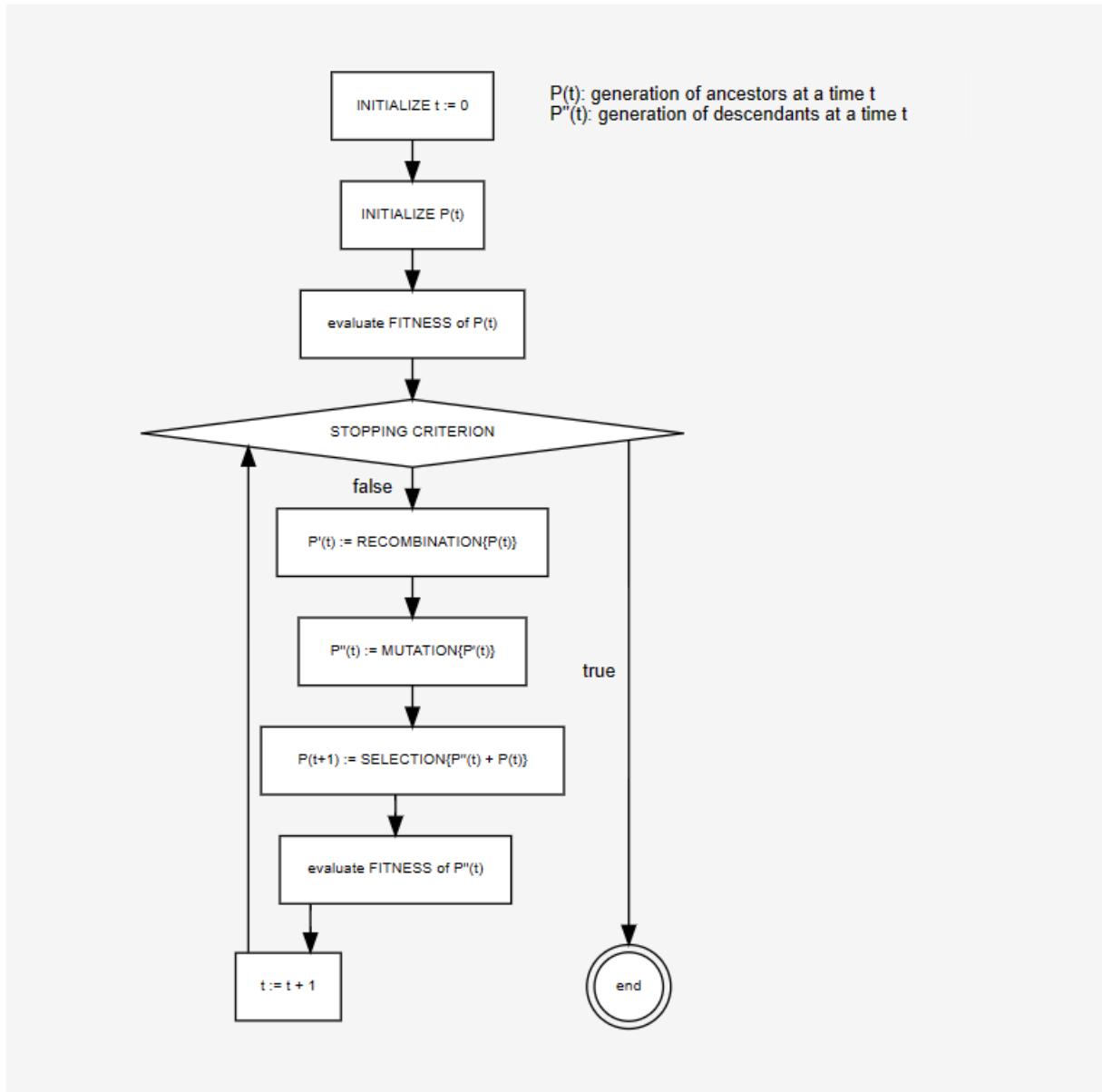
Cây kế hoạch cuối cùng bao gồm quét tuần tự hoặc quét trên chỉ mục của quan hệ, cộng với ba phép kết được nêu ở trên nếu cần thiết, cộng với bất kỳ bước phụ trợ nào nếu cần thiết (chẳng hạn như sắp xếp hoặc tính toán các hàm tổng hợp). Một trong những trách nhiệm của bộ lập kế hoạch là gắn các điều kiện từ câu lệnh WHERE và tính toán các biểu thức đầu ra cần thiết cho các node thích hợp nhất của cây kế hoạch.

1.5.1 Genetic Algorithms

Giải thuật Genetic Algorithms (GA) là một phương pháp tối ưu hóa dựa trên kinh nghiệm, nó thực hiện thông qua tìm kiếm ngẫu nhiên. Tập hợp các giải pháp khả thi cho bài toán tối ưu được coi là một tập hợp của các cá nhân. Mức độ thích ứng của một cá nhân với môi trường của nó được quy định bởi sự phù hợp của nó.

Tọa độ của một cá nhân trong không gian tìm kiếm được biểu diễn bằng các nhiễm sắc thể, về bản chất là một tập hợp các chuỗi ký tự. Gen là một phần phụ của nhiễm sắc thể mã hóa giá trị của một tham số duy nhất được tối ưu hóa. Mã hóa điển hình cho một gen có thể là số nhị phân hoặc số nguyên.

Qua mô phỏng các hoạt động tiến hóa tái tổ hợp, đột biến và chọn lọc, các thế hệ điểm tìm kiếm mới được tìm thấy cho thấy mức độ phù hợp trung bình cao hơn so với tổ tiên của chúng.



Hình 6: Structure of a Genetic Algorithm



Không thể nhấn mạnh quá mức rằng GA không phải là một tìm kiếm ngẫu nhiên thuần túy để tìm giải pháp cho một vấn đề. GA sử dụng các quy trình ngẫu nhiên, nhưng kết quả rõ ràng là không ngẫu nhiên (tốt hơn là ngẫu nhiên).

1.5.1.a Genetic Query Optimization (GEQO) in PostgreSQL

Mô hình GEQO tiếp cận vấn đề tối ưu hóa truy vấn như thế đó là vấn đề người bán hàng du lịch nổi tiếng (TSP). Các kế hoạch truy vấn có thể được mã hóa dưới dạng chuỗi số nguyên. Mỗi chuỗi đại diện cho thứ tự kết từ một quan hệ của truy vấn tới quan hệ tiếp theo. Ví dụ, cây tham gia được giải mã bởi

```
/\
/\ 2
/\ 3
4  1
```

Hình 7: *join tree*

chuỗi số nguyên 4-1-3-2; có nghĩa là kết '4' và '1', sau đó kết '3' và '2', trong đó 1, 2, 3, 4 là các ID của quan hệ trong bộ tối ưu hóa PostgreSQL.

Đặc điểm của việc thực thi GEQO trong PostgreSQL là:

- Việc sử dụng GA ở trạng thái ổn định (thay thế các cá thể kém phù hợp nhất trong quần thể, không phải thay thế cả thế hệ) cho phép hội tụ nhanh đối với các kế hoạch truy vấn được cải thiện. Điều này là cần thiết để xử lý truy vấn với thời gian hợp lý.
- Sử dụng phân tần tái tổ hợp cạnh đặc biệt phù hợp để giữ tổn thất cạnh thấp cho giải pháp TSP bằng GA
- Đột biến dưới dạng toán tử di truyền không được dùng nữa nên không cần cơ chế sửa chữa để tạo các chuyến tham quan TSP hợp pháp.

Mô hình GEQO cho phép bộ tối ưu hóa truy vấn của PostgreSQL hỗ trợ các câu truy vấn với số lượng lớn phép kết một cách hiệu quả mà không thông qua tìm kiếm toàn diện.

1.5.1.b Generating Possible Plans with GEQO (Sinh kế hoạch khả thi với GEQO)

Quy trình lập kế hoạch GEQO sử dụng môt lập kế hoạch tiêu chuẩn để tạo kế hoạch quét các mối quan hệ riêng lẻ. Sau đó, các kế hoạch kết được phát triển bằng cách sử dụng phương pháp di truyền. Như đã trình bày ở trên, mỗi kế hoạch kết ứng cử viên được biểu diễn bằng một trình tự để kết vào các quan hệ cơ sở. Trong giai đoạn đầu, môt GEQO chỉ đơn giản tạo ra một số trình tự kết có thể có một cách ngẫu nhiên. Đối với mỗi trình tự kết được xem xét, môt trình lập kế hoạch tiêu chuẩn được gọi để ước tính chi phí thực hiện truy vấn bằng cách sử dụng trình tự kết đó. (Đối với mỗi bước của trình tự kết, cả ba chiến lược kết khả thi đều được xem xét; và tất cả các kế hoạch quét quan hệ được xác định ban đầu đều khả dụng. Chi phí ước tính là rẻ nhất trong số các khả năng này). Các trình tự kết có chi phí ước tính thấp hơn được coi là “phù hợp” hơn so với những trình tự có chi phí cao hơn. Thuật toán di truyền loại bỏ các ứng cử viên ít phù hợp nhất. Sau đó, các ứng cử viên mới được tạo ra bằng cách kết hợp gen của các ứng cử viên phù hợp hơn - nghĩa là bằng cách sử dụng các phần được chọn ngẫu nhiên của các trình tự liên kết chi phí thấp đã biết để tạo ra các trình tự mới để xem xét. Quá trình này được lặp lại cho đến khi một số trình tự tham gia đặt trước đã được xem xét hết; sau đó cái tốt nhất được tìm thấy trong quá trình tìm kiếm được sử dụng để tạo kế hoạch hoàn chỉnh.

Quá trình này vốn không mang tính quyết định, bởi vì các lựa chọn ngẫu nhiên được thực hiện trong cả quá trình lựa chọn quần thể ban đầu và quá trình “đột biến” sau đó của các ứng cử viên tốt nhất. Để tránh những thay đổi đáng ngạc nhiên của gói đã chọn, mỗi lần chạy thuật toán GEQO sẽ khởi động lại trình tạo số ngẫu nhiên của nó với cài đặt tham số geqo_seed hiện tại.



1.5.2 EXPLAIN in PostgreSQL

Khi một câu lệnh được gửi đến máy chủ PostgreSQL để thực thi, nó phải phải được giải mã thành nhiều phần và xác định kế hoạch thực thi. Cho rằng dữ liệu có thể được truy xuất và quản lý/thao tác theo nhiều cách khác nhau, máy chủ PostgreSQL cố gắng tìm ra cách hiệu quả nhất để xây dựng tập kết quả cho truy vấn. Đặc biệt là trong trường hợp các truy vấn chạy trong thời gian dài có vấn đề hiệu suất, kế hoạch thực thi cho câu lệnh trên một nền tảng nhất định với cấu hình máy chủ hiện tại sẽ xác định hiệu suất truy vấn tổng thể (và trong hầu hết trường hợp là ứng dụng). Vì thế nó là quan trọng để hiểu bằng cách nào máy chủ PostgreSQL chia nhỏ việc thực thi câu truy vấn dưới dạng một kế hoạch thực thi. Để có thể làm được việc đó, PostgreSQL cung cấp cho người dùng câu lệnh "EXPLAIN" để mô tả quá trình thực hiện kế hoạch của một câu lệnh.

Quá trình truy vấn trong PostgreSQL được chia thành 5 components chính:

1. Parser
2. Analyzer
3. Rewriter
4. Planner
5. Executor

Với component (1) parser và (2) analyzer được quan tâm để bảo đảm rằng câu lệnh truy vấn được viết đúng cú pháp, và đầu ra là một cây truy vấn và được truyền vào component (3) rewriter để chuyển đổi dựa trên luật được định nghĩa bởi hệ thống luật pg_rules. Cây truy vấn sau khi được biến đổi được truyền vào component (4) planner nó sẽ sinh ra kế hoạch để component (5) thực thi.

Câu lệnh "EXPLAIN" chỉ giải thích câu lệnh, tuy nhiên khi chạy với lựa chọn "ANALYZE", nó cũng sẽ thực thi. Tập kết quả sẽ không bao giờ được quăng ra cho người dùng.

Chạy thử câu lệnh "EXPLAIN ANALYZE". Hãy nhìn vào kết quả của câu lệnh "EXPLAIN" tạo một bảng có tên là "foo_explain" và sau đó chạy câu lệnh "EXPLAIN" trên câu lệnh select trong bảng "foo_explain".

```
1 postgres=# CREATE TABLE foo_explain(A INT);
2 CREATE TABLE
3
4 postgres=# INSERT INTO foo_explain SELECT GENERATE_SERIES(1,50000);
5 INSERT 0 50000
6 postgres=# EXPLAIN SELECT * FROM foo_explain;
7           QUERY PLAN
8 -----
9      Seq Scan on foo_explain  (cost=1000000000.00..10000000015.00 rows=50000 width=4)
10     (1 row)
```

Kết quả câu lệnh EXPLAIN

Kế hoạch truy vấn là một điều cơ bản, nhưng có một số phần cần hiểu để có thể đánh giá được những việc mà bộ lên kế hoạch đã làm. Kế hoạch truy vấn cho chúng ta biết phương thức duyệt qua bảng, "Seq Scan" (duyệt tuần tự) trên bảng "foo_explain", chi phí khởi tạo để thực thi là "1000000000.00" và chi phí tổng cộng là "10000000015.00" cho "Seq Scan". Nó cũng đưa ra ước lượng bao nhiêu bản ghi được mong đợi trả về (50000) và kích cỡ của một bản ghi (4 bytes).

Chi phí ban đầu và chi phí tổng cộng là một con số bất kỳ nó cung cấp cho bộ lập kế hoạch phương tiện để có thể so sánh với các chiến lược khác để có thể thực hiện câu lệnh select một cách tối ưu. Tóm lại, chi phí thấp hơn biểu thị cho việc thực thi sẽ nhanh hơn.

Ước lượng sai



Cho đến hiện nay có còn tốt không? Điều thực sự quan trọng là phải hiểu rằng các hàng và độ rộng của hàng là ước tính chứ không phải là số thực. Vì vậy, có khả năng những điều này đôi khi có thể không chính xác. Sau đây là một ví dụ cụ thể.

```
1 postgres=# CREATE TABLE t(a INT, b VARCHAR(1));
2 CREATE TABLE
3 postgres=# INSERT INTO t VALUES(1, 'h');
4 INSERT 0 1
5 postgres=# SELECT * FROM t;
6   a | b
7   ---+-
8   1 | hello1
9  (1 row)
10
11 postgres=# EXPLAIN SELECT * FROM t;
12                                     QUERY PLAN
13 -----
14   Seq Scan on t  (cost=1000000000.00..10000000030.40 rows=2040 width=12)
15  (1 row)
```

Vì một lý do cũ, bộ lên kế hoạch cho rằng câu lệnh select trên sẽ trả về 2040 hàng và kích thước 12 bytes cho mỗi hàng, tổng cộng là 24480 bytes. Các hàng và kích thước là các fields được dự đoán tốt nhất dựa trên một số dữ liệu bảng và thống kê của PostgreSQL. Có thể nhìn thấy chúng trong bảng "pg_class" và "pg_stats".

```
1 postgres=# SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname =
2   't';
3   relname | relkind | reltuples | relpages
4   -----+-----+-----+-----
5   t      | r      |       0 |       0
6  (1 row)
7
8 postgres=# SELECT attname, inherited, n_distinct,
9   postgres-#          array_to_string(most_common_vals, E'\n') as most_common_vals
10  postgres-# FROM pg_stats
11  postgres-# WHERE tablename = 't';
12  attname | inherited | n_distinct | most_common_vals
13  -----+-----+-----+
14  (0 rows)
```

Có thể nhìn thấy rõ ràng rằng, máy chủ nghĩ rằng không có tuple nào trong bảng, và do đó cũng không có "relpages" và không có dữ liệu giúp chúng ta ước lượng tốt hơn kết quả trả ra của câu lệnh select. Vấn đề này có thể được dễ dàng sửa bằng cách chạy câu lệnh "VACUUM ANALYZE" trong bảng.

```
1 postgres=# VACUUM ANALYZE t;
2 VACUUM
```

Thực hiện lại "pg_class" và "pg_stats"

```
1 postgres=# SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname =
2   't';
3   relname | relkind | reltuples | relpages
4   -----+-----+-----+-----
5   t      | r      |       1 |       1
6  (1 row)
7
8 postgres=# SELECT attname, inherited, n_distinct,
9   postgres-#          array_to_string(most_common_vals, E'\n') as most_common_vals
10  postgres-# FROM pg_stats
11  postgres-# WHERE tablename = 't';
12  attname | inherited | n_distinct | most_common_vals
13  -----+-----+-----+
14  a      | f      |      -1 |
15  b      | f      |      -1 |
16  (2 rows)
```

Thực thi lại câu lệnh "EXPLAIN"



```
1 postgres=# EXPLAIN SELECT * FROM t;
2                                     QUERY PLAN
3 -----
4     Seq Scan on t  (cost=1000000000.00..1000000001.01 rows=1 width=6)
5     (1 row)
```

Bây giờ có thể nhìn thấy kết quả thực tế của bảng. Chỉ có 1 hàng được trả về, như mong đợi, và kích thước của một hàng là 6 bytes ít hơn so với giá trị 12 bytes trước đó. Có thể kết luận rằng thiếu số liệu có thể dẫn đến một kế hoạch dưới mức tối ưu dẫn đến hiệu suất truy vấn kém.

Thực hiện truy vấn từ kết quả kế hoạch được đưa ra

Cho rằng kế hoạch sau đây không phải là kế hoạch phức tạp nhất, nhưng nó vẫn có một vài node khác nhau. Để làm cho nó đơn giản, truy vấn "EXPLAIN" được chạy với tùy chọn "VERBOSE ON".

```
1                                     QUERY PLAN
2 -----
3 GroupAggregate  (cost=10000000010.30..10000000010.35 rows=1 width=23)
4   Output: count(*), t1.id, t1.name, t2.id
5   Group Key: t1.id, t1.name, t2.id
6   Filter: (count(*) > 1)
7   ->  Sort  (cost=10000000010.30..10000000010.30 rows=2 width=15)
8     Output: t1.id, t1.name, t2.id
9     Sort Key: t1.id DESC, t1.name
10    -> Nested Loop (cost=10000000004.20..10000000010.29 rows=2 width=15)
11      Output: t1.id, t1.name, t2.id
12      Join Filter: (t1.id = t2.id)
13      -> Seq Scan on public.t1  (cost=1000000000.00..10000000001.25 rows=6
14      width=11)
15        Output: t1.id, t1.name
16        Filter: (t1.id < 8)
17        -> Materialize (cost=4.20..8.34 rows=8 width=4)
18          Output: t2.id
19          -> Bitmap Heap Scan on public.t2  (cost=4.20..8.30 rows=8 width=4)
20            Output: t2.id
21            Recheck Cond: (t2.id < 5)
22            -> Bitmap Index Scan on t2_idx_id  (cost=0.00..4.20 rows=8
23            width=0)
24              Index Cond: (t2.id < 5)
25 (20 rows)
```

Trước khi đọc kế hoạch thực thi, nhìn vào dòng số 4 có thể thấy được các cột được chọn. Câu lệnh select bắt đầu với "SELECT COUNT(*), t1.id, t1.name, t2.id"

Bây giờ thực hiện từng bước, bắt đầu với node lá và sau đó đi lên các node trên như tái thiết kế lại cây lệnh truy vấn:

- Dòng 20 chỉ ra rằng điều kiện lọc trên t2.id; ví dụ: nó có thể là một phần của câu lệnh WHERE; t2.id < 5.
- Từ dòng 16 - 22 chỉ ra rằng bảng t2 là một phần của câu truy vấn. Thêm vào đó, nó cũng chỉ ra rằng chỉ mục t2_idx_id trên bảng t2 được tận dụng trong kế hoạch.
- tương tự, dòng 13 - 15 chỉ ra rằng bảng t1 là một phần của truy vấn và nó có một điều kiện lọc t1.id < 8.
- Từ dòng 10 - 12 chỉ rằng có một phép kết nối với điều kiện kết là t1.id = t2.id.

Từ những phần trên của kế hoạch chỉ ra rằng câu lệnh select trông như:

```
1 SELECT COUNT(*), t1.id, t1.name, t2.id
2   FROM t1 INNER JOIN t2
3     ON t1.id = t2.id
4     WHERE t1.id < 8 AND t2.id < 5;
```



Lên tiếp một bậc của cây truy vấn chúng ta tìm thấy một node SORT trên t1.id DESC, t1.name. Lên nữa, chúng ta thấy câu lệnh GROUP BY trên t1.id, t1.name, t2.id với điều kiện lọc, ví dụ: câu lệnh HAVING với COUNT(*) > 1. Bây giờ chúng ta đã duyệt qua toàn bộ cây, chúng ta có thể viết được một câu lệnh SQL hoàn chỉnh.

```
1  SELECT COUNT(*), t1.id, t1.name, t2.id
2   FROM t1 INNER JOIN t2
3    ON t1.id = t2.id
4   WHERE t1.id < 8 AND t2.id < 5
5   GROUP BY t1.id, t1.name, t2.id
6   HAVING COUNT(*) > 1
7   ORDER BY t1.id DESC, t1.name;
```

Hãy đào sâu hơn một chút về câu lệnh "EXPLAIN", tìm hiểu thêm về chi phí, "EXPLAIN ANALYZE", và so sánh các kế hoạch trong nỗ lực để hiểu rõ hơn các tiềm năng để tối ưu hóa các truy vấn.

Set Up

Tạo ra 2 bảng để tìm hiểu rõ hơn về câu lệnh "EXPLAIN".

```
1  -- big_table has 100 unique rows
2  postgres=# CREATE TABLE big_table AS (SELECT GENERATE_SERIES(1,100) id, 'hello' || 
3  GENERATE_SERIES(1,100) AS name);
4  SELECT 100
5
6  -- small_table has 3 sets of 10 rows with the same data.
7  postgres=# CREATE TABLE small_table AS (SELECT (GENERATE_SERIES(0,29) % 10) + 1 id, 'hello' || ((GENERATE_SERIES(0,29) % 10) + 1) AS name);
8  SELECT 30
```

Các hàng được nhân bảng sẽ giúp sử dụng hàm hợp aggregate function trong câu lệnh select.

Costs - Chi phí

PostgreSQL sử dụng bộ tối ưu hóa dựa trên chi phí. Nó có nghĩa rằng có nhiều cách để một câu truy vấn thực thi. Máy chủ cần một phương thức để chọn một tùy chọn tốt nếu không phải là tốt nhất trong số nhiều tùy chọn có sẵn. Nó yêu cầu những chiến lược so sánh thực thi khác nhau. Vì vậy PostgreSQL ấn định chi phí cho nhiều chiến lược cạnh tranh và chọn những chiến lược có chi phí thấp hơn.

Chi phí ước lượng là một giá trị tùy ý nó được gán cho mỗi bước trong quá trình thực thi truy vấn dựa trên tải tài nguyên dự kiến mà bước đó có thể tạo ra. Theo tài liệu PostgreSQL, nó là một giá trị tùy ý cho ước tính chi phí của bộ lập kế hoạch. "seq_page_cost" GUC được sử dụng như một giá trị chi phí cơ bản. Đó là một ước tính tùy ý của một lần tìm nạp page của disk là một phần của một loạt các lần tìm nạp tuần tự. Chi phí cho những hoạt động khác được xác định trong sự so sánh với giá trị này. Xem phần "Planner Cost Constants" trong phần "postgresql.conf" để biết các thông số chi phí khác.

```
1  postgres=# EXPLAIN SELECT COUNT(*), bt.id, bt.name, st.id
2   FROM big_table bt INNER JOIN small_table st
3    ON bt.id = st.id
4   WHERE bt.id < 8 AND st.id < 5
5   GROUP BY bt.id, bt.name, st.id
6   HAVING COUNT(*) > 1
7   ORDER BY bt.id DESC, bt.name;
8
9
10
11
12
13
14
15
```

QUERY PLAN

```
-- GroupAggregate (cost=3.79..3.82 rows=1 width=24)
  Group Key: bt.id, bt.name, st.id
  Filter: (count(*) > 1)
    -> Sort (cost=3.79..3.80 rows=1 width=16)
      Sort Key: bt.id DESC, bt.name
        -> Hash Join (cost=1.50..3.78 rows=1 width=16)
```



```
16      Hash Cond: (bt.id = st.id)
17          -> Seq Scan on big_table bt  (cost=0.00..2.25 rows=6 width=12)
18              Filter: (id < 8)
19          -> Hash   (cost=1.38..1.38 rows=10 width=4)
20              -> Seq Scan on small_table st  (cost=0.00..1.38 rows=10 width
21                  =4)
22                  Filter: (id < 5)
(12 rows)
```

Có một số điểm quan trọng cần lưu ý liên quan đến chi phí:

- Đầu tiên và quan trọng nhất, chi phí là một loại dữ liệu kép giúp linh hoạt hơn trong việc thiết lập các giá trị tùy ý rã hoặc rất lớn.
- Chi phí là một phạm vi.

Giá trị đầu tiên là estimated startup cost. Nó là thời gian tiêu tốn trước khi giai đoạn output bắt đầu. Nó bao gồm tất cả thời gian tiêu tốn ở các node con. Nhìn vào chi phí khởi động cho node ở dòng 20 và chi phí khởi động cho node cha ở dòng 19. Node cha bắt đầu sau khi node con đã hoàn tất. Đó là sự hợp nhất dữ liệu; node cha đợi để thu thập tất cả các bản ghi trước khi bắt đầu hiện thực. Các node nào có giá trị chi phí khởi tạo là 0.00 thì chỉ ra rằng những node đó có thể bắt đầu gần như ngay lập tức.

Giá trị thứ hai được gọi là estimated total cost trước khi giai đoạn output có thể bắt đầu truyền các bản ghi đến node cha. Vì vậy, node cha ở dòng 19 nhận tất cả các bản ghi đủ tiêu chuẩn và sau đó phát sinh thêm chi phí 0.30.

Trong một vài trường hợp, node cha có thể bắt đầu nhận dữ liệu ngay từ node con. Xem xét đầu ra của dòng lệnh sau.

```
1 postgres=# EXPLAIN SELECT * FROM big_table, small_table;
2                                     QUERY PLAN
3 -----
4     Nested Loop  (cost=0.00..40.88 rows=3000 width=48)
5         -> Seq Scan on big_table  (cost=0.00..2.00 rows=100 width=12)
6             -> Materialize  (cost=0.00..1.45 rows=30 width=36)
7                 -> Seq Scan on small_table  (cost=0.00..1.30 rows=30 width=36)
8 (4 rows)
```

Nó là phép kết tích descartes giữa bảng big_table và small_table. Có thể thấy rằng, node "Materialize" bắt đầu nhận dữ liệu khi quá trình duyệt tuần tự bắt đầu trên bảng small_table. Nested Loop - kết lòng cũng bắt đầu cùng thời điểm.

Chi phí thực sự là một thước đo tương đối về khả năng các nguồn tài nguyên được sử dụng. Nó chỉ ra những điểm cần cải thiện.

Optimizing Cost - tối ưu chi phí

Để tối ưu hiệu suất, có thể hiệu chỉnh chi phí được đặt trong tệp "postgresql.conf" theo phần cứng. Ví dụ, "random_page_cost" được ước tính có chi phí gấp 4 lần so với truy cập tuần tự page. Nó là vì sự mong đợi kernel thực hiện đọc trước tối ưu hóa và do đó giảm được chi phí truy nạp tuần tự. Chi phí truy cập trang ngẫu nhiên trên SSD sẽ thấp so với HDD. Có thể đặt chi phí cho một khung gian bảng nằm trên một thiết bị lưu trữ khác.

Explain Analyze

```
1 postgres=# EXPLAIN ANALYZE SELECT COUNT(*), bt.id, bt.name, st.id
2   FROM big_table bt INNER JOIN small_table st
3   ON bt.id = st.id
4   WHERE bt.id < 8 AND st.id < 5
5   GROUP BY bt.id, bt.name, st.id
6   HAVING COUNT(*) > 1
7   ORDER BY bt.id DESC, bt.name;
```



```
8                                     QUERY PLAN
9
10    GroupAggregate  (cost=3.79..3.82  rows=1  width=24) (actual time=2.499..2.829  rows=4
11      loops=1)
12        Group Key: bt.id, bt.name, st.id
13        Filter: (count(*) > 1)
14          ->  Sort  (cost=3.79..3.80  rows=1  width=16) (actual time=2.412..2.550  rows=12
15              loops=1)
16            Sort Key: bt.id DESC, bt.name
17            Sort Method: quicksort  Memory: 25kB
18            ->  Hash Join  (cost=1.50..3.78  rows=1  width=16) (actual time=1.683..1.990
19                rows=12  loops=1)
20                  Hash Cond: (bt.id = st.id)
21                  ->  Seq Scan on big_table bt  (cost=0.00..2.25  rows=6  width=12) (
22                      actual time=0.646..0.742  rows=7  loops=1)
23                      Filter: (id < 8)
24                      Rows Removed by Filter: 93
25                      ->  Hash  (cost=1.38..1.38  rows=10  width=4) (actual time=0.958..0.969
26                          rows=12  loops=1)
27                            Buckets: 1024  Batches: 1  Memory Usage: 9kB
28                            ->  Seq Scan on small_table st  (cost=0.00..1.38  rows=10  width
=4) (actual time=0.599..0.749  rows=12  loops=1)
29                                Filter: (id < 5)
30                                Rows Removed by Filter: 18
31
32    Planning Time: 2.358 ms
33    Execution Time: 3.093 ms
34
35    (18 rows)
```

Câu lệnh "EXPLAIN ANALYZE" cho chúng ta thời gian thực, thông tin các hàng và vòng lặp khi nó thực thi truy vấn. So sánh chi phí và thời gian cho node trên dòng 18 và 23, có thể thấy rằng chi phí 2.25 cần 0.742ms trong khi chi phí 1.38 cần 0.749ms.

Trong một vài trường hợp, truy vấn có thể thực thi trên một node nhiều lần. Chạy lại tích descartes với "ANALYZE", sẽ cần tìm nạp lại một trong hai bảng.

```
1  postgres=# EXPLAIN ANALYZE SELECT * FROM big_table, small_table;
2
3                                     QUERY PLAN
4
5    Nested Loop  (cost=0.00..40.88  rows=3000  width=48) (actual time=12.700..121.419  rows
=3000  loops=1)
6      ->  Seq Scan on big_table  (cost=0.00..2.00  rows=100  width=12) (actual time
=10.894..12.014  rows=100  loops=1)
7      ->  Materialize  (cost=0.00..1.45  rows=30  width=36) (actual time=0.017..0.376  rows
=30  loops=100)
8          ->  Seq Scan on small_table  (cost=0.00..1.30  rows=30  width=36) (actual time
=0.582..0.976  rows=30  loops=1)
9
10   Planning Time: 2.109 ms
11   Execution Time: 156.926 ms
12
13   (6 rows)
```

node ở dòng 6 thực thi 100 lần và đây thực sự là số hàng của bảng big_table.

Comparing Plans - So sánh kế hoạch thực thi

Chi phí giúp hiểu rõ những sự lựa chọn có sẵn khi thực thi một câu lệnh truy vấn SQL. Hãy lấy một ví dụ đơn giản lựa chọn giữa subquery và join. Trường hợp nào sẽ tốt hơn.

```
1  postgres=# SELECT * FROM big_table WHERE id IN (SELECT DISTINCT(id) FROM small_table)
2
3  postgres=# SELECT DISTINCT(big_table.*) FROM big_table INNER JOIN small_table ON
4  big_table.id = small_table.id ORDER BY id, name;
```

Truy vấn đầu tiên 4.537ms và truy vấn thứ hai 3.636ms.

2 Neo4j

2.1 Giới thiệu

Neo4j là hệ cơ sở dữ liệu NoSQL dạng đồ thị. Neo4j được giới thiệu vào năm 2007 và công bố chính thức vào năm 2010.

Neo4j không thực hiện lưu trữ dữ liệu dưới dạng bảng mà thay vào đó sẽ lưu trữ dưới dạng các node và sử dụng relation để liên kết các node lại với nhau.

Neo4j sử dụng ngôn ngữ Cypher là ngôn ngữ truy vấn đồ thị, cho phép người dùng lưu trữ và truy xuất dữ liệu từ cơ sở dữ liệu đồ thị. Cypher được dùng để truy vấn cũng như cập nhật các đồ thị (Graph). Cypher dựa trên nghệ thuật ASCII, vì vậy cú pháp của nó dễ hiểu và làm cho các truy vấn dễ hiểu hơn. Nó tập trung vào việc diễn đạt rõ ràng những gì cần truy xuất từ một biểu đồ, chứ không phải về cách truy vấn nó. Cypher Được coi là ngôn ngữ truy vấn đồ thị dễ tìm hiểu nhất.



Hình 8: Neo4j

2.1.1 Ưu điểm

- Hiệu suất cao: Neo4j có khả năng xử lý dữ liệu đồ thị với tốc độ nhanh hơn so với các hệ quản trị cơ sở dữ liệu quan hệ (RDBMS), đặc biệt là khi truy vấn dữ liệu với các quan hệ phức tạp và số lượng lớn các đối tượng. Điều này là do trong mô hình dữ liệu biểu đồ, truy vấn sẽ chỉ kiểm tra một phần của biểu đồ sẽ được truy vấn duyệt qua chứ không phải toàn bộ biểu đồ.
- Dữ liệu đa dạng: Neo4j có thể lưu trữ nhiều loại dữ liệu khác nhau, bao gồm các loại dữ liệu đồ thị như các nút (nodes), quan hệ (relationships) và thuộc tính (properties), và các loại dữ liệu khác như văn bản, hình ảnh, âm thanh, video, v.v.
- Truy xuất dữ liệu linh hoạt: Neo4j hỗ trợ nhiều ngôn ngữ truy vấn, bao gồm Cypher, Gremlin và SPARQL, cho phép truy vấn dữ liệu đồ thị với cấu trúc phức tạp.
- Đồng bộ hóa dữ liệu: Neo4j có khả năng đồng bộ hóa dữ liệu giữa các phiên bản khác nhau, cho phép dữ liệu được phân tán và truy cập trên nhiều nút khác nhau.
- Khả năng mở rộng: Neo4j có khả năng mở rộng để xử lý các tải trọng lớn và tăng cường hiệu suất. Vì mô hình biểu đồ rất linh hoạt nên ta có thể bắt đầu với mô hình nhỏ và dễ dàng cải thiện nó trong tương lai bằng cách thêm nhiều nút và mối quan hệ hơn với ít chi phí di chuyển và bảo trì hơn.

2.1.2 Nhược điểm

- Chi phí cao: Vì Neo4j là một công nghệ mới và phức tạp, nên chi phí để triển khai và bảo trì hệ thống Neo4j có thể cao.
- Khó khăn trong việc định hình cấu trúc dữ liệu: Với dữ liệu đồ thị phức tạp, việc thiết kế cấu trúc dữ liệu trong Neo4j có thể trở nên khó khăn và tốn thời gian.
- Lưu trữ: Neo4j có một số giới hạn giới hạn trên đối với kích thước biểu đồ và có thể hỗ trợ các biểu đồ đơn có hàng chục tỷ nút, mỗi quan hệ và thuộc tính. Phiên bản Neo4j hiện tại hỗ trợ tới khoảng 34 tỷ nút và mỗi quan hệ cũng như khoảng 274 tỷ thuộc tính. Điều này là khá đủ cho các biểu đồ lớn có kích thước biểu đồ mạng tương tự như Facebook. Trên thực tế, các hạn chế về bộ nhớ này không đặt ra bất kỳ giới hạn nào vì chỉ các doanh nghiệp lớn như Google mới có thể đẩy các giới hạn này và các giới hạn này được đặt để tối ưu hóa bộ nhớ và có thể tăng lên trong các phiên bản sau.
- Cấu hình phức tạp: Việc cấu hình Neo4j có thể trở nên phức tạp đối với những người mới sử dụng và đòi hỏi kỹ năng chuyên môn.



2.2 Indexing

Lập chỉ mục trong Neo4j đề cập đến việc tạo cấu trúc dữ liệu ánh xạ các giá trị thuộc tính tới các nút hoặc mỗi quan hệ có các giá trị đó. Cấu trúc dữ liệu này cho phép truy xuất hiệu quả các nút và mỗi quan hệ dựa trên các giá trị thuộc tính, cải thiện hiệu suất của các truy vấn liên quan đến các thuộc tính đó.

Trong Neo4j có các loại Indexing chính sau :

- Range index
- Lookup index
- Text index
- Point index
- Full-text index

2.2.1 Range index

Range index Là loại indexing dùng để tăng tốc độ truy vấn trong các câu truy vấn thuộc khoảng giá trị. Loại index này dùng để tăng tốc độ cho tình huống truy vấn đặc thù về khoảng giá trị chứ không phải là một số cụ thể. Range index sử dụng kiến trúc B+-tree để lưu trữ các giá trị thuộc tính của node.

Range index trong Neo4j cũng được xây dựng dựa trên cây B+ (B-plus tree) tương tự như Lookup index. Tuy nhiên, điểm khác biệt chính giữa hai loại index này là cách giá trị thuộc tính được lưu trữ trong cây B+.

Với Range index, cây B+ được sử dụng để tạo ra một bảng ánh xạ giữa các giá trị thuộc tính và các node trong Neo4j, tuy nhiên mỗi node được đại diện bởi một khoảng giá trị (range) của thuộc tính được tạo ra index. Các khoảng giá trị này được chia thành các phạm vi (range) dựa trên giá trị min và max của thuộc tính.

Khi thực hiện truy vấn với Range index, hệ thống sử dụng cây B+ để tìm tất cả các node có thuộc tính nằm trong khoảng giá trị được cung cấp. Các khoảng giá trị này có thể được xác định bằng cách sử dụng toán tử so sánh như "=", "<", "<=", ">=", ">".

Với Range index, tìm kiếm sẽ được thực hiện nhanh chóng hơn so với việc quét toàn bộ cơ sở dữ liệu để tìm các node phù hợp với điều kiện của truy vấn. Tuy nhiên, việc sử dụng Range index cũng có thể ảnh hưởng đến hiệu suất của cơ sở dữ liệu nếu không được sử dụng đúng cách, đặc biệt là khi cập nhật và thêm mới dữ liệu.

Để tạo range index trong Neo4j, ta có thể sử dụng mệnh đề CREATE INDEX trong Cypher. Dưới đây là ví dụ về cách tạo range index trên thuộc tính có tên là "dấu thời gian" cho các nút thuộc loại "timestamp":

Ví dụ :

Giả sử cơ sở dữ liệu hiện tại gồm nhiều node có label là Person và 2 thuộc tính gồm name và age

Tạo index:

```
CREATE RANGE INDEX IF NOT EXISTS my_index_name FOR (n:Person) ON (n.age)
```

Ở đây ta chọn trường age để sử dụng index vì age có kiểu dữ liệu int để so sánh >, <,...

Sử dụng index:

```
MATCH (p:Person)
WHERE p.age >= 20 AND p.age <= 40
RETURN p.name
```

2.2.2 Lookup index

Trong Neo4j, Lookup Index (hay còn gọi là Point Lookup Index) là một loại index sử dụng để tìm kiếm các nút (nodes) hoặc mỗi quan hệ (relationships) bằng một giá trị đơn (single value) của một thuộc tính (property) cụ thể. Ví dụ, nếu ta có một graph với các nút là các người dùng (User) và mỗi người dùng có một thuộc tính là "username", Lookup Index cho phép ta tìm kiếm một người dùng với username cụ thể một cách nhanh chóng và hiệu quả.



Lookup index trong Neo4j được xây dựng dựa trên cấu trúc cây B+ (B-plus tree). B+ tree là một loại cây tìm kiếm có cấu trúc dữ liệu phân cấp, mỗi nút có thể chứa nhiều giá trị dữ liệu và các nút đều được liên kết với nhau.

Với Lookup index, cây B+ được sử dụng để tạo ra một bảng ánh xạ giữa giá trị thuộc tính và node trong Neo4j. Cụ thể, mỗi node được đại diện bởi một giá trị duy nhất trong thuộc tính được tạo ra index. Khi thực hiện truy vấn, hệ thống sẽ sử dụng giá trị được cung cấp để tìm node tương ứng trong cây B+.

Sử dụng Lookup index có thể giúp tối ưu thời gian tìm kiếm trong Neo4j bằng cách cung cấp một cơ chế ánh xạ nhanh chóng từ giá trị thuộc tính đến node tương ứng, giảm thời gian tìm kiếm và truy cập dữ liệu.

Syntax:

```
CREATE INDEX <index_name> FOR (n:<node_label>) ON (n.<property_name>)
```

Ví dụ:

```
CREATE INDEX lookup_name FOR (n.Person) ON (n.name)
```

Sử dụng:

```
MATCH (n:Person) WHERE n.name = "abc" RETURN n
```

2.2.3 Text index

Text index trong Neo4j là một loại index dùng để tìm kiếm các từ hoặc cụm từ trong các thuộc tính chứa văn bản (text) của các node. Điều này rất hữu ích khi muốn tìm kiếm các nút có chứa các từ cụ thể hoặc có cụm từ giống nhau. Text index cho phép tìm kiếm theo phương pháp full-text search, trong đó tìm kiếm được thực hiện trên các từ được tách ra từ văn bản và bao gồm cả sự khớp mờ (fuzzy matching), phù hợp (matching) phụ âm (phonetic matching) và các quy tắc khác.

Kiến trúc của Text index trong Neo4j bao gồm 2 thành phần chính:

Inverted index: đây là thành phần chính của Text index, sử dụng để tạo ra các index term từ các nội dung của thuộc tính text trong các nút hoặc mối quan hệ. Ví dụ, khi một nút có thuộc tính "description" chứa nội dung "This is a sample description", thì các index term được tạo ra có thể là "this", "is", "a", "sample", "description". Mỗi index term này sẽ trỏ đến một hoặc nhiều nút hoặc mối quan hệ chứa nội dung tương ứng.

Phrase index: đây là một thành phần tùy chọn của Text index, được sử dụng để tạo ra các index cho các cụm từ hoặc câu hỏi trong nội dung text. Ví dụ, khi một nút chứa một mô tả sản phẩm như "A sleek and stylish laptop that is perfect for work and play", thì các cụm từ hoặc câu hỏi có thể được tạo ra bao gồm "sleek and stylish laptop", "perfect for work", "perfect for play", v.v. Các index term này sẽ trỏ đến một hoặc nhiều nút hoặc mối quan hệ chứa nội dung tương ứng.

Text index cung cấp tính năng tìm kiếm văn bản mạnh mẽ trong Neo4j, cho phép người dùng tìm kiếm các nút hoặc mối quan hệ dựa trên các cụm từ hoặc từ khóa xuất hiện trong nội dung văn bản của chúng.

Syntax:

```
CREATE TEXT INDEX [index_name] [IF NOT EXISTS]
FOR (n:LabelName)
ON (n.propertyName)
```

Ví dụ:

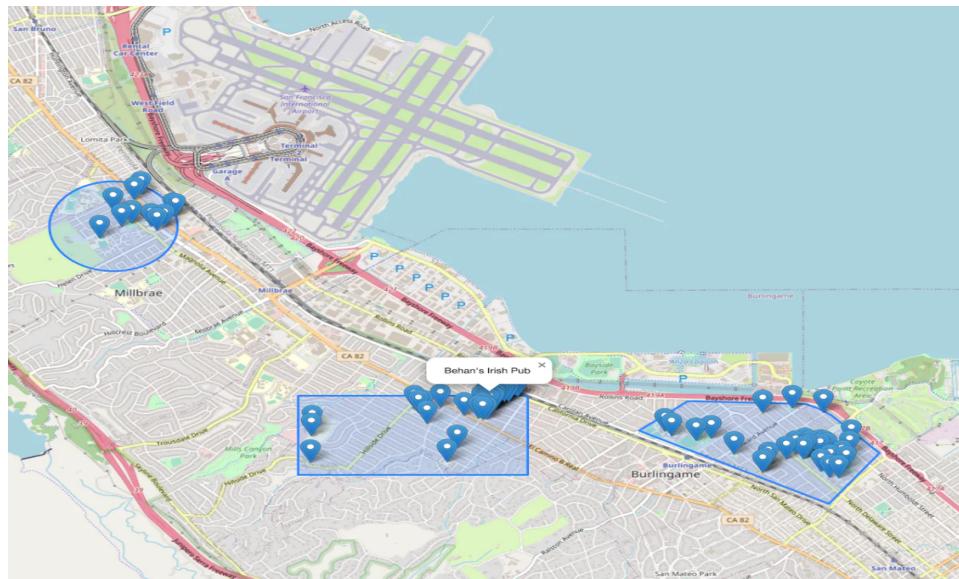
```
CREATE TEXT INDEX text_index IF NOT EXISTS
FOR (n:Person)
ON (n.name)
```

2.2.4 Point index.

Trong Neo4j, Point Index là 1 loại chỉ mục đặc biệt, chỉ mục đơn thuộc tính (single-property index) và chúng chỉ dùng để chỉ mục cho các giá trị điểm (Point values), không như range indexes.

Đối với chỉ mục Point index, Neo4j sử dụng đường cong lấp đầy trong không gian 2D hoặc 3D dựa trên 1 cây B+ tổng quát cơ bản. Các điểm sẽ được lưu trữ trong tối đa 4 cây khác nhau trong 4 hệ thống tham chiếu tọa độ (WGS-84, WGS-84-3D, Cartesian và Cartesian 3D). Điều này cho phép câu truy vấn với điều kiện bằng hoặc trong phạm vi được sử dụng chính xác với cùng 1 cú pháp như các loại thuộc tính khác.

Point index được thiết kế để tăng tốc độ truy vấn trong không gian, cụ thể là truy vấn dựa vào khoảng cách (distance) và hộp giới hạn (bounding box). Ứng dụng phổ biến của việc tìm kiếm không gian cho chỉ mục này là dùng để tìm những điểm nằm gần những điểm khác trong giới hạn vùng (có thể là 1 hộp hoặc 1 đa giác, phạm vi bán kính...). Ví dụ là việc tìm kiếm tất cả các địa điểm du lịch trong phạm vi 500m đối với điểm trường đại học Bách khoa thành phố Hồ chí Minh.



Hình 9: Ví dụ point index

Syntax:

```
CREATE POINT INDEX [index_name] [IF NOT EXISTS]
FOR (n:LabelName)
ON (n.propertyName)
[OPTIONS "{" option: value [, ...] "}]
```

Ví dụ:

```
CREATE POINT INDEX index_name IF NOT EXISTS
FOR (n:Person)
ON (n.name)
```

2.2.5 Full-text index.

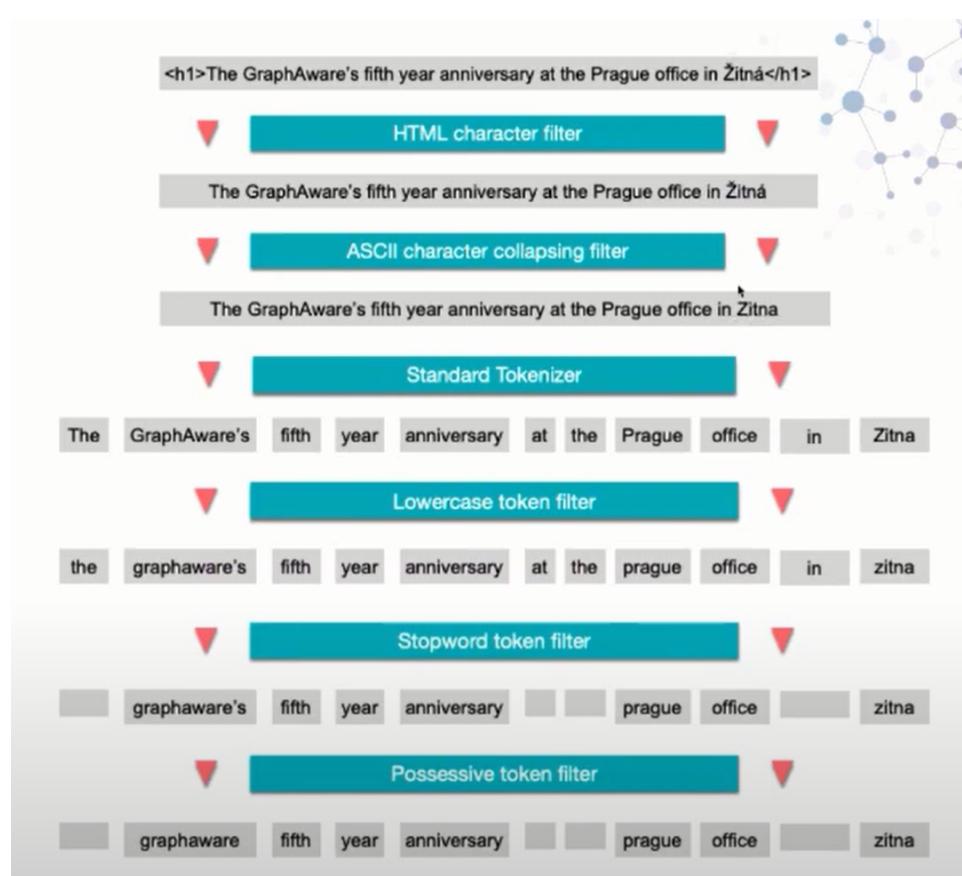
Full-text index là chỉ mục dùng để tối ưu cho việc tìm kiếm các văn bản.

Mặc dù các chỉ mục text và full-text index nhìn có vẻ như giải quyết các vấn đề giống nhau nhưng chúng thực chất có nhiều điểm khác biệt. Không giống như text index chỉ lập chỉ mục trên thuộc tính chuỗi đơn, full-text index có thể chỉ mục trên bất kì loại dữ liệu chuỗi nào (có thể trên nhiều thuộc tính). Text index giải quyết vấn đề tìm chuỗi giống 1 phần (substring match) và tìm chuỗi giống chính xác dựa trên ngữ nghĩa định nghĩa bởi Cypher. Trong khi đó, full-text index sử dụng công cụ phân tích tích hợp cung cấp xử lý ngôn ngữ cụ thể của văn bản. Chúng cho phép các truy vấn tinh vi hơn so với việc tìm chuỗi con giống 1 phần đơn giản (substring match). Tùy thuộc vào trình phân tích được sử dụng, full-text index có thể sử dụng cho các loại tìm kiếm văn bản khác nhau. Ngoài ra kết quả được sắp xếp thứ tự theo mức độ liên quan.

Cấu trúc dữ liệu và việc tìm kiếm đối chỉ mục full-text được thực hiện thông qua các bước cơ bản như: Phân tích (Analysis), chỉ số đảo ngược (Inverted Index), tìm kiếm(Searching) và đánh giá (Scoring)

- Phân tích (Analysis)

Phân tích là quá trình chuyển đổi văn bản thành các đơn vị nhỏ hơn và chính xác vì mục đích tìm kiếm: các token



Hình 10: Analysis

- Chỉ số đảo ngược (Inverted Index)

Chỉ số đảo ngược là cấu trúc dữ liệu của công cụ tìm kiếm. Nói 1 cách đơn giản thì nó ánh xạ các tài liệu cho các từ khóa như 1 bảng chú giải ở cuối cuốn sách.



Term dictionary		Postings List	
mastering	0	0	[0]
highly	1	1	[0]
distributed	2	2	[0]
architecture	3	3	[0]
graph	4	4	[1,3]
algorithms	5	5	[1,3]
in	6	6	[1,2,3]
neo4j	7	7	[1,2,3]
all	8	8	[1]
pairs	9	...	
shortest	10		
path	11		
announcing	12		
the	13		
graphtour	14		
8+	15		
cities	16		
worldwide	17		
single	18		
source	19		

Hình 11: Inverted index

- Tìm kiếm(Searching)

Khi chỉ mục được xây dựng, chúng ta có thể tìm kiếm chỉ mục đó dựa vào truy vấn và trình chỉ mục (IndexSearcher).

- Dánh giá (Scoring)

Hàm đánh giá mặc định của Apache Lucene dựa trên mô hình không gian vectơ tối ưu hóa cao. Hàm đánh giá phổ biến là độ tương đồng TFIDF.

$$w_{x,y} = tf_{x,y} \times \log \left(\frac{N}{df_x} \right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

Hình 12: Cách tính hàm TF.IDF

Ví dụ về trường hợp sử dụng full-text index là phân tích 1 cuốn sách cho 1 thuật ngữ nhất định và tận dụng kiến thức mà cuốn sách được viết bằng 1 ngôn ngữ nhất định. Việc sử dụng chương trình phân tích cho ngôn ngữ đó cho phép loại trừ các từ dừng, chẳng hạn như "if" và "and", và bao gồm các hình thức từ khác...

Trái với text và range index thì full-text index được truy vấn bằng các quy trình tích hợp. Tuy nhiên, chúng được tạo ra và hủy bỏ bằng cách dùng lệnh Cypher.

Syntax:

```
CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS]
FOR (n:LabelName[" | " ...])
ON EACH "[" n.propertyName[, ...] "]"
[OPTIONS "{" option: value[, ...] "}]
```

Ví dụ:

```
CREATE FULLTEXT INDEX titlesAndDescriptions FOR (n:Movie|Book) ON EACH [n.title, n.
description]
```

2.3 Query processing:



Hình 13: Neo4j Processing

Phân tích cú pháp: Phân tích cú pháp là quá trình phân tích cú pháp truy vấn và xây dựng cấu trúc dữ liệu dạng cây, được gọi là Cây cú pháp trừu tượng (AST), đại diện cho cấu trúc của truy vấn. Giai đoạn phân tích cú pháp kiểm tra các lỗi cú pháp, chẳng hạn như sử dụng từ khóa không chính xác, dấu chấm câu bị thiếu hoặc không liên quan và biểu thức không hợp lệ. AST sau đó được sử dụng để xử lý thêm truy vấn. Cây cấu trúc này bao gồm các phần tử như nodes, relationships và predicates.

Phân tích ngữ nghĩa: Phân tích ngữ nghĩa là quá trình kiểm tra tính hợp lệ của truy vấn theo lược đồ đồ thị và dữ liệu chứa trong đồ thị. Điều này liên quan đến việc kiểm tra xem các nút và mối quan hệ được tham chiếu có tồn tại không, truy vấn có hợp lý với cấu trúc biểu đồ không và truy vấn đó có được xây dựng đúng không. Phân tích ngữ nghĩa cũng có thể bao gồm kiểm tra kiểu, trong đó truy vấn được kiểm tra để đảm bảo rằng kiểu của biểu thức trong truy vấn nhất quán với kiểu dữ kiện.

Tối ưu hóa: Tối ưu hóa truy vấn là quá trình tìm kế hoạch thực hiện hiệu quả nhất cho một truy vấn nhất định. Trình tối ưu hóa truy vấn dựa trên chi phí của Neo4j xem xét nhiều kế hoạch thực hiện và ước tính chi phí của từng kế hoạch dựa trên số lượng nút, mối quan hệ và thuộc tính được truy cập. Trình tối ưu hóa sau đó chọn gói có chi phí thấp nhất, thường là gói nhanh nhất. Trình tối ưu hóa cũng có thể tính đến bất kỳ chỉ mục hoặc ràng buộc nào được xác định trên biểu đồ, điều này có thể cải thiện hiệu suất truy vấn..

Thực thi: Sau khi kế hoạch truy vấn được tối ưu hóa đã được chọn, truy vấn sẽ được thực hiện trên dữ liệu biểu đồ. Công cụ lưu trữ giao dịch của Neo4j sử dụng kết hợp các cấu trúc trong bộ nhớ và trên đĩa để thực hiện truy vấn một cách hiệu quả. Kế hoạch thực hiện có thể bao gồm các hoạt động khác nhau, chẳng hạn như truyền tải, lọc, sắp xếp và tổng hợp, được thực hiện theo thứ tự hiệu quả nhất để giảm thiểu số lượng hoạt động và giảm thời gian thực hiện tổng thể.

Xử lý kết quả: Bước cuối cùng trong xử lý truy vấn là xử lý kết quả. Kết quả có thể ở dạng nút, mối quan hệ, đường dẫn hoặc cấu trúc dữ liệu biểu đồ khác. Neo4j cung cấp nhiều API và công cụ khác nhau để xử lý và phân tích kết quả, bao gồm trực quan hóa biểu đồ, phân tích thống kê và học máy.

Nhìn chung, quá trình xử lý truy vấn của Neo4j được thiết kế để xử lý các truy vấn phức tạp trên các biểu đồ lớn một cách hiệu quả. Các phần khác nhau của quá trình xử lý truy vấn làm việc cùng nhau để đảm bảo rằng các truy vấn được thực hiện nhanh chóng và hiệu quả nhất có thể. Tuy nhiên, hiệu suất truy vấn có thể bị ảnh hưởng bởi độ phức tạp của truy vấn và kích thước của biểu đồ, vì vậy, điều quan trọng là phải tối ưu hóa các truy vấn và cấu trúc dữ liệu biểu đồ bên dưới càng nhiều càng tốt.

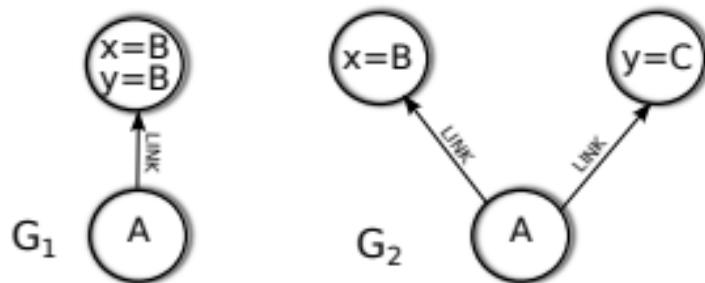
2.4 Query optimization:

2.4.1 General Architecture

Trình tối ưu hóa Cypher thực hiện tối ưu 1 quy vấn thông qua các bước sau:

- Biến câu truy vấn thành Cây cú pháp trừu tượng (Abstract Syntax Tree), kiểm tra ngữ nghĩa (ví dụ: tất cả các nhãn tồn tại), chuẩn hóa nó.
- Xây dựng biểu diễn đồ thị cho câu truy vấn. Cạnh trong đồ thị truy vấn không tương đương với các phép join mà nó có tác dụng mô tả quan hệ.

```
MATCH (A) - [:LINK] -> (x) ,  
      (A) - [:LINK] -> (y)  
RETURN x , y
```



Hình 14: Ví dụ về biểu diễn đồ thị cho câu truy vấn

- Xử lý các truy vấn con và sau đó tìm thứ tự tối ưu cho các toán tử (cho phép join hay các toán tử mở rộng...). Chi tiết cho phần này sẽ được mô tả ở phần sau (Phần 2.4.2)
- Biến kế hoạch thực thi tối ưu thành các phương thức API để truy cập vào dữ liệu gốc.

2.4.2 Operator Ordering

Ở phần này chúng ta sẽ xem xét cách xây dựng cho 1 kế hoạch hợp lý cho biểu đồ truy vấn theo cách từ dưới lên bằng thuật toán tham lam.

Algorithm 13: Greedy operator ordering for Cypher

```

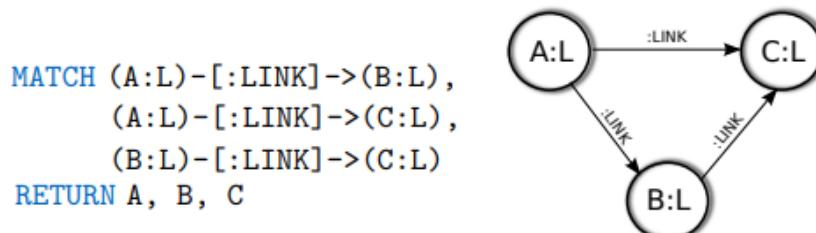
Input: Query graph  $Q$ 
Result: Logical plan  $P$  that covers all nodes from  $Q$ 

1  $\mathcal{P} \leftarrow []$  ▷ PlanTable
2 foreach  $n \in Q$  do ▷ every node in the query graph
3    $T \leftarrow \text{constructLeafPlan}(n)$  ▷ take selections into account
4    $\mathcal{P}.\text{insert}(T)$ 
5 do
6    $\text{Cand} \leftarrow []$  ▷ candidate solutions
7   foreach  $P_1 \in \mathcal{P}$  do
8     foreach  $P_2 \in \mathcal{P}$  do
9       if  $\text{CanJoin}(P_1, P_2)$  then
10          $T \leftarrow \text{constructJoin}(P_1, P_2)$ 
11          $\text{Cand}.\text{insert}(T)$ 
12   foreach  $P_1 \in \mathcal{P}$  do
13      $T \leftarrow \text{constructExpand}(P_1)$ 
14      $\text{Cand}.\text{insert}(T)$ 
15   if  $\text{Cand.size} \geq 1$  then
16      $T_{\text{best}} \leftarrow \text{pickBest}(\text{Cand})$  ▷ pick the plan with the smallest cost
17     foreach  $T \in \mathcal{P}$  do
18       if  $\text{covers}(T_{\text{best}}, T)$  then
19          $\mathcal{P}.\text{erase}(T)$  ▷ delete plans covered by  $T_{\text{best}}$ 
20      $T_{\text{best}} \leftarrow \text{applySelections}(T_{\text{best}})$ 
21      $\mathcal{P}.\text{insert}(T_{\text{best}})$ 
22 while  $\text{Cand.size} \geq 1$ 
23 return  $\mathcal{P}[0]$ 

```

Hình 15: Thuật toán tham lam cho việc xếp thứ tự toán tử trong Cypher

Dễ cho đơn giản thì ta sẽ đến ví dụ cho áp dụng thuật toán Greedy cho việc xếp thứ tự thực thi cho các toán tử:



Hình 16: Ví dụ minh họa cho thuật toán Greedy

Chúng ta sẽ tiến hành từng bước thuật toán cho truy vấn này như trong Hình 16, ta hiển thị PlanTable cùng với danh sách ứng cử viên và kế hoạch tốt nhất.

Ban đầu (Bước 1), bảng được khởi tạo với các kế hoạch cung cấp quyền truy cập tốt nhất vào các nút đơn lẻ. Trong trường hợp này, có ba lần quét nhãn $\sigma : L$ trích xuất các nút của nhãn L đã cho. Ở Bước 1, cách duy nhất để mở rộng ba kế hoạch này là hình thành các bản mở rộng khác nhau từ các nút đọc theo các cạnh của biểu đồ truy vấn. Có tổng cộng 6 khả năng mở rộng và dựa trên cách tính chi phí. Ta chọn T_{best} , loại bỏ tất cả các phương án từ P được bao phủ bởi T_{best} (cụ thể là các kế hoạch cho các nút A và B) và tiếp tục sang Bước 2. Một lần nữa, chỉ có thể mở rộng từ các kế hoạch hiện có, với ứng cử viên tốt nhất là $E_B \rightarrow C(\sigma : L(B))$. Kế hoạch này thay thế kế hoạch cho B, nhưng để lại kế hoạch cho A, C vì không bao gồm đầy đủ chúng.

Ở Bước 3, chúng ta lần đầu tiên có phép kết hai bài toán con, vì bây giờ có hai kế hoạch P và Q trong PlanTable với các bộ biến chồng chéo. Các kế hoạch kết quả cho ta kết quả tốt hơn bất kỳ ứng cử viên

nào khác và thay thế tất cả các kế hoạch một phần trong P. Tuy nhiên, ở bước tiếp theo, chúng ta phát hiện ra rằng có một kế hoạch nữa khả năng mở rộng từ kế hoạch này, cụ thể là cạnh $A \rightarrow B$, dẫn ta đến kế hoạch được mô tả trong P ở Bước 5. Toán tử Mở rộng cuối cùng sẽ tạo ra các liên kết B' và nó phải được theo sau bởi một lựa chọn đảm bảo $B = B'$, nhưng chúng ta bỏ qua lựa chọn này để cho biểu thức đơn giản.

Thuật toán tham lam được trình bày ở đây chỉ mang lại giá trị gần đúng cho phương án tối ưu. Công việc trong tương lai bao gồm việc khai thác Lập trình động - giống như cách thuật toán đơn giản hóa đồ thị truy vấn khi cần thiết.

Step 1:

$\{A\}$	$\sigma_{:L}(A)$
$\{B\}$	$\sigma_{:L}(B)$
$\{C\}$	$\sigma_{:L}(C)$

Cand: $A \rightarrow B \quad A \rightarrow C \quad B \rightarrow C$
 $B \leftarrow A \quad C \leftarrow A \quad C \leftarrow B$
 $T_{best}: A \rightarrow C$

Step 2:

$\{A, C\}$	$\mathcal{E}_{A \rightarrow C}(\sigma_{:L}(A))$
$\{B\}$	$\sigma_{:L}(B)$

Cand: $A \xrightarrow{\nearrow} C \quad A \rightarrow C \leftarrow B \quad B \rightarrow C$
 $A \xrightarrow{\searrow} B \quad C \leftarrow B$
 $T_{best}: B \rightarrow C$

Step 3:

$\{A, C\}$	$\mathcal{E}_{A \rightarrow C}(\sigma_{:L}(A))$
$\{B, C\}$	$\mathcal{E}_{B \rightarrow C}(\sigma_{:L}(B))$

Cand: $A \xrightarrow{\nearrow} C \quad B \xrightarrow{\nearrow} C \quad B \rightarrow C \leftarrow A$
 $A \xrightarrow{\searrow} B \quad (A \rightarrow C) \bowtie (B \rightarrow C)$
 $T_{best}: (A \rightarrow C) \bowtie (B \rightarrow C)$

Step 4:

$\{A, B, C\}$	$\mathcal{E}_{A \rightarrow C}(\sigma_{:L}(A)) \bowtie \mathcal{E}_{B \rightarrow C}(\sigma_{:L}(B))$
---------------	---

Cand: $((A \rightarrow C) \bowtie (B \rightarrow C)) \rightarrow B$

Step 5:

$\{A, B, C\}$	$\mathcal{E}_{A \rightarrow B}(\mathcal{E}_{A \rightarrow C}(\sigma_{:L}(A)) \bowtie \mathcal{E}_{B \rightarrow C}(\sigma_{:L}(B)))$
---------------	--

Figure 5.4.: Greedy Operator ordering for Cypher

Hình 17: Áp dụng thuật toán Greedy cho ví dụ minh họa

2.4.3 Profile a query

Có hai tùy chọn khi muốn phân tích câu truy vấn Neo4j bằng cách xem xét kế hoạch thực thi:

EXPLAIN

Nếu muốn xem kế hoạch thực thi nhưng không chạy câu lệnh, thì dùng lệnh EXPLAIN. Câu lệnh sẽ luôn trả về kết quả rỗng và không làm thay đổi cơ sở dữ liệu.

PROFILE

Nếu muốn chạy câu lệnh và xem toán tử nào đang thực hiện hầu hết công việc truy vấn, thì sử dụng PROFILE. Việc sử dụng Profile sẽ chạy câu lệnh xuống dữ liệu và ta theo dõi được có bao nhiêu hàng thông qua mỗi toán tử và mỗi toán tử cần tương tác bao nhiêu với lớp lưu trữ để truy xuất dữ liệu cần thiết. Việc lập hồ sơ cho truy vấn sử dụng nhiều tài nguyên hơn, vì vậy ta không nên lập hồ sơ trừ khi đang làm việc trực tiếp với một truy vấn xuống dữ liệu gốc.

2.4.4 Example

Các **dữ liệu** trong ví dụ minh họa này nằm trong mục Nguồn dữ liệu trong **phần 4 (Application)**

2.4.4.a Example 1

Chúng ta sẽ tiến hành tìm node có tên là "Terry Notary" trong cơ sở dữ liệu:

```
MATCH (p {name: 'Terry Notary'})  
RETURN p
```

Kết quả trả về:

```
"p"  
{ "name": "Terry Notary" }
```

Hình 18: Kết quả cho câu truy vấn tìm "Terry Notary"

Ta sẽ xem kế hoạch thực thi câu lệnh đó bằng lệnh Profile:

```
PROFILE  
MATCH (p {name: 'Terry Notary'})  
RETURN p
```



Kết quả trả về:



Hình 19: Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary"

Nhìn kết quả thực thi câu truy vấn trên thì ta thấy được là AllNodesScan đã được sử dụng, nó chiếm phần lớn lượng thời gian trong thực thi câu truy vấn, công cụ lập kế hoạch truy vấn đã quét qua tất cả các nút trong cơ sở dữ liệu.

Toán tử Bộ lọc sẽ kiểm tra thuộc tính tên trên mỗi nút được AllNodesScan truyền qua.

Đây có vẻ là một cách làm không hiệu quả để tìm 'Terry Notary' vì ta đang xem xét quá nhiều nút thậm chí không phải là người và do đó không phải là thứ mà ta đang tìm kiếm.

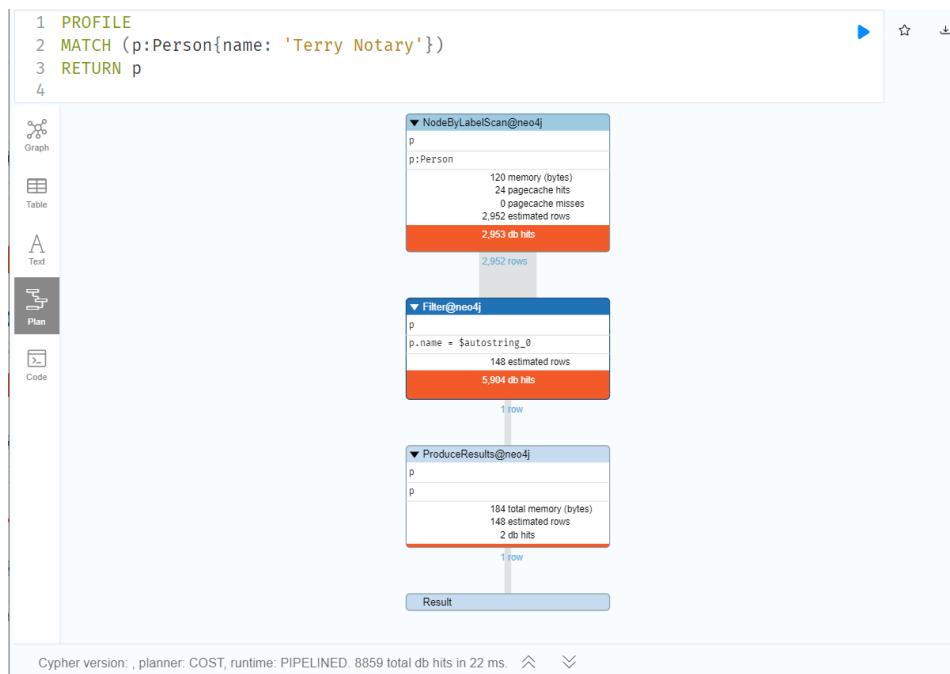
Giải pháp cho vấn đề này là bắt cứ khi nào ta tìm kiếm một nút, ta nên chỉ định một nhãn để giúp trình lập kế hoạch truy vấn thu hẹp không gian tìm kiếm.

Ta xét đối với truy vấn này, ta cần thêm nhãn Người:

```
PROFILE  
MATCH (p:Person {name: 'Terry Notary'})  
RETURN p
```



Kết quả trả về:



Hình 20: Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary" có chỉ định nhãn là People

Truy vấn này sẽ nhanh hơn truy vấn đầu tiên, nhưng khi số người trong cơ sở dữ liệu tăng lên thì truy vấn có thể sẽ chậm lại đáng kể.

Lần này thì giá trị của hàng cuối cùng đã giảm đi đáng kể, ta không quét hết tất cả các nút như câu truy vấn trước đây. Toán tử NodeByLabelScan cho ta biết rằng ta đã đạt được điều này bằng cách quét tuyển tính tất cả các nút Person trong cơ sở dữ liệu (không quét hết tất cả các nút như trước). Sau đó so sánh thuộc tính tên từng nút của nút có nhãn là Person.

Điều này có thể chấp nhận được trong một số trường hợp nhưng nếu ta tra cứu mọi người theo tên 1 cách thường xuyên thì ta sẽ thấy hiệu suất tốt hơn nếu tạo một chỉ mục trên thuộc tính tên cho nhãn Person:

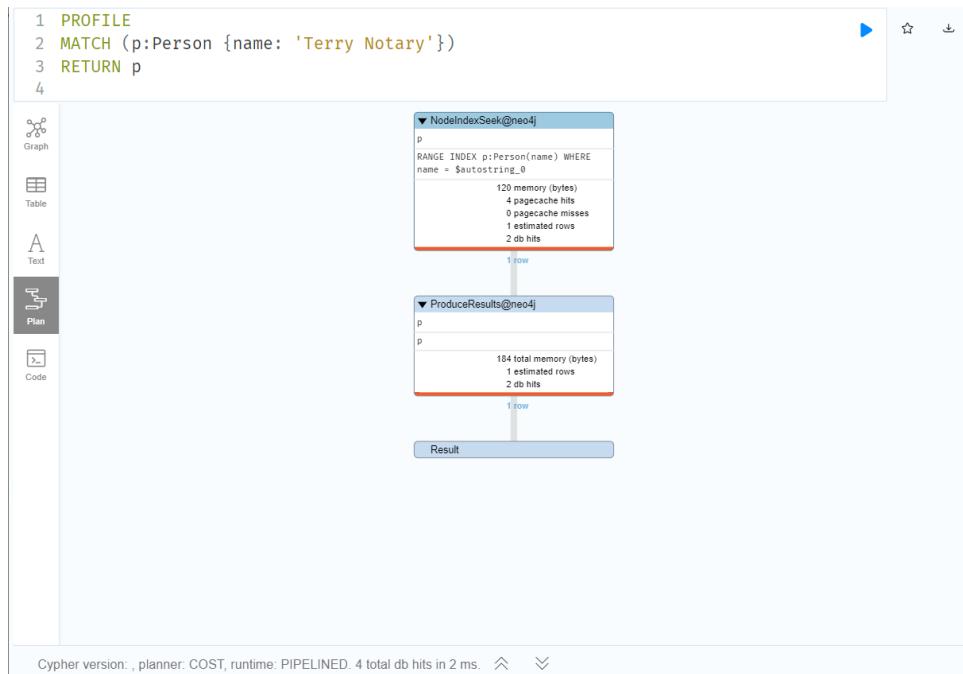
```
CREATE INDEX FOR (p:Person)
ON (p.name)
```

Sau khi tạo Index, ta chạy lại lệnh trên để kiểm tra kết quả:

```
PROFILE
MATCH (p:Person {name: 'Terry Notary'})
RETURN p
```



Kết quả trả về:



Hình 21: Kế hoạch thực thi cho câu truy vấn tìm "Terry Notary" có index và chỉ định nhãn là People

Kết quả đạt được tốt hơn đáng kể so với các câu truy vấn trước đây, tổng số db hits chỉ là 4, tốt hơn nhiều so với 2 câu truy vấn trước (16299 và 8859 db hits)

2.4.4.b Example 2

Chúng ta sẽ tiến hành tìm người có tên bắt đầu là "Terry" và đếm số lượng bộ phim mà người đó đã đóng.

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Terry'
RETURN
  p.name AS name,
  count(m) AS count
```



Kết quả trả về:

name	count
"Terry Chen"	7
"Terry Crews"	19
"Terry Kinney"	6
"Terry Notary"	7
"Terry O'Quinn"	5
"Terry Serpico"	6

Started streaming 6 records after 3 ms and completed after 5 ms.

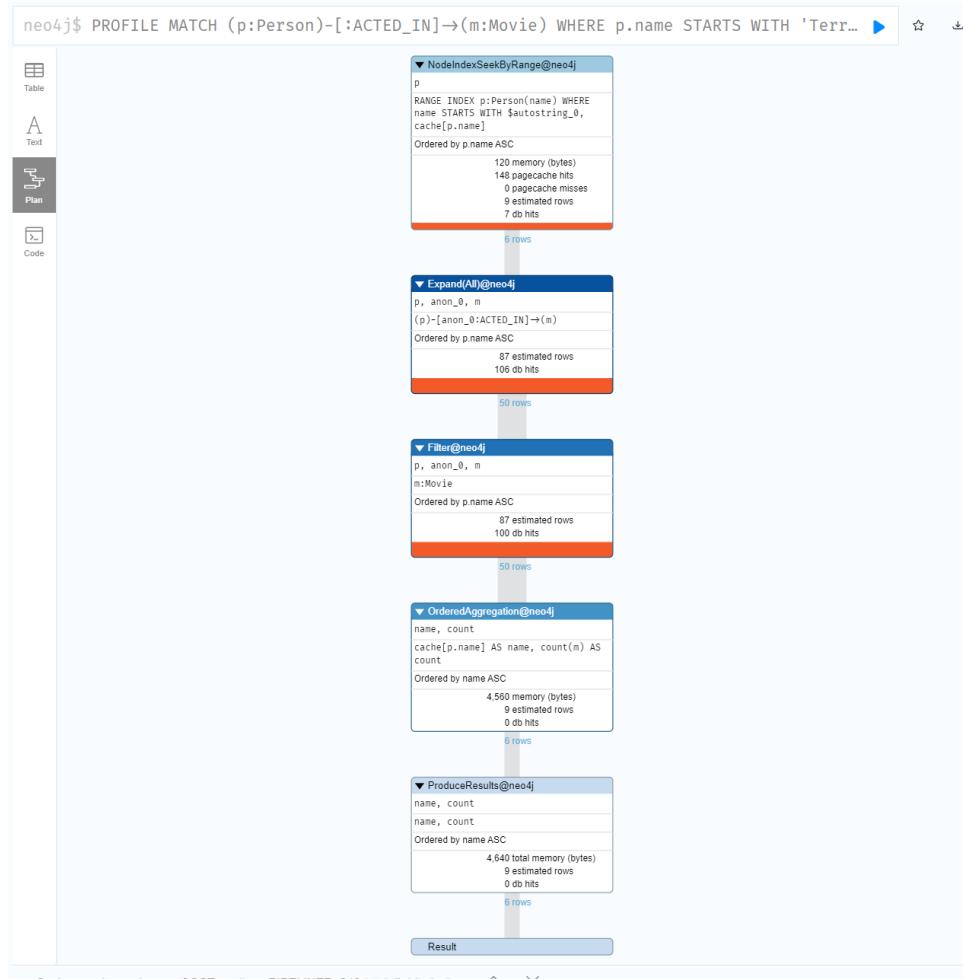
Hình 22: Kết quả cho câu truy vấn người có tên bắt đầu là "Terry"

Truy vấn yêu cầu cơ sở dữ liệu trả về tất cả các diễn viên có tên bắt đầu là 'Terry'. Với chỉ mục gốc, ta có thể tận dụng các chỉ mục lưu trữ các giá trị thuộc tính. Điều này có nghĩa là tên có thể được tra cứu trực tiếp từ chỉ mục. Nếu lập hồ sơ cho câu truy vấn trên, chúng ta sẽ thấy rằng NodeIndexSeekByRange trong cột Chi tiết chứa bộ đệm[p.name], có nghĩa là p.name được truy xuất từ chỉ mục. Chúng ta cũng có thể thấy rằng OrderedAggregation không có Lượt truy cập DB, nghĩa là nó không phải truy cập lại cơ sở dữ liệu.

```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Terry'
RETURN
    p.name AS name,
    count(m) AS count
```



Kết quả trả về:



Hình 23: Kế hoạch thực thi cho câu truy vấn người có tên bắt đầu là "Terry"

2.4.4.c Example 3

Aggregating functions: Chúng ta sẽ tiến hành đếm số lượng tên người riêng biệt trong cơ sở dữ liệu.

```
PROFILE
MATCH (p:Person)
RETURN count(DISTINCT p.name) AS numberOfNames
```

Kết quả trả về:



Hình 24: Kế hoạch thực thi cho câu truy vấn số lượng tên người riêng biệt trong CSDL

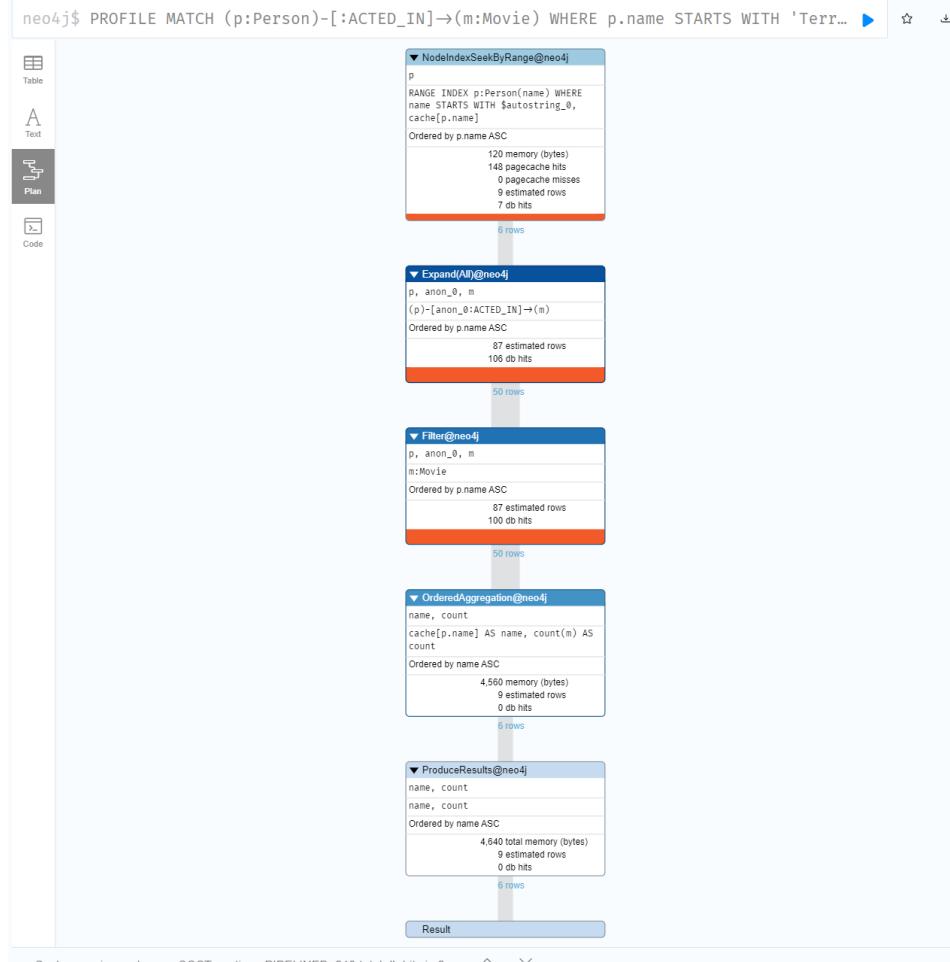
NodeIndexScan trong cột Chi tiết chứa bộ nhớ cache[p.name] và EagerAggregation không có Lần truy cập DB nào. Trong trường hợp này, ngữ nghĩa của hàm tổng hợp hoạt động giống như một phép vị từ tồn tại (predicate) ngầm định.

2.4.4.d Example 4

Chúng ta sẽ tiến hành tìm người có tên bắt đầu là "Terry" và đếm số lượng bộ phim mà người đó đã đóng và xếp thứ tự hiển thị tăng dần theo tên.

```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Terry'
RETURN
    p.name AS name,
    count(m) AS count
ORDER BY name
```

Kết quả trả về:



Hình 25: Kế hoạch thực thi cho câu truy vấn người có tên bắt đầu là "Terry" và xếp thứ tự theo tên

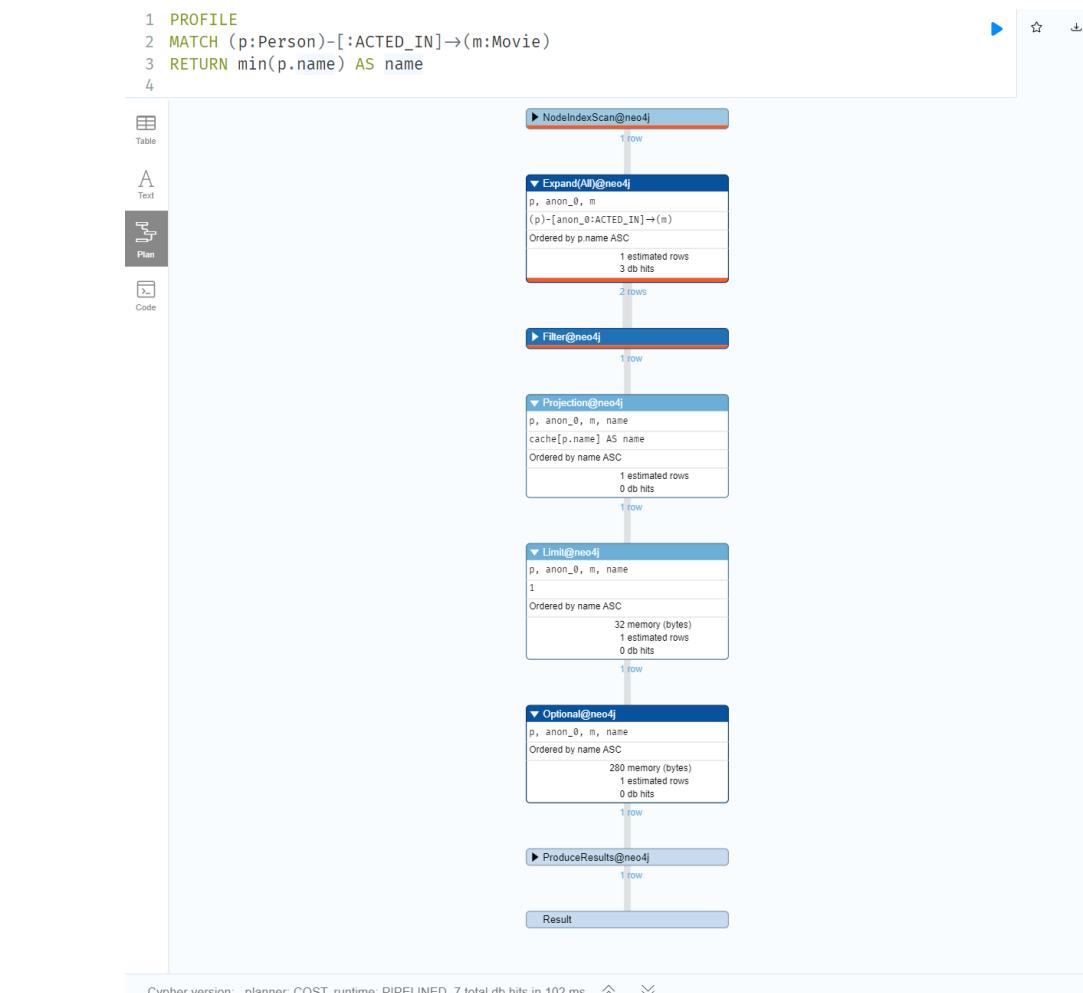
Công cụ lập kế hoạch Cypher nhận ra rằng chỉ mục đã trả về dữ liệu theo đúng thứ tự và bỏ qua thao tác Sắp xếp.

2.4.4.e Example 5

Hàm Min và Max: Đối với các hàm Min và Max, chúng có thể sử dụng tối ưu hóa ORDER BY được hỗ trợ bởi chỉ mục để tránh tổng hợp và thay vào đó là tận dụng thực tế là giá trị tối thiểu/tối đa là giá trị đầu tiên/cuối cùng trong một chỉ mục được sắp xếp. Ta xem xét câu truy vấn sau đây trả về diễn viên đầu tiên xuất hiện theo thứ tự bảng chữ cái:

```
PROFILE
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
RETURN min(p.name) AS name
```

Kết quả trả về:



Hình 26: Kế hoạch thực thi cho câu truy vấn diễn viên đầu tiên xuất hiện theo thứ tự bảng chữ cái

Câu truy vấn được lập kế hoạch với Phép chiếu, tiếp theo là Giới hạn (limit), tiếp theo là Tùy chọn(Optional). Điều này sẽ chỉ chọn giá trị đầu tiên từ chỉ mục. Đối với tập dữ liệu lớn thì điều này có thể cải thiện đáng kể hiệu suất. ORDER BY được hỗ trợ bởi chỉ mục cũng có thể được sử dụng cho các truy vấn tương ứng với hàm max, nhưng với hiệu suất thấp hơn một chút.

Hạn chế

Việc tối ưu hóa chỉ có thể hoạt động trên các chỉ mục gốc. Nó không hoạt động đối với các vị từ chỉ truy vấn cho loại không gian Điểm (spatial type Point).

Các vị từ có thể được sử dụng để kích hoạt tối ưu hóa này là:

- Existence (Ví dụ: WHERE n.name IS NOT NULL)
- Equality (Ví dụ: WHERE n.name = 'Tom Hanks')
- Range (Ví dụ: WHERE n.uid > 1000 AND n.uid < 2000)
- Prefix (Ví dụ: WHERE n.name STARTS WITH 'Tom')
- Suffix (Ví dụ: WHERE n.name ENDS WITH 'Hanks')
- Substring (Ví dụ: WHERE n.name CONTAINS 'a')

Các vị từ sẽ không hoạt động đối với các trường hợp như:

- Một số vị từ kết hợp sử dụng OR.



- Dảng thức hoặc phạm vi vị từ cho truy vấn điểm (ví dụ: WHERE n.place > point(x: 1, y: 2))
- Vị từ khoảng cách không gian (ví dụ: WHERE point.distance(n.place, point(x: 1, y: 2)) < 2)

3 Comparison between PostgreSQL and Neo4j

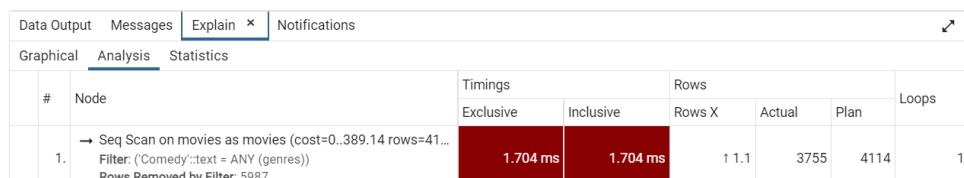
	Neo4j	PostgreSQL
Hiệu suất	Truy vấn dữ liệu với thường nhanh hơn so với PostgreSQL, đặc biệt là đối với các tác vụ liên quan đến phân tích mạng lưới hoặc các quan hệ phức tạp giữa các đối tượng.	PostgreSQL có thể có hiệu suất tốt hơn đối với các tác vụ liên quan đến tính toán số liệu và các truy vấn đơn giản, nhưng đối với các truy vấn phức tạp, hiệu suất có thể chậm hơn so với Neo4j.
Mô hình dữ liệu	Neo4j là một hệ thống cơ sở dữ liệu đồ thị, cho phép lưu trữ dữ liệu dưới dạng các nút và các mối quan hệ giữa chúng.	PostgreSQL là một hệ thống cơ sở dữ liệu quan hệ, sử dụng bảng để lưu trữ dữ liệu và quan hệ giữa các bảng để thực hiện các truy vấn phức tạp.
Khả năng mở rộng	Hỗ trợ mở rộng dữ liệu để đáp ứng nhu cầu của các ứng dụng ngày càng lớn. Tuy nhiên, với các ứng dụng đòi hỏi tính linh hoạt cao và khả năng mở rộng theo hướng dọc, Neo4j có thể hiệu quả hơn so với PostgreSQL.	Hỗ trợ mở rộng dữ liệu để đáp ứng nhu cầu của các ứng dụng ngày càng lớn.
Khả năng tích hợp:	Hỗ trợ các công nghệ tích hợp phổ biến, chẳng hạn như Java, Python, Ruby, Node.js, và nhiều hơn nữa. Tuy nhiên, với các ứng dụng yêu cầu tích hợp dữ liệu và các ứng dụng web, Neo4j có thể hiệu quả hơn so với PostgreSQL.	Hỗ trợ các công nghệ tích hợp phổ biến, chẳng hạn như Java, Python, Ruby, Node.js, và nhiều hơn nữa.
Chi phí	Đều là các sản phẩm mã nguồn mở miễn phí để sử dụng, Nhưng chi phí để lưu trữ dữ liệu cao hơn PostgreSQL	Đều là các sản phẩm mã nguồn mở miễn phí để sử dụng

So sánh thời gian thực hiện của một số câu query:

- Chọn các phim có thể loại là comedy
- PostgreSQL:

```
SELECT * FROM movies WHERE 'Comedy' = ANY(genres);
```

Kết quả:



Hình 27: Kết quả thực thi movie thể loại comedy ở PostgreSQL

Với câu lệnh này: PostgreSQL thực thi bằng cách tìm kiếm tuần tự trên bảng movies và tổng thời gian thực thi là 1.704ms.

Neo4j:

```
MATCH (m:Movie)-[:IS_GENRE]->(g:Genre) WHERE g.name CONTAINS "Comedy" RETURN
```

Kết quả:



The screenshot shows the Neo4j browser interface with a query result. The query is: `neo4j$ MATCH (m:Movie)-[:IS_GENRE]->(g:Genre) WHERE g.name CONTAINS "Comedy" RETURN m`. The results are displayed in a table with two rows. Row 1 represents a movie node with properties: identity=5316, labels=[Movie], properties={tmdbId=11796, id=8270, title='Hairdresser's Husband, The (Le mari de la coiffeuse) (1990)'}, elementId=5316. Row 2 represents another movie node with properties: identity=3258, labels=[Movie]. A note at the bottom says "Started streaming 3756 records after 1 ms and completed after 4 ms, displaying first 1000 rows."

Hình 28: Kết quả thực thi movie thẻ loại comedy ở Neo4j

Với câu lệnh này: Neo4j thực thi với tổng thời gian là 4ms

Có thể thấy PostgreSQL cho kết quả thực thi ở câu lệnh này tốt hơn Neo4j (1.704ms so với 4ms).

2. Chọn movies có thẻ loại là comedy và năm xuất bản là 1993

PostgreSQL:

```
SELECT * FROM movies WHERE 'Comedy' = ANY(genres)
AND title like '%1993%';
```

Kết quả:

The screenshot shows the PostgreSQL Explain plan for the query. It includes the Data Output, Messages, Explain, and Notifications tabs. The Explain tab shows the execution plan: 1. Seq Scan on movies as movies (cost=0..415.82 rows=83 width=1425). Filter: ((title)::text ~~~ '%1993%'::text) AND ('Comedy'::text = ANY (genres)). Rows Removed by Filter: 9665. The plan is highlighted in red. The Explain output table has columns: # (1), Node (Seq Scan on movies as movies), Timings (Exclusive: 1.425 ms, Inclusive: 1.425 ms), Rows (Rows X: 1.08, Actual: 77, Plan: 83, Loops: 1).

Hình 29: Kết quả thực thi movie thẻ loại comedy và năm xuất bản 1993 ở PostgreSQL

Với câu lệnh này: PostgreSQL thực thi bằng cách tìm kiếm tuần tự trên bảng movies và tổng thời gian thực thi là 1.425ms. Neo4j:

```
MATCH (m:Movie)-[:IS_GENRE]->(g:Genre) WHERE g.name CONTAINS "Comedy" AND m.title CONTAINS "1993" RETURN m
```

Kết quả:

The screenshot shows the Neo4j browser interface with a query result. The query is: `neo4j$ MATCH (m:Movie)-[:IS_GENRE]->(g:Genre) WHERE g.name CONTAINS "Comedy" AND m.title CONTAINS "1993" RETURN m`. The results are displayed in a table with two rows. Row 1 represents a movie node with properties: identity=605, labels=[Movie], properties={tmdbId=788, id=500, title='Mrs. Doubtfire (1993)'}, elementId=605. Row 2 represents another movie node with properties: identity=5754, labels=[Movie]. A note at the bottom says "Started streaming 77 records after 1 ms and completed after 10 ms."

Hình 30: Kết quả thực thi movie thẻ loại comedy và năm xuất bản 1993 ở Neo4j

Với câu lệnh này: Neo4j thực thi với tổng thời gian là 10ms
Có thể thấy PostgreSQL cho kết quả thực thi ở câu lệnh này tốt hơn Neo4j (1.425 so với 10ms).

3. Chọn diễn viên trong các phim mà họ đóng có phim có trung bình đánh giá lớn hơn 4.

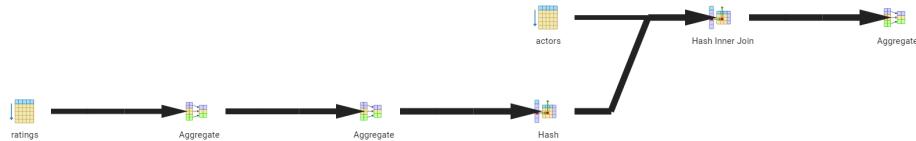
PostgreSQL:

```
SELECT DISTINCT person_name FROM actors
  WHERE actors.movieid IN (SELECT distinct ratings.movieid
                            FROM ratings GROUP BY movieid
                            HAVING AVG(rating) > 4);
```

Kết quả:

#	Node	Timings		Rows		Loops
		Exclusive	Inclusive	Rows X	Actual	
1.	→ Aggregate (cost=3330.23 .. 3359.75 rows=2952 width=14) (actual=54.644, 54.7...	0.446 ms	54.705 ms	1 4.88	605	2952
	Buckets: Batches: Memory Usage: 177 kB					
2.	→ Hash Inner Join (cost=2598.6 .. 3288.24 rows=16793 width=14) (actual=46...	5.386 ms	54.259 ms	1 23.82	705	16793
	Hash Cond: ((actors.movieid)=text + (ratings.movieid)=text)					
3.	→ Seq Scan on actors as actors (cost=0..601.7 rows=33470 width=19) (...	2.705 ms	2.705 ms	1 1	33470	33470
4.	→ Hash (cost=2571.56 .. 2571.56 rows=2163 width=5) (actual=46.167, 4...	0.536 ms	46.169 ms	1 1.72	1260	2163
	Buckets: 4096 Batches: 1 Memory Usage: 79 kB					
5.	→ Aggregate (cost=2528.3 .. 2549.93 rows=2163 width=5) (actual=4...	0.935 ms	45.633 ms	1 1.72	1260	2163
	Buckets: Batches: Memory Usage: 177 kB					
6.	→ Aggregate (cost=2425.54 .. 2522.89 rows=2163 width=5) (actu...	37.808 ms	44.699 ms	1 1.72	1260	2163
	filter: (avg(rating) > 4 - double precision)					
	Rows Removed by Filter: 8464					
	Buckets: Memory Usage: 1937 kB					
7.	→ Seq Scan on ratings as ratings (cost=0..1921.36 rows=...	6.891 ms	6.891 ms	1 1	100836	100836

Hình 31: Kết quả thực thi chọn diễn viên trong các phim mà họ đóng có phim có trung bình đánh giá lớn hơn 4 ở PostgreSQL



Hình 32: Quá trình thực thi câu lệnh

Với câu lệnh này: PostgreSQL thực thi bằng cách:

- Thực hiện scan tuần tự trên bảng ratings, với tổng thời gian là 6.891ms.
- Sau đó thực hiện hàm aggregation để lọc ra những phim có đánh giá trên 4, với thời gian là 37.808ms.
- Sau đó tiếp tục thực hiện aggregation trước khi băm, với thời gian thực thi là 0.935ms.
- Sau đó thực hiện băm với thời gian thực thi là 0.536ms.
- Thực hiện scan tuần tự trên bảng actors với thời gian thực thi là 2.705ms.
- Thực hiện kết băm nội (Hash Inner Join) với trường băm là movieid trên cả 2 bảng actors và ratings.
- Sau đó thực hiện aggregation với thời gian thực thi là 0.446ms.

Tổng thời gian thực thi cho toàn bộ câu lệnh là 94ms.

Neo4j:

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)<--[r:RATED]-(u:User)
WITH m, avg(r.rating) AS avg_rating
WHERE avg_rating > 4
MATCH (p:Person)-[:ACTED_IN]->(m)
RETURN DISTINCT p
```

Kết quả:

```

p
{
  "identity": 12003,
  "labels": [
    "Person"
  ],
  "properties": {
    "name": "Udo Kier"
  },
  "elementId": "12003"
}

{
  "identity": 1,
  "labels": [
    "Person"
  ],
}

```

Started streaming 1130 records after 3 ms and completed after 4 ms, displaying first 1000 rows.

Hình 33: Kết quả thực thi chọn diễn viên trong các phim mà họ đóng có phim có trung bình đánh giá lớn hơn 4 ở Neo4j

Với câu lệnh này: Neo4j thực thi với tổng thời gian là 4ms

Có thể thấy Neo4j cho kết quả thực thi ở câu lệnh này tốt hơn PostgreSQL (4ms so với 94ms).

4. Chọn ra những người vừa làm diễn viên vừa làm đạo diễn PostgreSQL:

```

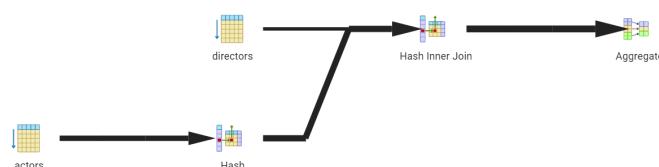
SELECT DISTINCT actors.person_name FROM actors JOIN directors ON actors.
person_name = directors.person_name

```

Kết quả:

#	Node	Timings		Rows	Plan	Loops
		Exclusive	Inclusive			
1.	→ Aggregate (cost=1508.45..1537.97 rows=2952 width=14) (actual=12.705..12...	0.569 ms	12.72 ms	1 98.4	30	2952
2.	→ Hash Inner Join (cost=1020.08..1451.96 rows=22595 width=14) (actual=...	1.003 ms	12.152 ms	1 7.35	3075	22595
3.	→ Seq Scan on directors as directors (cost=0..33.58 rows=2058 width=...	0.144 ms	0.144 ms	1 1	2058	2058
4.	→ Hash (cost=601.7..601.7 rows=33470 width=14) (actual=11.004..11...	7.086 ms	11.005 ms	1 1	33470	33470
5.	→ Seq Scan on actors as actors (cost=0..601.7 rows=33470 width=...	3.919 ms	3.919 ms	1 1	33470	33470

Hình 34: Kết quả thực thi những người vừa làm diễn viên vừa làm đạo diễn ở PostgreSQL



Hình 35: Quá trình thực thi câu lệnh

Với câu lệnh này: PostgreSQL thực thi bằng cách:

- Thực hiện scan tuần tự trên bảng actor, với tổng thời gian thực thi là 3.919ms.
- Thực hiện băm trên kết quả vừa thực thi, với tổng thời gian thực thi là 7.086ms.
- Thực hiện scan tuần tự trên bảng directors, với tổng thời gian thực thi là 0.144ms.
- Sau đó thực hiện kết băm nội (Hash Inner Join), với tổng thời gian thực thi là 1.003ms.
- Sau đó thực hiện chỉ lấy các giá trị giống nhau trên trường tên 1 lần, với tổng thời gian là 0.569ms.



Tổng thời gian thực thi câu lệnh là 49ms.

Neo4j:

```
MATCH (d:Person)-[:DIRECTED]->(m:Movie)<-[:ACTED_IN]-(a:Person)
WHERE d.name = a.name
RETURN DISTINCT d
```

Kết quả:

```
d
1
{
  "identity": 71,
  "labels": [
    "Person"
  ],
  "properties": {
    "born": 1956,
    "name": "Tom Hanks"
  },
  "elementId": "71"
}
2
{
  "identity": 99,
  "labels": [
    "Person"
  ]
}
```

Started streaming 30 records after 14 ms and completed after 80 ms.

Hình 36: Kết quả thực thi những người vừa làm diễn viên vừa làm đạo diễn ở Neo4j

Với câu lệnh này: Neo4j thực thi với tổng thời gian là 80ms

Có thể thấy PostgreSQL cho kết quả thực thi ở câu lệnh này tốt hơn Neo4j (49ms so với 80ms).

4 Application

4.1 Nguồn dữ liệu

- Bộ dữ liệu MovieLens (bản small), theo mô tả của chính nó, mô tả xếp hạng 5 sao và hoạt động gắn thẻ văn bản tự do từ MovieLens (<https://grouplens.org/datasets/movielens>), một dịch vụ đề xuất phim. Nó chứa 100836 đánh giá và 3683 nhãn trên 9742 phim. Những dữ liệu này được tạo bởi 610 người dùng trong khoảng thời gian từ ngày 29 tháng 3 năm 1996 đến ngày 24 tháng 9 năm 2018. Tập dữ liệu này được tạo vào ngày 26 tháng 9 năm 2018.
- Bộ dữ liệu phim TMDB 5000. Bộ dữ liệu này được tạo từ Movie Database API. Nó chứa 2 tệp csv: một tệp có thông tin chi tiết về phim (ngân sách, thể loại, ngôn ngữ gốc, v.v.), tệp thứ hai – chứa phần dữ liệu của phim về diễn viên, đạo diễn, nhà sản xuất. Nguồn: <https://www.kaggle.com/tmdb/tmdb-movie-metadata>

4.2 Mô tả dữ liệu

Tập Links.csv chứa 3 id khác nhau của mỗi phim: movieId – id được sử dụng trong tập dữ liệu MovieLens, imbdId – tương ứng với tập dữ liệu IMDB và tmdbId – id tương ứng với tập dữ liệu tmdb (<https://www.themoviedb.org/>) . Chúng ta sẽ sử dụng chúng để lấy thông tin về diễn viên và đạo diễn của phim. VD: phim Toy Story có link là <https://www.themoviedb.org/movie/862>.

movieId	imdbId	tmdbId
1	114709	862
2	113497	8844
3	113228	15602
4	114885	31357
5	113041	11862
6	113277	949
7	114319	11860
8	112302	45325
9	114576	9091
10	113189	710
11	112346	9087
12	112896	12110
13	112453	21032
14	113987	10858
15	112760	1408
16	112641	524
17	114388	4584
18	113101	5
19	112281	9273

Hình 37: File links.csv

Tập movies.csv có thông tin về id phim, tiêu đề cùng với năm phát hành và thể loại (được phân tách bằng dấu "|"):



movielid	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action Crime Thriller
7	Sabrina (1995)	Comedy Romance
8	Tom and Huck (1995)	Adventure Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action Adventure Thriller
11	American President, The (1995)	Comedy Drama Romance
12	Dracula: Dead and Loving It (1995)	Comedy Horror
13	Balto (1995)	Adventure Animation Children
14	Nixon (1995)	Drama
15	Cutthroat Island (1995)	Action Adventure Romance
16	Casino (1995)	Crime Drama
17	Sense and Sensibility (1995)	Drama Romance
18	Four Rooms (1995)	Comedy
19	Ace Ventura: When Nature Calls (1995)	Comedy
20	Money Train (1995)	Action Comedy Crime Drama Thriller
21	Get Shorty (1995)	Comedy Crime Thriller

Hình 38: File movies.csv

Tệp ratings.csv chứa điểm đánh giá được thực hiện trên thang điểm 5 sao (0,5 sao - 5,0 sao) của một bộ phim bởi một người dùng. Người dùng chỉ được đại diện bởi id. Id người dùng đã được ẩn danh.

userId	movielid	rating	timestamp
1	1	4	964982703
1	3	4	964981247
1	6	4	964982224
1	47	5	964983815
1	50	5	964982931
1	70	3	964982400
1	101	5	964980868
1	110	4	964982176
1	151	5	964984041
1	157	5	964984100
1	163	5	964983650
1	216	5	964981208
1	223	3	964980985
1	231	5	964981179
1	235	4	964980908
1	260	5	964981680
1	296	3	964982967
1	316	3	964982310
1	333	5	964981179

Hình 39: File ratings.csv

Mỗi dòng trong tệp tags.csv là nhãn mà 1 người gắn cho 1 bộ phim.



userId	movielid	tag	timestamp
2	60756	funny	1445714994
2	60756	Highly quotable	1445714996
2	60756	will ferrell	1445714992
2	89774	Boxing story	1445715207
2	89774	MMA	1445715200
2	89774	Tom Hardy	1445715205
2	106782	drugs	1445715054
2	106782	Leonardo DiCaprio	1445715051
2	106782	Martin Scorsese	1445715056
7	48516	way too long	1169687325
18	431	Al Pacino	1462138765
18	431	gangster	1462138749
18	431	mafia	1462138755
18	1221	Al Pacino	1461699306
18	1221	Mafia	1461699303
18	5995	holocaust	1455735472
18	5995	true story	1455735479
18	44665	twist ending	1456948283
18	52604	Anthony Hopkins	1457650696
18	52604	courtroom drama	1457650711
18	52604	twist ending	1457650682

Hình 40: File tags.csv

Mỗi hàng của tệp tmdb_5000_credits.csv chứa thông tin movie_id, tiêu đề, dàn diễn viên và đoàn làm phim.

movie_id	title	cast	crew
19995	Avatar	[{"cast_id": 242, "char": [{"credit_id": "52fe48009251416c75"}]}	
285	Pirates of the Caribbean: On Stranger Tides	[{"cast_id": 4, "char": [{"credit_id": "52fe4232c3a36847f8"}]}	
206647	Spectre	[{"cast_id": 1, "char": [{"credit_id": "54805967c3a36829b"}]}	
49026	The Dark Knight Rises	[{"cast_id": 2, "char": [{"credit_id": "52fe4781c3a36847f8"}]}	
49529	John Carter	[{"cast_id": 5, "char": [{"credit_id": "52fe479ac3a36847f8"}]}	
559	Spider-Man 3	[{"cast_id": 30, "char": [{"credit_id": "52fe4252c3a36847f8"}]}	
38757	Tangled	[{"cast_id": 34, "char": [{"credit_id": "52fe46db9251416c91"}]}	
99861	Avengers: Age of Ultron	[{"cast_id": 76, "char": [{"credit_id": "55d5f7d4c3a3683e7e"}]}	
767	Harry Potter and the Half-Blood Prince	[{"cast_id": 3, "char": [{"credit_id": "52fe4273c3a36847f8"}]}	
209112	Batman v Superman: Dawn of Justice	[{"cast_id": 18, "char": [{"credit_id": "553bf23692514135c8"}]}	
1452	Superman Returns	[{"cast_id": 3, "char": [{"credit_id": "553bef6a9251416874"}]}	
10764	Quantum of Solace	[{"cast_id": 1, "char": [{"credit_id": "52fe43b29251416c75"}]}	
58	Pirates of the Caribbean: On Stranger Tides	[{"cast_id": 37, "char": [{"credit_id": "52fe4211c3a36847f8"}]}	
57201	The Lone Ranger	[{"cast_id": 4, "char": [{"credit_id": "52fe4928c3a36847f8"}]}	
49521	Man of Steel	[{"cast_id": 2, "char": [{"credit_id": "52fe4799c3a36847f8"}]}	
2454	The Chronicles of Narnia: Prince Caspian	[{"cast_id": 1, "char": [{"credit_id": "55a239e6925141297"}]}	
24428	The Avengers	[{"cast_id": 46, "char": [{"credit_id": "52fe4495c3a368484e"}]}	
1865	Pirates of the Caribbean: On Stranger Tides	[{"cast_id": 15, "char": [{"credit_id": "566b4f54c3a3683f56"}]}	
41154	Men in Black 3	[{"cast_id": 4, "char": [{"credit_id": "52fe45b7c3a36847f8"}]}	
122917	The Hobbit: The Desolation of Smaug	[{"cast_id": 10, "char": [{"credit_id": "548ad49a9251414fa2"}]}	
1930	The Amazing Spider-Man	[{"cast_id": 56, "char": [{"credit_id": "5395a60dc3a368641"}]}	
20662	Robin Hood	[{"cast_id": 1, "char": [{"credit_id": "52fe43f2c3a368484e"}]}	
57158	The Hobbit: The Desolation of Smaug	[{"cast_id": 3, "char": [{"credit_id": "52fe4926c3a36847f8"}]}	
2268	The Golden Compass	[{"cast_id": 43, "char": [{"credit_id": "52fe4348c3a36847f8"}]}	
254	King Kong	[{"cast_id": 5, "char": [{"credit_id": "52fe422ec3a36847f8"}]}	

Hình 41: File tmdb_5000_credits.csv

Chúng ta sẽ chỉ sử dụng tên diễn viên từ cột diễn viên và đạo diễn từ cột đoàn làm phim. Mỗi giá trị của cột “cast” chứa thông tin dữ liệu ở dạng JSON như sau:

```
[
  {
    "cast_id": 4,
    "character": "Captain Jack Sparrow",
    "credit_id": "52fe4232c3a36847f800b50d",
    "gender": 2,
    "id": 85,
    "name": "Johnny Depp",
    "order": 0
  },
  {
    "cast_id": 5,
    "character": "Will Turner",
    "credit_id": "52fe4232c3a36847f800b511",
    "gender": 2,
    "id": 114,
    "name": "Orlando Bloom",
    "order": 1
  },
  ...
]
```

Cột crew có dạng:

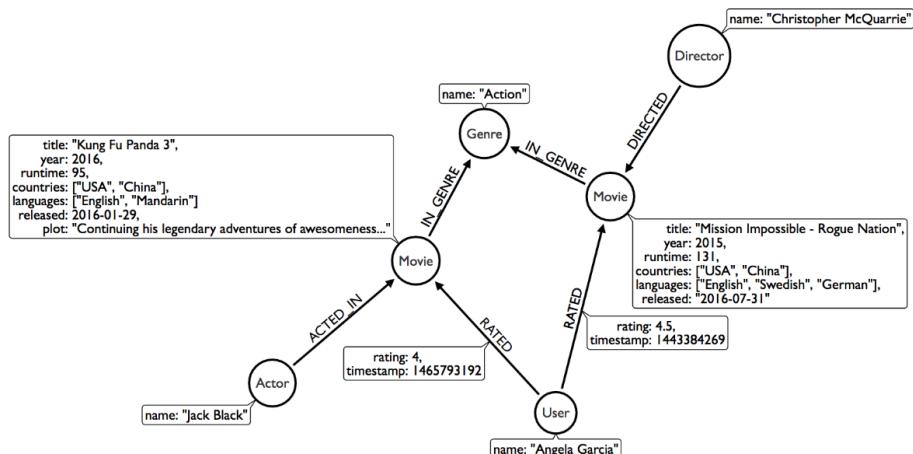
```
[
```

```
{
    "credit_id": "52fe4273c3a36847f801fa8d",
    "department": "Writing",
    "gender": 1,
    "id": 10966,
    "job": "Novel",
    "name": "J.K. Rowling"
},
{
    "credit_id": "52fe4273c3a36847f801fa81",
    "department": "Directing",
    "gender": 2,
    "id": 11343,
    "job": "Director",
    "name": "David Yates"
},
...
]
```

4.3 Xử lý dữ liệu

Bây giờ, chúng ta sẽ tiến hành đầy dữ liệu từ các tệp tin ở trên vào cơ sở dữ liệu Neo4j bằng câu lệnh Cypher.

Cơ sở dữ liệu cho bài toán được thiết kế ở dưới dạng như sau:



Hình 42: File tags.csv

Chúng ta sẽ tiến hành tạo nhãn Movie với các thuộc tính là id, title và nhãn Genre cùng với thuộc tính là title:

```
LOAD CSV WITH HEADERS FROM "file:///movies.csv" AS line
MERGE (m:Movie{ id:line.movieId, title:line.title})
FOREACH (gName IN split(line.genres, '|') |
    MERGE (g:Genre {name:gName})
    MERGE (m)-[:IS_GENRE]->(g)
)
```

Tiếp theo, chúng ta sẽ tạo nhãn User với thuộc tính là id, quan hệ là RATED với thuộc tính là điểm rating: (User)-[:RATED rating]->(Movie)

```
LOAD CSV WITH HEADERS FROM "file:///ratings.csv" AS line
MATCH (m:Movie {id:line.movieId})
MERGE (u:User {id:line.userId})
MERGE (u)-[:RATED { rating:toFloat(line.rating)}]->(m);
```



Chúng ta tạo quan hệ người dùng gắn nhãn cho bộ phim: (User)-[:TAGGED tag]->(Movie)

```
LOAD CSV WITH HEADERS FROM "file:///tags.csv" AS line
MATCH (m:Movie {id:line.movieId})
MATCH (u:User {id:line.userId})
CREATE (u)-[:TAGGED {tag: line.tag}]->(m);
```

Sau khi ta xử lý tệp tmdb_5000_credits.csv với 2 cột là cast và crew thì ta sẽ được thông tin về các diễn viên và đạo diễn của phim:

movield	person_name
19995	Sam Worthington
19995	Zoe Saldana
19995	Sigourney Weaver
19995	Stephen Lang
19995	Michelle Rodriguez
19995	Giovanni Ribisi
19995	Joel David Moore
19995	CCH Pounder
19995	Wes Studi
19995	Laz Alonso
19995	Scott Lawrence
19995	Peter Mensah
19995	Terry Notary
285	Johnny Depp
285	Orlando Bloom
285	Keira Knightley

Hình 43: File directors.csv

movieId	person_name	role
19995	Sam Worthington	Jake Sully
19995	Zoe Saldana	Neytiri
19995	Sigourney Weaver	Dr. Grace Augustine
19995	Stephen Lang	Col. Quaritch
19995	Michelle Rodriguez	Trudy Chacon
19995	Giovanni Ribisi	Selfridge
19995	Joel David Moore	Norm Spellman
19995	CCH Pounder	Moat
19995	Wes Studi	Eytukan
19995	Laz Alonso	Tsu'Tey
19995	Scott Lawrence	Venture Star Crew Chief
19995	Peter Mensah	Horse Clan Leader
19995	Terry Notary	Banshee (uncredited)
285	Johnny Depp	Captain Jack Sparrow
285	Orlando Bloom	Will Turner
285	Keira Knightley	Elizabeth Swann

Hình 44: File actors.csv

Bây giờ ta sẽ tiến hành tạo nhãn Person (có quan hệ là DIRECTED và ACTED_IN) với nhãn Movie vào cơ sở dữ liệu Neo4j:

```
LOAD CSV WITH HEADERS FROM "file:///directors.csv" AS line
MATCH (m:Movie{tmdbId:line.movieId})
MERGE (p:Person{name:line.person_name})
MERGE (p)-[:DIRECTED]->(m);

LOAD CSV WITH HEADERS FROM "file:///actors.csv" AS line
MATCH (m:Movie{tmdbId:line.movieId})
MERGE (p:Person{name:line.person_name})
CREATE (p)-[r:ACTED_IN] ->(m)
SET r.role= line.role;
```

4.4 Demo ứng dụng

Ứng dụng được lập trình bằng ngôn ngữ Python với thư viện Flask kết hợp với hệ quản trị cơ sở dữ liệu ở dạng đồ thị Neo4j. Ứng dụng này làm về 1 trang web xem phim trực tuyến và cho phép người dùng có những tính năng như đăng nhập, đăng ký, xem danh sách bộ phim yêu thích, đánh giá bộ phim và liệt kê thông tin chi tiết về bộ phim (như thể loại, đạo diễn, diễn viên, ...). Dưới đây là link demo ứng dụng của nhóm:

<https://www.youtube.com/watch?v=OEQCy4BHpSQ>

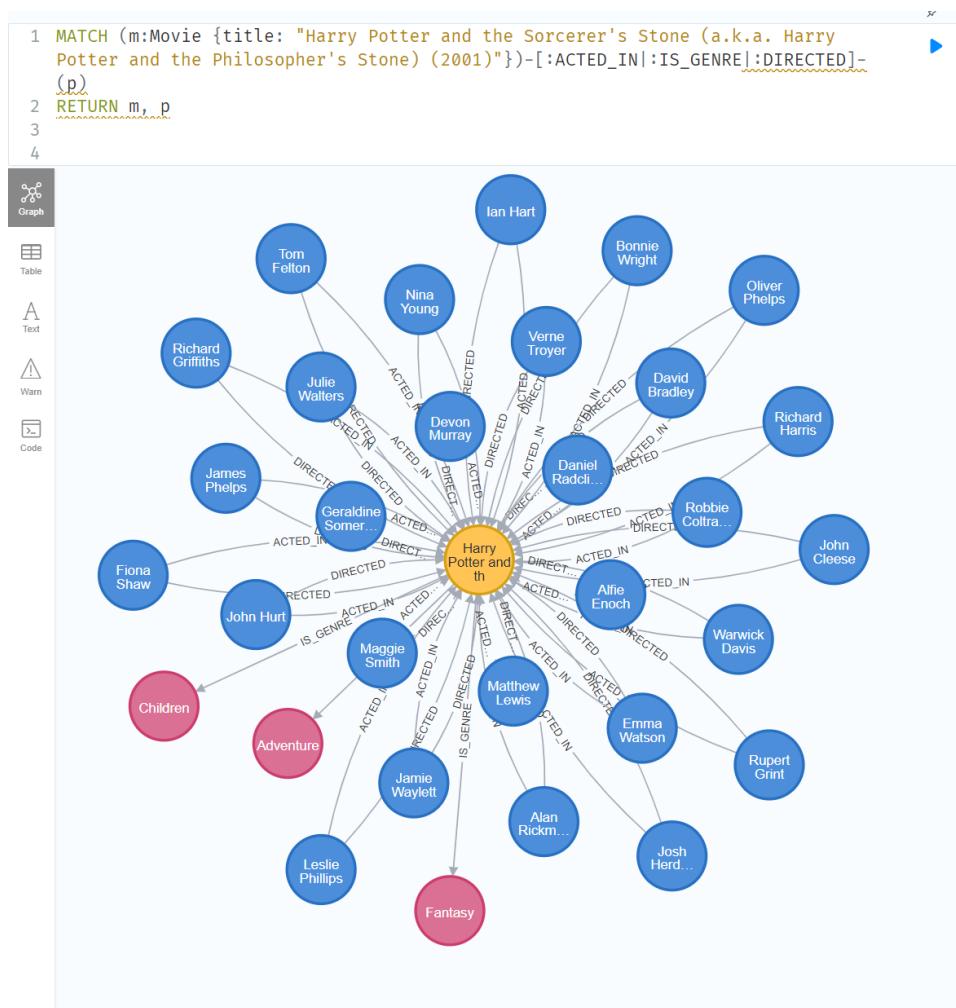
4.5 Bonus

Trong phần bonus này, chúng ta sẽ tiến hành tạo Recommendation cho hệ thống bằng cơ sở dữ liệu đang đồ thị.

Bây giờ ta sẽ tiến hành khảo sát sơ qua dữ liệu: Vd: Xem tất cả các thể loại, diễn viên và đạo diễn của một bộ phim tên là "Harry Potter and the Sorcerer's Stone":

```
MATCH (m:Movie {title: "Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)"} )-[:ACTED_IN|:IS_GENRE|:DIRECTED]-(p)
RETURN m, p
```

Kết quả trả về:

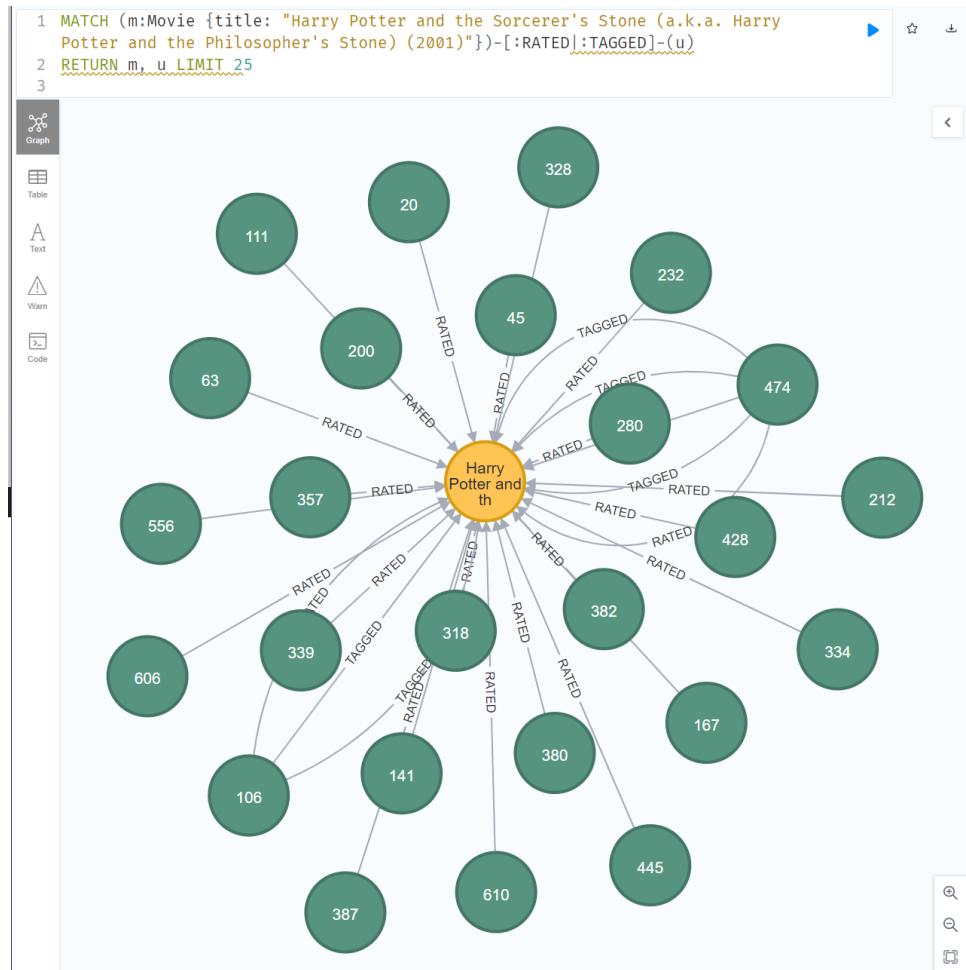


Hình 45: Kết quả trả về thể loại, diễn viên và đạo diễn của một bộ phim tên là "Harry Potter and the Sorcerer's Stone"

Vd: Xem những người đã xếp hạng hoặc gắn nhãn 1 bộ phim có tên "Harry Potter and the Sorcerer's Stone":

```
MATCH (m:Movie {title: "Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)"} )-[:RATED|:TAGGED]-(u)
RETURN m, u LIMIT 25
```

Kết quả trả về:



Hình 46: Kết quả trả về những người đã xếp hạng hoặc gắn nhãn 1 bộ phim "Harry Potter and the Sorcerer's Stone"

Sau khi đã làm quen sơ với dữ liệu thì ta sẽ đi xây dựng 1 số thuật toán cho bài toán gợi ý phim cho người dùng.

Collaborative Filtering: Cách tiếp cận này sẽ lọc ra các phim mà người dùng có thể thích dựa trên cơ sở hành vi của những người dùng tương tự.

Ta sẽ đi tìm những bộ phim mà người dùng thích, sau đó tìm những người dùng khác cũng thích bộ phim đó và đề xuất những bộ phim mà những người dùng khác thích nhưng mà người dùng của chúng ta chưa xem (xếp hạng), kết quả sẽ được sắp xếp theo thứ tự phim có thể thích giảm dần.

```

MATCH (me:User{id: '220'})-[r1:RATED]-(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]->(m2:Movie)
WHERE r1.rating > 3 AND r2.rating > 3 AND r3.rating > 3 AND NOT (me)-[:RATED]->(m2)
RETURN distinct m2 AS recommended_movie, count(*) AS score
ORDER BY score DESC
LIMIT 15
    
```

Kết quả trả về:



```
1 MATCH (me:User{id:'220'})-[r1:RATED]→(m:Movie)←[r2:RATED]-(other:User)-  
[r3:RATED]→(m2:Movie)  
2 WHERE r1.rating > 3 AND r2.rating > 3 AND r3.rating > 3 AND NOT (me)-[:RATED]→  
(m2)  
3 RETURN distinct m2 AS recommended_movie, count(*) AS score  
4 ORDER BY score DESC  
5 LIMIT 15  
6  
7  
8
```

Graph

Table

Text

Code

"recommended_movie"	"score"
{"tmdbId": "274", "id": "593", "title": "Silence of the Lambs, The (1991)"}	7203
{"tmdbId": "120", "id": "4993", "title": "Lord of the Rings: The Fellowship of the Ring, The (2001)"}	6563
{"tmdbId": "14", "id": "2858", "title": "American Beauty (1999)"}	6227
{"tmdbId": "197", "id": "110", "title": "Braveheart (1995)"}	5894
{"tmdbId": "98", "id": "3578", "title": "Gladiator (2000)"}	5777
{"tmdbId": "424", "id": "527", "title": "Schindler's List (1993)"}	5663
{"tmdbId": "762", "id": "1130", "title": "Monty Python and the Holy Grail ([1975])"}	5377
{"tmdbId": "161", "id": "4963", "title": "Ocean's Eleven (2001)"}	5011
{"tmdbId": "348", "id": "1214", "title": "Alien (1979)"}	4951
{"tmdbId": "37165", "id": "1682", "title": "Truman Show, The (1998)"}	4867
{"tmdbId": "38", "id": "7361", "title": "Eternal Sunshine of the Spotless Mind (2004)"}	4726
{"tmdbId": "5503", "id": "457", "title": "Fugitive, The (1993)"}	4687
{"tmdbId": "180", "id": "5445", "title": "Minority Report (2002)"}	4610
{"tmdbId": "240", "id": "1221", "title": "Godfather: Part II, The (1974)"}	4599
{"tmdbId": "1422", "id": "48516", "title": "Departed, The (2006)"}	4446

Hình 47: Kết quả trả về đề xuất phim cho người dùng có id là "220"

Ta sẽ lọc theo xếp hạng đánh giá phim trung bình của người dùng cụ thể, thay vì gán chỉ số định "3":

```
MATCH (me:User{id:'220'})-[r:RATED]-(m)  
WITH me, avg(r.rating) AS average  
MATCH (me)-[r1:RATED]→(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]→(m2:Movie)  
WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND NOT (me)-[:  
RATED]→(m2)  
RETURN distinct m2 AS recommended_movie, count(*) AS score  
ORDER BY score DESC  
LIMIT 15
```

Kết quả trả về:



```
1 MATCH (me:User{id:'220'})-[r:RATED]-(m)
2 WITH me, avg(r.rating) AS average
3 MATCH (me)-[r1:RATED]→(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]→
(m2:Movie)
4 WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND
NOT (me)-[:RATED]→(m2)
5 RETURN distinct m2 AS recommended_movie, count(*) AS score
6 ORDER BY score DESC
7 LIMIT 15
8
```

Graph

Table

A Text

Code

"recommended_movie"	"score"
{"tmdbId": "274", "id": "593", "title": "Silence of the Lambs, The (1991)"}	5322
{"tmdbId": "120", "id": "4993", "title": "Lord of the Rings: The Fellowship of the Ring, The (2001)"}	4276
{"tmdbId": "14", "id": "2858", "title": "American Beauty (1999)"}	4129
{"tmdbId": "424", "id": "527", "title": "Schindler's List (1993)"}	4086
{"tmdbId": "197", "id": "110", "title": "Braveheart (1995)"}	3982
{"tmdbId": "98", "id": "3578", "title": "Gladiator (2000)"}	3537
{"tmdbId": "348", "id": "1214", "title": "Alien (1979)"}	3502
{"tmdbId": "762", "id": "1136", "title": "Monty Python and the Holy Grail (1975)"}	3408
{"tmdbId": "240", "id": "1221", "title": "Godfather: Part II, The (1974)"}	3330
{"tmdbId": "5503", "id": "457", "title": "Fugitive, The (1993)"}	3222
{"tmdbId": "38", "id": "7361", "title": "Eternal Sunshine of the Spotless Mind (2004)"}	3076
{"tmdbId": "1422", "id": "48516", "title": "Departed, The (2006)"}	3062
{"tmdbId": "161", "id": "4963", "title": "Ocean's Eleven (2001)"}	2931
{"tmdbId": "27205", "id": "79132", "title": "Inception (2010)"}	2926
{"tmdbId": "37165", "id": "1682", "title": "Truman Show, The (1998)"}	2853

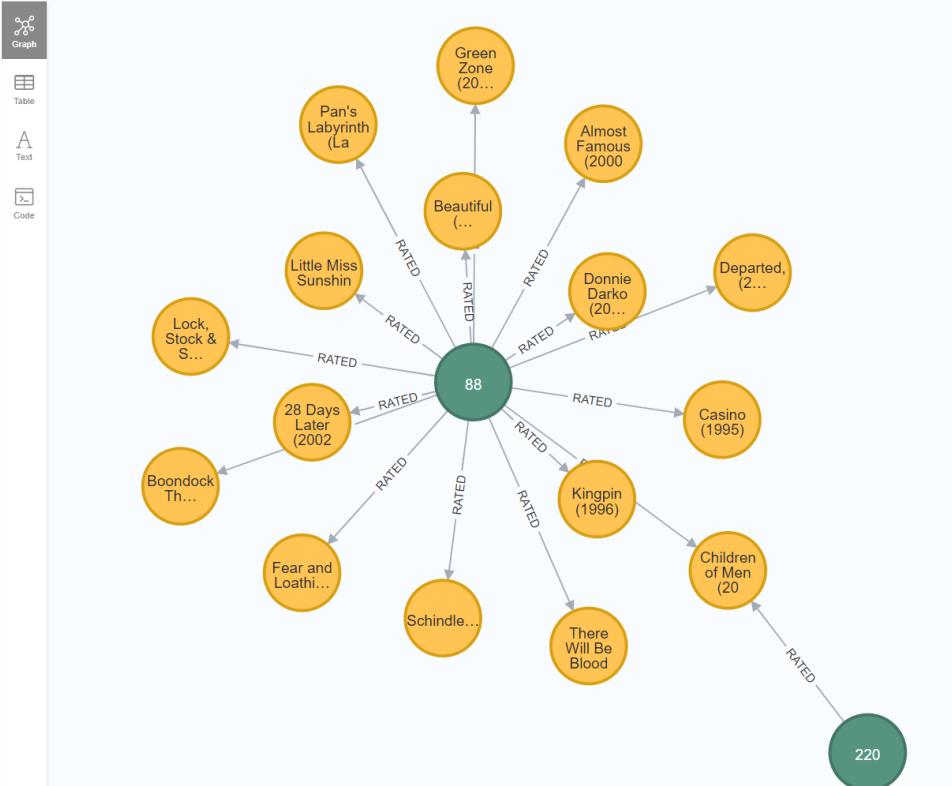
Hình 48: Kết quả trả về đề xuất phim cho người dùng có id là "220" (improved version)

Sau đây là minh họa hình ảnh cho câu truy vấn ở trên:

```
MATCH (me:User{id: '220'})-[r:RATED]-(m)
WITH me, avg(r.rating) AS average
MATCH (me)-[r1:RATED]→(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]→(m2:Movie)
WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND NOT (me)-[:RATED]→(m2)
RETURN distinct m2, other, me, m AS recommended_movie, count(m2) AS score
ORDER BY score DESC
LIMIT 15
```

```

1 MATCH (me:User{id:'220'})-[r:RATED]-(m)
2 WITH me, avg(r.rating) AS average
3 MATCH (me)-[r1:RATED]→(m:Movie)←[r2:RATED]-(other:User)-[r3:RATED]→
(m2:Movie)
4 WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND
NOT (me)-[:RATED]→(m2)
5 RETURN distinct m2, other, me, m AS recommended_movie, count(m2) AS score
6 ORDER BY score DESC
7 LIMIT 15
  
```



Hình 49: Danh sách bộ phim đề xuất cho người dùng có id là 220

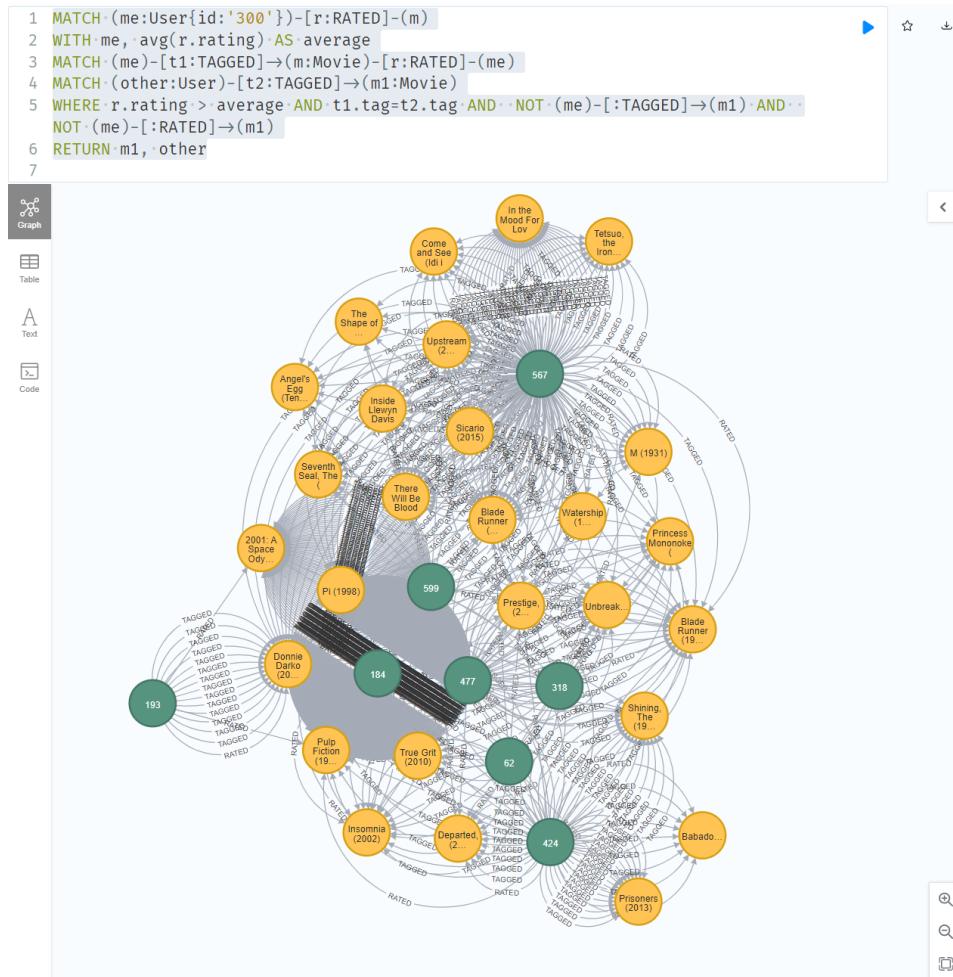
Tuy nhiên thì cách làm trên vẫn còn hạn chế bởi vì có thể có người dùng dễ tính và cũng có người dùng khó tính, cách làm trên chỉ xét cho trung bình đánh giá của người dùng hệ thống đang muốn đề xuất nhưng chưa so sánh với trung bình đánh giá của người dùng khác cũng xem bộ phim đó.

Bây giờ chúng ta sẽ dùng thẻ (Tags) mà người dùng gắn nhãn cho bộ phim người đó thích và dùng nó để tìm những bộ phim có chung thẻ (Tags) rồi gợi ý chúng cho người dùng đó.

```

MATCH (me:User{id:'300'})-[r:RATED]-(m)
WITH me, avg(r.rating) AS average
MATCH (me)-[t1:TAGGED]→(m:Movie)-[r:RATED]-(me)
MATCH (other:User)-[t2:TAGGED]→(m1:Movie)
WHERE r.rating > average AND t1.tag=t2.tag AND NOT (me)-[:TAGGED]→(m1) AND NOT (me)-[:RATED]→(m1)
RETURN m1, other
  
```

Kết quả trả về:



Hình 50: Danh sách bộ phim đề xuất cho người dùng có id là 300 dựa trên nhãn Tags

Mỗi bộ phim trong biểu đồ con này chứa một thẻ mà người dùng có id 300 thích.

Content-Based Filtering:

Bây giờ, chúng ta sẽ sử dụng phương pháp collaborative kết hợp với thông tin về nội dung của bộ phim (thông tin về diễn viên, đạo diễn và thể loại).

Trước tiên, ta tìm các diễn viên trong các bộ phim mà người dùng thích được sắp xếp theo số lần diễn viên cụ thể xuất hiện trong các bộ phim đó:

```

MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
WITH me, avg(r.rating) AS average
MATCH (me)-[r:RATED]->(m:Movie)-[:ACTED_IN]-(p:Person)
WHERE r.rating > average
RETURN p AS actor, COUNT(*) AS score
ORDER BY score DESC LIMIT 10
  
```

Kết quả trả về:



```
1 MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
2 WITH me, avg(r.rating) AS average
3 MATCH (me)-[r:RATED]->(m:Movie)-[:ACTED_IN]-(p:Person)
4 WHERE r.rating > average
5 RETURN p as actor, COUNT(*) AS score
6 ORDER BY score DESC LIMIT 10
```

Graph

Table

A Text

Code

actor	score
{"name": "Harrison Ford"}	7
{"name": "Ian Holm"}	5
{"name": "Kenny Baker"}	5
{"name": "Willem Dafoe"}	5
{"name": "Burnell Tucker"}	5
{"name": "Alec Guinness"}	4
{"name": "Anthony Daniels"}	4
{"name": "James Earl Jones"}	4
{"name": "Dick Miller"}	4
{"name": "Jim Broadbent"}	4

Hình 51: Danh sách các diễn viên mà người dùng có id 312 thích theo thứ tự giảm dần

Nhìn kết quả thì ta biết được là người dùng có id 312 thích diễn viên tên là "Harrison Ford".
Tương tự cho các đạo diễn các bộ phim mà người dùng có id 312 thích:

```
MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
WITH me, avg(r.rating) AS average
MATCH (me)-[r:RATED]->(m:Movie)-[:DIRECTED]-(p:Person)
WHERE r.rating > average
RETURN p as director, COUNT(*) AS score
ORDER BY score DESC LIMIT 10
```

Kết quả trả về:

```
1 MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
2 WITH me, avg(r.rating) AS average
3 MATCH (me)-[r:RATED]->(m:Movie)-[:DIRECTED]-(p:Person)
4 WHERE r.rating > average
5 RETURN p as director, COUNT(*) AS score
6 ORDER BY score DESC LIMIT 10
```

Graph

Table

A Text

Code

director	score
{"name": "Steven Spielberg"}	8
{"name": "Martin Scorsese"}	4
{"name": "James Cameron"}	4
{"name": "Robert Zemeckis"}	4
{"name": "Joe Dante"}	3
{"name": "Peter Jackson"}	2
{"name": "Terry Gilliam"}	2
{"name": "George Lucas"}	2
{"name": "John Carpenter"}	2
{"name": "Ridley Scott"}	2

Hình 52: Danh sách các đạo diễn mà người dùng có id 312 thích theo thứ tự giảm dần

Và chúng ta làm tương tự cho thể loại phim mà người dùng thích:



```
MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
WITH me, avg(r.rating) AS average
MATCH (me)-[r:RATED]->(m:Movie)-[:IS_GENRE]-(p:Genre)
WHERE r.rating > average
RETURN p.name AS genre, COUNT(*) AS score
ORDER BY score DESC LIMIT 10
```

Kết quả trả về:

The screenshot shows a Neo4j browser window. On the left, there are three tabs: 'Table' (selected), 'Text', and 'Code'. The 'Text' tab contains the Cypher query shown above. The 'Table' tab displays the results in a grid format:

genre	score
"Drama"	68
"Sci-Fi"	49
"Thriller"	47
"Action"	37
"Horror"	35
"Adventure"	29
"Comedy"	28
"Crime"	21
"War"	20
"Mystery"	19

Hình 53: Danh sách các đạo diễn mà người dùng có id 312 thích theo thứ tự giảm dần

Bây giờ, chúng ta sẽ sử dụng thông tin kết hợp về diễn viên, đạo diễn và thể loại yêu thích để cung cấp cho người dùng phim được đề xuất và sắp xếp theo trọng số yêu thích giảm dần:

```
MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
WITH me, avg(r.rating) AS average
MATCH (me)-[r:RATED]->(m:Movie)
WHERE r.rating > average
MATCH (m)-[:IS_GENRE]->(g:Genre)<-[:IS_GENRE]-(rm:Movie)
WITH me, m, rm, COUNT(*) AS gs
OPTIONAL MATCH (m)<-[ACTED_IN]-(a:Person)-[:ACTED_IN]->(rm)
WITH me, m, rm, gs, COUNT(a) AS as
OPTIONAL MATCH (m)<-[DIRECTED]-(d:Person)-[:DIRECTED]->(rm)
WITH me, m, rm, gs, as, COUNT(d) AS ds
MATCH (rm)
WHERE NOT (me)-[:RATED]->(rm)
RETURN rm.title AS recommendation,
gs AS genre_score, as AS actor_score, ds AS director_score,
(5*gs)+(2*as)+(5*ds) AS weighed_score
ORDER BY weighed_score DESC LIMIT 10
```

Kết quả trả về:



The screenshot shows a Neo4j browser window. At the top, there is a query in the Cypher editor:

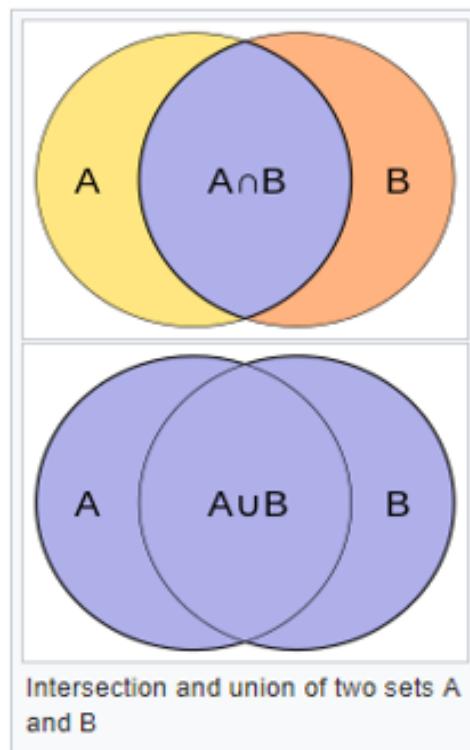
```
1 MATCH (me:User{id:'312'})-[r:RATED]-(m:Movie)
2 WITH me, avg(r.rating) AS average
3 MATCH (me)-[r:RATED]-(m:Movie)
4 WHERE r.rating > average
5 MATCH (m)-[:IS_GENRE]-(g:Genre)<-[IS_GENRE]-(rm:Movie)
6 WITH me, m, rm, COUNT(*) AS gs
7 OPTIONAL MATCH (m)<[:ACTED_IN]-(a:Person)-[:ACTED_IN]-(rm)
8 WITH me, m, rm, gs, COUNT(a) AS as
9 OPTIONAL MATCH (m)<[:DIRECTED]-(d:Person)-[:DIRECTED]-(rm)
10 WITH me, m, rm, gs, as, COUNT(d) AS ds
11 MATCH (rm)
12 WHERE NOT (me)-[:RATED]-(rm)
```

Below the query, there are three tabs: "Table", "Text", and "Code". The "Table" tab is selected, displaying the results in a table format:

	recommendation	genre_score	actor_score	director_score	weighed_score
1	"Lord of the Rings: The Return of the King, The (2003)"	2	16	1	47
2	"Star Wars: Episode III - Revenge of the Sith (2005)"	3	11	1	42
3	"Lord of the Rings: The Fellowship of the Ring, The (2001)"	2	12	1	39
4	"Back to the Future (1985)"	3	9	1	38
5	"Back to the Future (1985)"	3	5	1	30
6	"Hobbit: An Unexpected Journey, The (2012)"	2	7	1	29
7	"Evil Dead II (Dead by Dawn) (1987)"	4	2	1	29
8	"Star Wars: Episode III - Revenge of the Sith (2005)"	3	6	0	27
9	"The Hobbit: The Desolation of Smaug (2013)"	2	6	1	27
10	"Star Wars: Episode III - Revenge of the Sith (2005)"	3	3	1	26

Hình 54: Danh sách bộ phim đề xuất cho người dùng id 312 theo thứ tự trọng số yêu thích giảm dần

Bây giờ, chúng ta sẽ sử dụng chỉ mục Jaccard (content-Based Similarity Metrics) làm chỉ số tương tự. Nó được tính bằng cardinality (số phần tử) của giao của 2 tập hợp chia cho cardinality của hợp của 2 tập hợp:



$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

$$0 \leq J(A, B) \leq 1$$

Hình 55: Jaccard

Vd: chúng ta sẽ đề xuất phim cho người dùng có id là 220 dựa trên bộ phim mà anh ấy thích nhất (đánh giá 5 sao), dựa trên độ đo tương đồng là jaccard cho bộ phim đề xuất.

```

MATCH (me:User{id:'220'})-[r:RATED]-(m:Movie)
WITH me, avg(r.rating) AS mean
MATCH (me)-[r:RATED]->(m:Movie)
WHERE r.rating =5
MATCH (m)-[:ACTED_IN|:DIRECTED]-(t)-[:ACTED_IN|:DIRECTED]-(other:Movie)
WHERE NOT (me)-[:RATED]->(other)
WITH me, m, other, COUNT(t) AS intersection, COLLECT(t.name) AS i
MATCH (m)-[:ACTED_IN|:DIRECTED]-(mt)
WITH me, m, other, intersection, i, COLLECT(mt.name) AS s1
MATCH (other)-[:ACTED_IN|:DIRECTED]-(ot)
WITH me, m, other, intersection, i, s1, COLLECT(ot.name) AS s2
WITH me, m, other, intersection, s1, s2
WITH me, m, other, intersection, s1+ [x IN s2 WHERE NOT x IN s1] AS union, s1, s2
RETURN m.title, other.title, s1,s2,((1.0*intersection)/SIZE(union)) AS jaccard ORDER BY
jaccard DESC LIMIT 20
    
```

Kết quả trả về:



"m.title"	"other.title"	"s1"	"s2"	"jaccard"
"Casablanca (1942)"	"Top Hat (1935)"	["Gino Corrado"]	["Gino Corrado"]	1.0
"Pirates of the Caribbean: The Curse of the Black Pearl (2003)"	"Pirates of the Caribbean: At World's End (2007)"	["Gore Verbinski", "Annie Mumolo", "Gus Barnett", "Johnny Depp", "Martin Klebba", "Jack Davenport", "Maggie Q", "Reggie Lee", "Geoffrey Rush", "Angus Barnacle", "Bainbridge", "Kevin McNally", "Bainbridge", "Jonathan Pryce", "David Bailie", "Chow Yun-fat", "Tom Holland", "Keira Knightley", "Gregory Hines", "Geoffrey Rush", "David Schofield", "Zoe Saldana", "Oscar Isaac", "Jack Davenport", "Natalie Portman", "Rlando Bloom"]	["Gore Verbinski", "Johnny Depp", "Jonathan Pryce", "Martin Klebba", "Reggie Lee", "Geoffrey Rush", "Angus Barnacle", "Bainbridge", "Kevin McNally", "Bainbridge", "Jonathan Pryce", "David Bailie", "Chow Yun-fat", "Tom Holland", "Keira Knightley", "Gregory Hines", "Geoffrey Rush", "David Schofield", "Zoe Saldana", "Oscar Isaac", "Jack Davenport", "Natalie Portman", "Rlando Bloom"]	0.5909090909090909
"Clerks (1994)"	"Clerks II (2006)"	["Kevin Smith", "Kevin Smith", "Jason Mewes", "Scott Mosier", "Brian O'Halloran"]	["Ben Affleck", "Wanda Sykes", "Kevin Michael Richardson", "Jason Lee", "Rosario Dawson", "Jason Mewes", "Scott Mosier", "Kevin Smith", "Jennifer Schwalbach Smith", "Ethan Suplee", "Brian O'Halloran", "Kevin Smith"]	0.5833333333333334
"Clerks (1994)"	"Chasing Amy (1997)"	["Kevin Smith", "Kevin Smith", "Jason Mewes", "Scott Mosier", "Brian O'Halloran"]	["Kevin Smith", "Scott Mosier", "Bria O'Halloran", "Joey Lauren Adams", "Matt Damon", "Ben Affleck", "Ethan Suplee", "Jason Mewes", "Casey Affleck", "Illeana Douglas", "Jason Lee"]	0.5833333333333334

Hình 56: Danh sách bộ phim đề xuất cho người dùng id 220 theo hàm độ đo tương tự jaccard

Bây giờ ta sẽ trở lại phương pháp **Collaborative Filtering**. Thay vì xem xét ý kiến của tất cả người dùng trong hệ thống, hãy tìm những người dùng “tương tự” nhất (người dùng có cùng sở thích). Cách dễ nhất để làm như vậy là tìm hệ số tương quan giữa người dùng nhầm đến và những người dùng khác, sau đó sử dụng xếp hạng để đưa ra những người dùng “có cùng sở thích” nhất. Chúng ta sẽ sử dụng hệ số tương quan Pearson, được định nghĩa như sau:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Hình 57: Pearson's formula

Trong đó:

- n là độ lớn mẫu
- x_i , y_i là những điểm mẫu có index tại i;
- \bar{x} là trung bình mẫu x; và \bar{y} là trung bình mẫu y.

Chúng ta sẽ đi tìm những người dùng có hệ số tương quan lớn đối với người dùng có id là 220. (Tức là người dùng có khả năng có cùng ý nghĩ cũng như sở thích về bộ phim):



```
MATCH (me:User {id: "220"})-[r:RATED]->(m:Movie)
WITH me, avg(r.rating) AS my_average
MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)
WITH me, my_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10
MATCH (other)-[r:RATED]->(m:Movie)
WITH me, my_average, other, avg(r.rating) AS other_average, ratings
UNWIND ratings AS r
WITH sum( (r.r1.rating - my_average) * (r.r2.rating - other_average) ) AS a,
sqrt( sum( (r.r1.rating - my_average)^2) * sum( (r.r2.rating - other_average)^2)) AS b,
me, other
WHERE b <> 0
RETURN me.id, other.id, a/b AS correlation
ORDER BY correlation DESC LIMIT 10
```

Kết quả trả về:

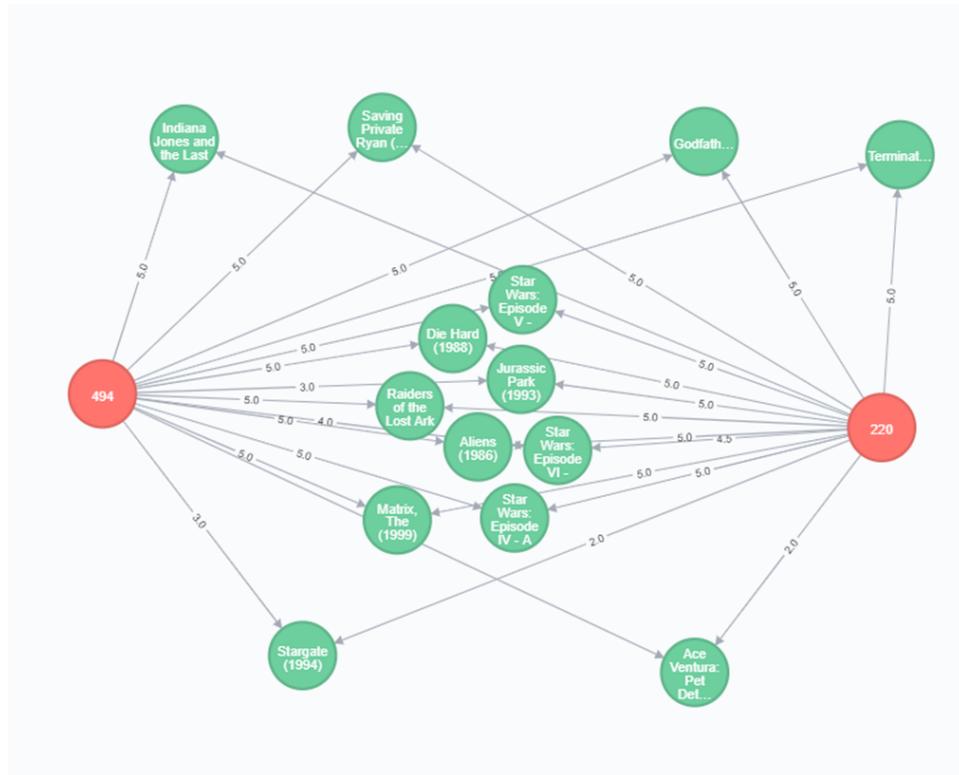
	"me.id"	"other.id"	"correlation"
	"220"	"494"	0.7825315077845472
	"220"	"32"	0.7818916367269144
	"220"	"485"	0.7633105914491698
	"220"	"97"	0.7547965924537345
	"220"	"79"	0.7399103445131806
	"220"	"88"	0.7328107458190379
	"220"	"124"	0.7185930116701769
	"220"	"235"	0.716539350928329
	"220"	"436"	0.7043763345369647
	"220"	"500"	0.6597414172261901

Hình 58: Danh sách người dùng tương đồng với người dùng có id là 220

Như đã thấy ở trên thì người dùng 494 có độ tương đồng giống người dùng 220 nhất, bây giờ chúng ta sẽ tiến hành khảo sát để minh họa xem liệu chúng tương đồng như thế nào:

```
MATCH (me:User {id: "220"})-[:RATED]->(m:Movie)
MATCH (other:User {id: "494"})-[:RATED]->(m:Movie)
RETURN me, other, m
```

Kết quả:



Hình 59: Minh họa độ tương đồng giữa người dùng id 220 và người dùng id 494

Như chúng ta cũng có thể thấy được thì khi người dùng id 220 đánh giá cao 1 bộ phim thì người dùng 494 cũng có xu hướng đánh giá cao bộ phim đó, tương tự cho bộ phim bị đánh giá thấp.
Chúng ta sẽ kết hợp lại những ý trên để tạo ra 1 hệ thống đề xuất phim hoàn chỉnh hơn:

```

MATCH (me:User {id: "220"})-[r:RATED]->(m:Movie)
WITH me, avg(r.rating) AS my_average
MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)
WITH me, my_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10
MATCH (other)-[r:RATED]->(m:Movie)
WITH me, my_average, other, avg(r.rating) AS other_average, ratings
UNWIND ratings AS r
WITH sum( (r.r1.rating- my_average) * (r.r2.rating- other_average) ) AS a,
sqrt( sum( (r.r1.rating - my_average)^2) * sum( (r.r2.rating - other_average) ^2)) AS b,
me, other
WHERE b <> 0
WITH me, other, a/b AS correlation
ORDER BY correlation DESC LIMIT 10
MATCH (other)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (me)-[:RATED]->(m) )
WITH m, SUM( correlation* r.rating) AS score, COLLECT(other) AS other
RETURN m, other, score
ORDER BY score DESC LIMIT 10
    
```

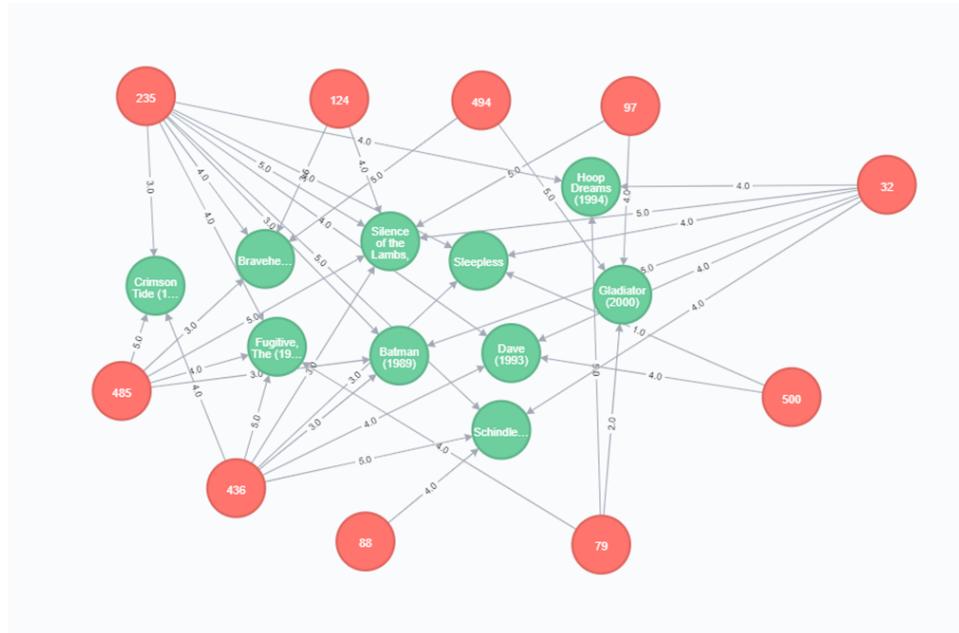
Kết quả:

"m"	"other"	"score"
{"tmdbId": "274", "id": "593", "title": "Silence of the Lambs, The (1991)"}	[{"id": "32"}, {"id": "405"}, {"id": "97"}, {"id": "124"}, {"id": "235"}, {"id": "436"}]	20.070191908082343
{"tmdbId": "424", "id": "527", "title": "Schindler's List (1993)"}	[{"id": "32"}, {"id": "88"}, {"id": "235"}, {"id": "436"}]	13.16338795751028
{"tmdbId": "5503", "id": "457", "title": "Fugitive, The (1993)"}	[{"id": "405"}, {"id": "79"}, {"id": "235"}, {"id": "436"}]	12.400922820247542
{"tmdbId": "197", "id": "110", "title": "Braveheart (1995)"}	[{"id": "494"}, {"id": "485"}, {"id": "124"}, {"id": "235"}]	11.58382225782918
{"tmdbId": "11566", "id": "440", "title": "Dave (1993)"}	[{"id": "32"}, {"id": "235"}, {"id": "436"}, {"id": "500"}]	11.450194957673592
{"tmdbId": "268", "id": "592", "title": "Batman (1989)"}	[{"id": "32"}, {"id": "405"}, {"id": "235"}, {"id": "436"}]	10.462137014377962
{"tmdbId": "14275", "id": "246", "title": "Hoop Dreams (1994)"}	[{"id": "32"}, {"id": "79"}, {"id": "235"}]	9.693275673186877
{"tmdbId": "8963", "id": "161", "title": "Crimson Tide (1995)"}	[{"id": "485"}, {"id": "235"}, {"id": "436"}]	8.783676348178695
{"tmdbId": "858", "id": "539", "title": "Sleepless in Seattle (1993)"}	[{"id": "32"}, {"id": "235"}, {"id": "436"}, {"id": "500"}]	8.766594371458057
{"tmdbId": "98", "id": "3578", "title": "Gladiator (2000)"}	[{"id": "494"}, {"id": "97"}, {"id": "79"}]	8.411664597764036

Hình 60: Dề xuất phim cho người dùng id là 220 dựa trên 10 người dùng tương đồng nhất và xếp hạng theo cách tính toán trọng số thích score giảm dần

Kết quả trên cho ta thấy được tiêu đề phim, danh sách người dùng, ý kiến của những người được xem xét để tính điểm số (bằng tổng số lượng người dùng xếp hạng do người dùng đưa ra nhân với hệ số tương quan của những người dùng này)

Hình minh họa:



Hình 61: Minh họa cho kết quả trả về câu truy vấn trên

Chúng ta cũng có thể thấy được những người dùng có mối tương quan cao với người dùng 220 và xếp hạng của họ đối với các bộ phim đã chọn. 6 người trong số những người dùng đó đã xếp hạng cao cho bộ phim "Silence of the Lambs".



Tương tự, chúng ta có thể tìm thấy những người dùng có độ tương quan nghịch: nếu người dùng được nhầm đến thích một bộ phim cụ thể, thì những người dùng có độ tương quan nghịch cao với người đó sẽ ghét bộ phim đó và ngược lại. Sau đó, chúng ta sẽ có thể sử dụng tính năng ẩn những bộ phim này khỏi người dùng đó để không làm họ cảm thấy khó chịu.

```
MATCH (me:User {id: "220"})-[r:RATED]->(m:Movie)
WITH me, avg(r.rating) AS my_average
MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)
WITH me, my_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10
MATCH (other)-[r:RATED]->(m:Movie)
WITH me, my_average, other, avg(r.rating) AS other_average, ratings
UNWIND ratings AS r
WITH sum( (r.r1.rating - my_average) * (r.r2.rating - other_average) ) AS a,
sqrt( sum( (r.r1.rating - my_average)^2) * sum( (r.r2.rating - other_average)^2)) AS b,
me, other
WHERE b <> 0
WITH me, other, a/b AS correlation
ORDER BY correlation ASC LIMIT 10
MATCH (other)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (me)-[:RATED]->(m) )
WITH m, SUM( correlation * r.rating) AS score, COLLECT(other) AS other
RETURN m, other, score
ORDER BY score ASC LIMIT 10
```

Kết quả:

"m"	"other"	"score"
{"tmdbId": "194", "id": "4973", "title": "Amélie (Fabuleux destin d'Amélie Poulain, Le) (2001)"}		-7.210099377998988
{"tmdbId": "424", "id": "527", "title": "Schindler's List (1993)"}		-6.143148263336504
{"tmdbId": "935", "id": "750", "title": "Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964)"}		-6.019002146132842
{"tmdbId": "641", "id": "3949", "title": "Requiem for a Dream (2000)"}		-5.239954677736696
{"tmdbId": "627", "id": "778", "title": "Trainspotting (1996)"}		-5.13865536762029
{"tmdbId": "68", "id": "1199", "title": "Brazil (1985)"}		-4.380639388558397
{"tmdbId": "600", "id": "1222", "title": "Full Metal Jacket (1987)"}		-4.274487014141835
{"tmdbId": "268", "id": "592", "title": "Batman (1989)"}		-4.177435625142373
{"tmdbId": "14", "id": "2858", "title": "American Beauty (1999)"}		-3.8195079942537506

Hình 62: Kết quả danh sách xếp hạng những bộ phim mà người có id 220 không thích nhất