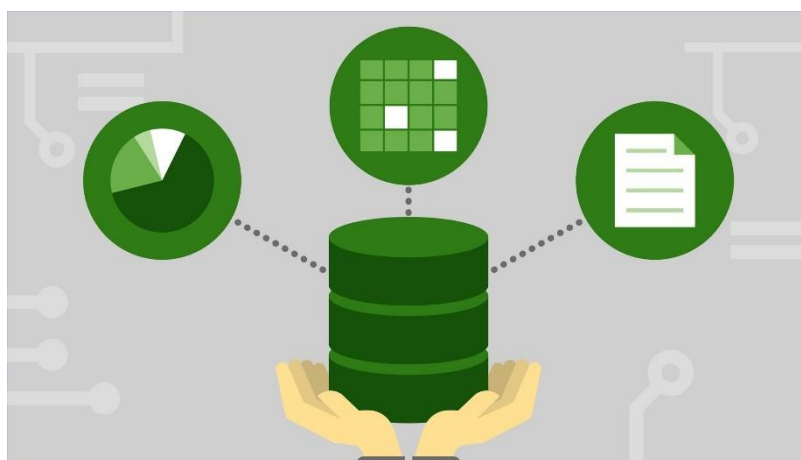


به نام خدا



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



آزمایشگاه پایگاه داده

دستور کار شماره ۳

سجاد علی‌زاده

۸۱۰۱۹۷۵۴۷

آبان ماه ۱۴۰۰

گزارش دستورکار انجام شده

آموزش سایت quackit:

گام اول (About): اطلاعاتی در مورد neo4j به دست آوردم. Neo4j محبوب ترین DBMS گرافی است و یکی از محبوب ترین DBMS های NoSql است.

گام دوم (Installation): در این قسمت توانستم این پایگاه داده را دانلود و نصب کنم. یکی از پیش نیازهای آن openjdk 11 است و ابتدا آن را دانلود کردم و برای کانفیگ کردن آن متغیرهای PATH و JAVA_HOME را تنظیم کردم. پس از آن سرویس neo4i را نصب کردم و سپس آن را اجرا کردم. سپس به آدرس <http://localhost:7474/browser> رفتم. در آنجا از یوزرنیم و پسورد دیفالت که neo4j است استفاده کردم و بعد از آن از من خواسته شد تا پسورد را عوض کنم. بعد از تعویض پسورد به صفحه اصلی هدایت شدم.

گام سوم (browser): در این قسمت با browser آشنا شدم که قابلیت کار با دیتابیس را فراهم میسازد. گام چهارم (Cypher): با زبان جستجوی Neo4j آشنا شدم که مشابه SQL است ولی تفاوت هایی نیز دارد. سپس با مدل داده neo4j آشنا شدم که از نودها و روابط بین آنها تشکیل شده است. بعد از آن با نودها و relation های بین آنها و خواص آنها آشنا شدم. به عنوان مثال:

```
MATCH (p:Person { name:"Homer Flinstone" })
RETURN p
```

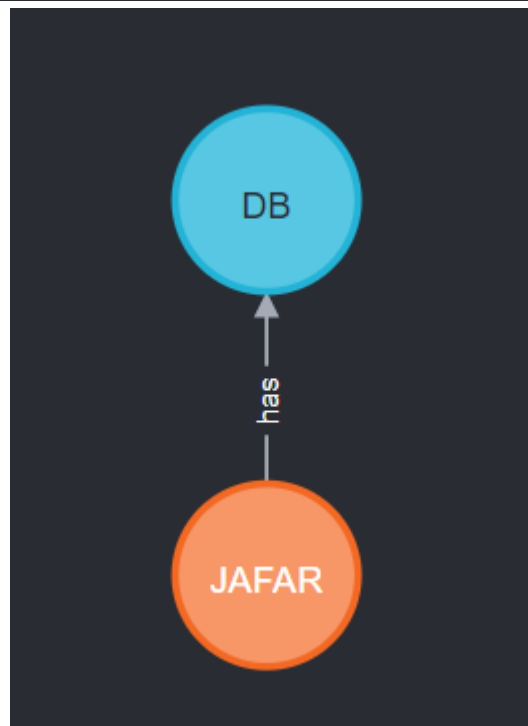
در این کوئری به دنبال نودی هستیم که خصیصه ای به نام name دارد و مقدار این خصیصه برای آن Homer Flinstone است. همچنین لیبل برای این نود Person است. با دقت در این کوئری با سینتکس سایفر نیز آشنا شدم. گام پنجم (Create a Node): در این گام با روش ساختن نود به صورت تکی و یا چندتایی آشنا شدم. مثلاً با دستور زیر یک دانشجو ساختم و نتیجه را مشاهده کردم:

```
1 CREATE (s:Student { Name : "Sajjad", StudentID : "810197547" })
2 RETURN s
```



گام ششم (Create a Relationship): در این مرحله با نحوه ساخت رابطه بین دو نود آشنا شدم. یک دانشجو و یک درس ساختم و بین این دو نود رابطه has اضافه کردم که نتیجه به شکل زیر است:

```
1 MATCH (s:Student), (c:Course)
2 WHERE s.Name = "JAFAR" AND c.Name = "DB"
3 CREATE (s)-[r:has]-(c)
4 RETURN r
```



گام هفتم (Create an index): در این قسمت با ساخت یک ایندکس و روش مشاهده آن آشنا شدم. از ایندکس ها به صورت پیش فرض برای کوئری ها استفاده میشود اما میتوان با دستور using index از این موضوع اطمینان حاصل کرد. به عنوان نمونه، روی نام دانشجویان یک ایندکس تولید میکنیم:

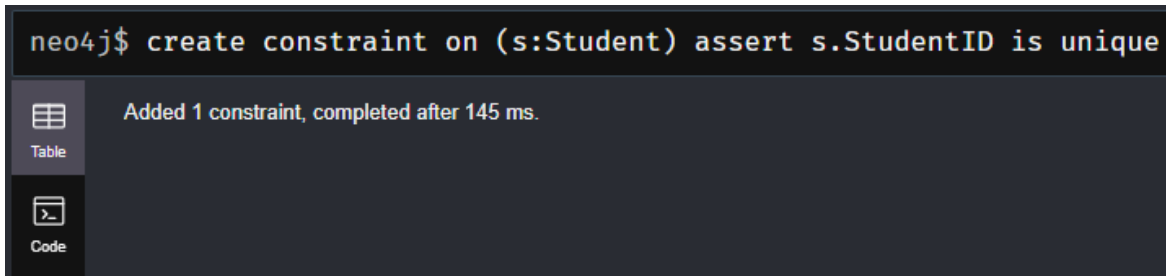
```
neo4j$ create index on :Student(Name)
```

Added 1 index, completed after 16 ms.

Table

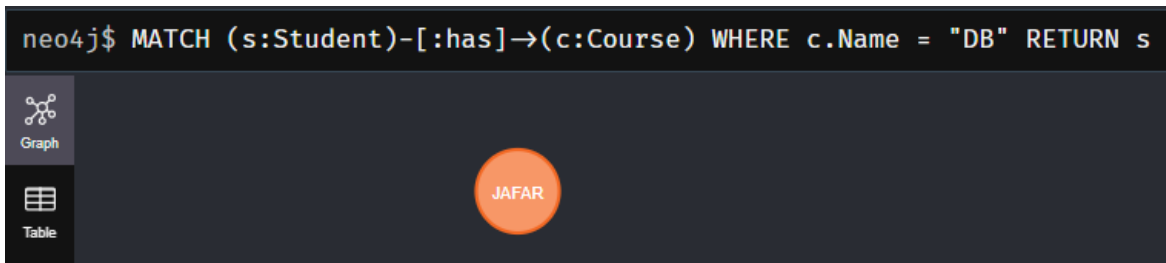
گام هشتم (Create a Constraint): در این قسمت با تعریف constraint روی نودها آشنا شدم. دو نوع محدودیت وجود دارد. یکی محدودیت یکتایی (یعنی مثلاً دو دانشجو با یک شماره دانشجویی وجود نداشته باشند) و دیگری محدودیت property existence یعنی یک نود حتماً یک خاصیت را داشته باشد. (مثلاً هر دانشجو نام داشته باشد) حال به عنوان مثال محدودیت یکتایی را روی نود دانشجو اعمال میکنیم:

```
neo4j$ create constraint on (s:Student) assert s.StudentID is unique
```



گام نهم (Select data): در این گام با نحوه گرفتن داده ها آشنا شدم. این عملیات مانند SELECT در SQL است. میتوان یک نود را با توجه به خصایص آن یا با توجه به روابطی که وجود دارد دریافت کرد. مانند SQL میتوان تمام نودها را دریافت کرد و حتی روی تعداد بازگشتی LIMIT گذاشت. به عنوان مثال برای اینکه بفهمیم کدام دانشجو کلاس دیتابیس را دارد میتوانیم از کوئری زیر استفاده کنیم:

```
neo4j$ MATCH (s:Student)-[:has]→(c:Course) WHERE c.Name = "DB" RETURN s
```

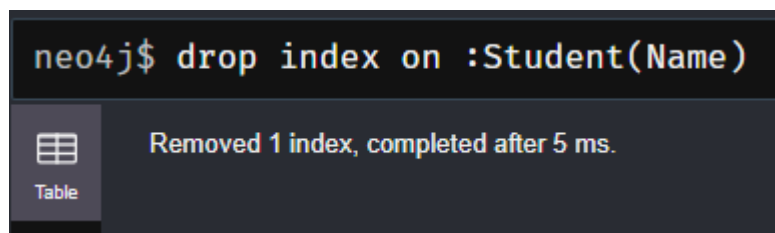


گام دهم (Load a CSV): در این قسمت با روش لود کردن فایل csv آشنا شدم و آپشن های مختلف آن را مشاهده کردم. به طور کلی لود کردن csv با دستور LOAD CSV و بعد از آن استفاده از CREATE امکان پذیر است. مثلاً با دستور زیر دیتابیس ژانرها از آدرس مشخص در دیتابیس لود میشود:

```
LOAD CSV FROM 'https://www.quackit.com/neo4j/tutorial/genres.csv' AS
line
CREATE (:Genre { GenreId: line[0], Name: line[1]})
```

گام یازدهم (Drop an index): در این گام با روش حذف ایندکس آشنا شدم. مثلاً ایندکسی را که روی اسامی دانشجویان ساختیم حذف میکنیم:

```
neo4j$ drop index on :Student(Name)
```



گام دوازدهم (Drop a constraint): در این گام با روش حذف محدودیت آشنا شدم. مثلاً محدودیتی که روی شماره دانشجویی اعمال کردیم را حذف میکنیم:

```
neo4j$ drop constraint on (s:Student) assert s.StudentID is unique
```

Removed 1 constraint, completed after 5 ms.

Table

گام سیزدهم (Delete a node): با دستور حذف یک نود آشنا شدم. میتوان یک نود، چندین نود همزمان و یا همه نودها را همزمان حذف کرد. همچنین اگر یک نود با نود دیگری در رابطه باشد ابتدا باید رابطه را حذف کرد سپس نودها را حذف کرد. (مثال در قسمت بعد)

گام چهاردهم (Delete a relationship): در این قسمت با حذف روابط آشنا شدم. میتوان یک رابطه مشخص را از بین تمام نودها حذف کرد. میتوان نوع نودهای آن را گفت یا حتی میتوان دقیق تر شد و خصیصه های نودها را نیز بیان کرد تا روابط فقط بین آن نودها حذف شود. همچنین روشی ارائه شد تا اگر نودی در رابطه ای است بتوان آن را حذف کرد و در نهایت روشی برای حذف کل دیتابیس ارائه شد. حال برای مثال ابتدا رابطه در کلاس بودن را حذف میکنیم:

```
neo4j$ MATCH ()-[h:has]-() DELETE h
```

Deleted 1 relationship, completed after 13 ms.

Table

Code

سپس نود دانشجو که متعلق به این رابطه بود را حذف میکنیم:

```
neo4j$ MATCH (s:Student {Name: "JAFAR"}) DELETE s
```

Deleted 1 node, completed after 3 ms.

Table

Code

سپس کل دیتابیس را حذف میکنیم:

```
neo4j$ MATCH (n) DETACH DELETE n
```

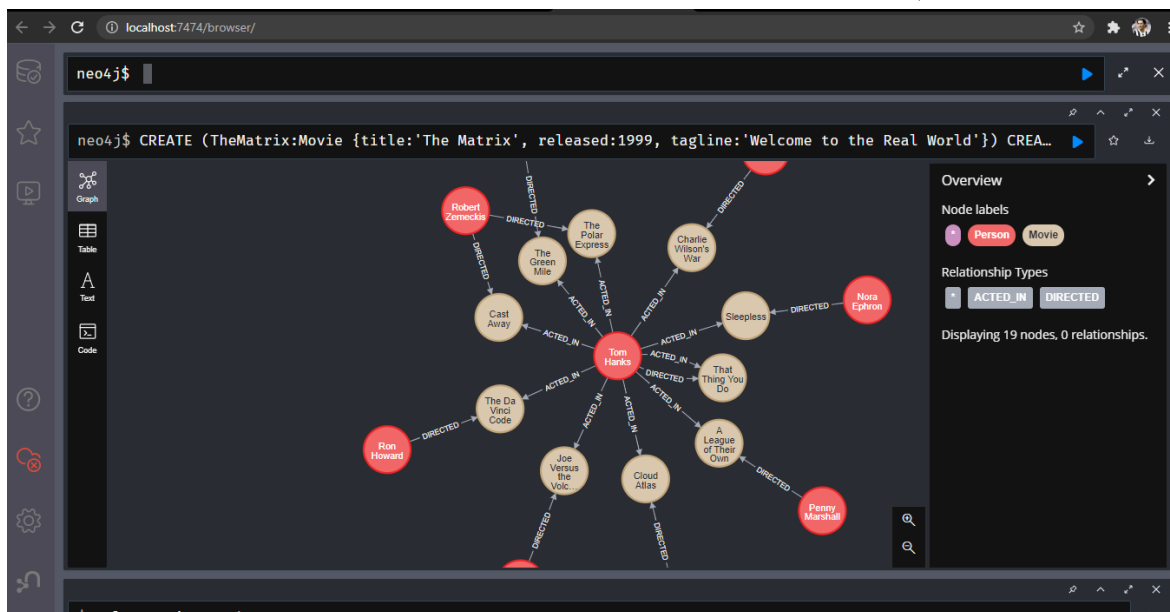
Deleted 4 nodes, deleted 4 relationships, completed after 7 ms.

Table

Code

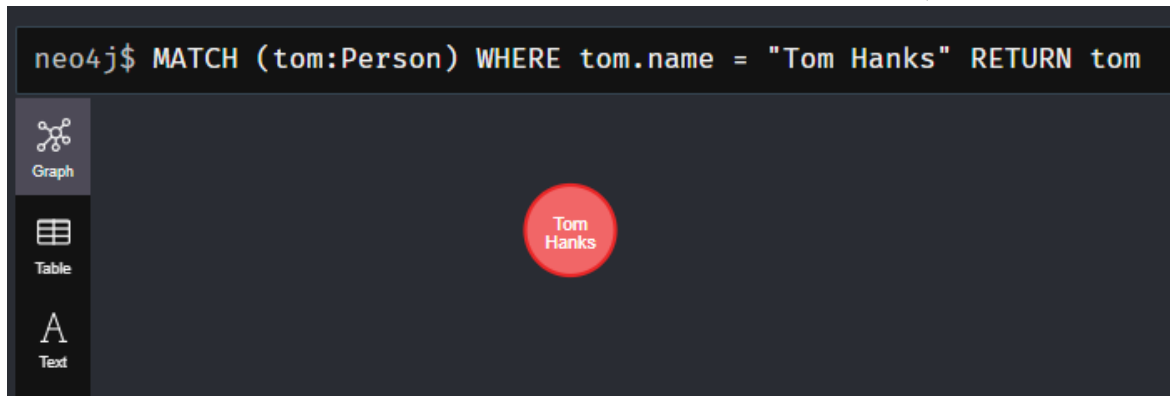
آموزش سایت neo4j:

ابتدا با روش گفته شده در داکيومنتیشن، دیتابیس مربوطه را لود میکنیم و تمام دستورات create آن را اجرا میکنیم. در نهایت به گراف زیر میرسیم:



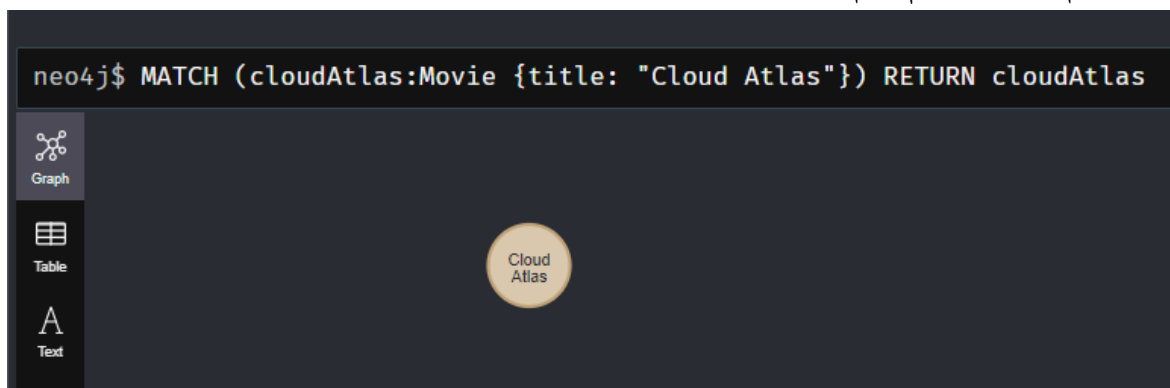
حال در ادامه به هر کوئری میپردازیم و هر کدام را توضیح مختصری میدهم.

کوئری اول: پیدا کردن تام هنکس



با استفاده از match به دنبال person ای خواهیم بود که لیبل name آن تام هنکس است. سپس این person را بازمیگردانیم.

کوئری دوم: پیدا کردن فیلم با نام cloud atlas



با استفاده از match به دنبال Movie ای خواهیم بود که لیبل title آن cloud atlas است. سپس این Movie را بازمیگردانیم.

کوئری سوم: پیدا کردن ده عدد آدم

```
neo4j$ MATCH (people:Person) RETURN people.name LIMIT 10
```

	people.name
1	"Keanu Reeves"
2	"Carrie-Anne Moss"
3	"Laurence Fishburne"
4	"Hugo Weaving"
5	"Lilly Wachowski"
6	"Lana Wachowski"
7	"..."

Started streaming 10 records after 12 ms and completed after 13 ms.

در این کوئری ۱۰ نود Person بازگردانده شده است. تعداد ۱۰ با دستور limit مشخص شده است که صرفاً ۱۰ خروجی اول MATCH را ارائه میکند.

کوئری چهارم: پیدا کردن فیلم‌هایی که در دهه ۹۰ عرضه شده اند

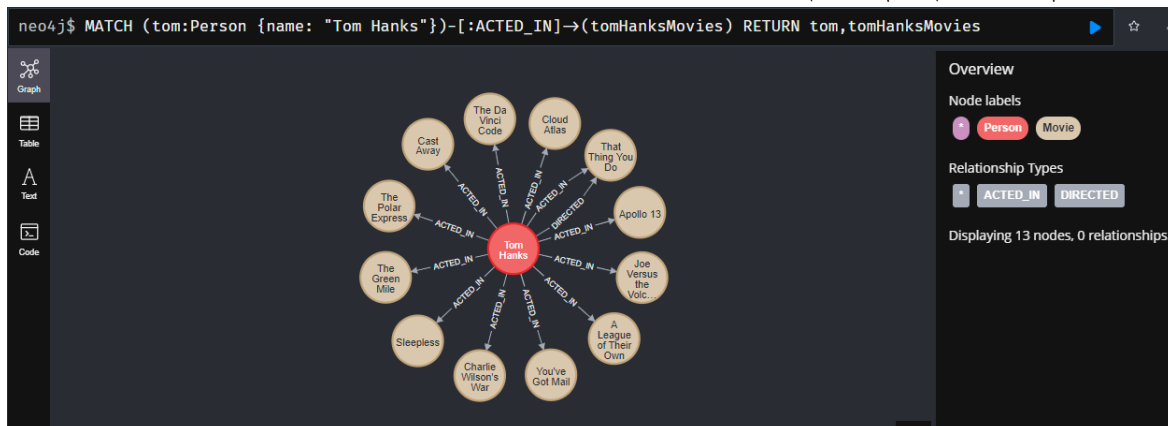
```
1 MATCH (nineties:Movie)
2 WHERE nineties.released > 1990 AND nineties.released < 2000
3 RETURN nineties.title
```

neo4j\$ MATCH (nineties:Movie) WHERE nineties.released > 1990 AND nineties.released < 2000 RETURN nineties.tit...

	nineties.title
1	"The Matrix"
2	"The Devil's Advocate"
3	"A Few Good Men"
4	"As Good as It Gets"
5	"What Dreams May Come"
6	"Snow Falling on Cedars"
7	

خروجی ۱۹ نود است که برخی از آنها را مشاهده میکنید. از بین تمام فیلم‌ها آنهایی که سال عرضه آنها (released) بین ۱۹۹۰ و ۲۰۰۰ است بازگردانده میشود. این شرط در قسمت where تعیین شده است.

کوئری پنجم: یافتن تمام فیلم‌های تام هنکس



تمام نودهایی که نود تام هنکس رابطه ACTED_IN را با آنها دارد برگردانده میشوند. نتیجه به گونه ای بازگردانده میشود تا به صورت گراف قابل مشاهده باشد. توجه کنید برای یافتن تام هنکس از همان کوئری اول استفاده کردیم و در ادامه نودهایی که با آن رابطه ACTED_IN را دارد بازگرداندیم.

کوئری ششم: کارگردان cloud atlas چه کسی است؟

```
1 MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})←[:DIRECTED]-(directors)
2 RETURN directors.name
```

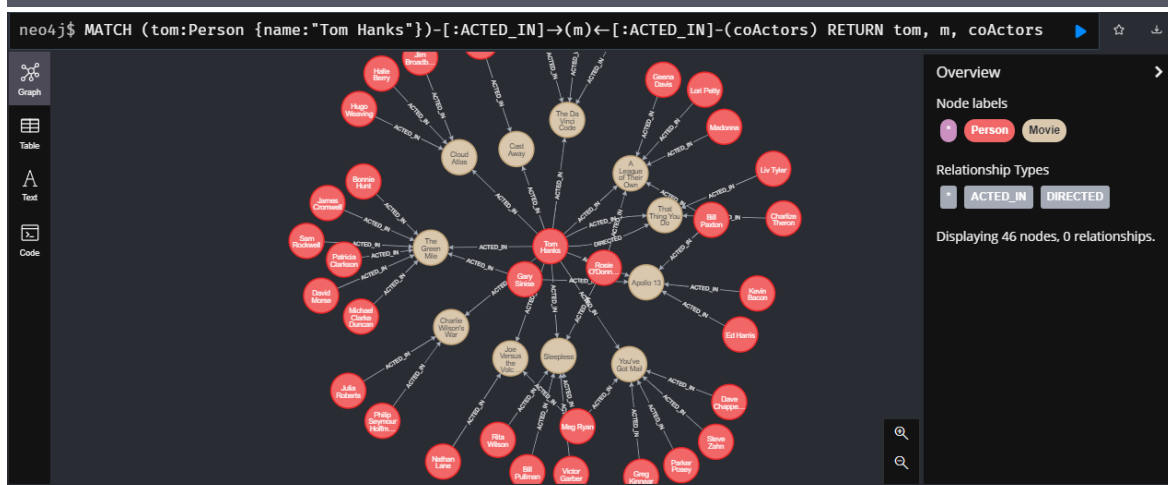
```
neo4j$ MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})←[:DIRECTED]-(directors) RETURN directors.name
```

	directors.name
1	"Tom Tykwer"
2	"Lana Wachowski"
3	"Lilly Wachowski"

نام نودهایی که با نود فیلم cloud atlas رابطه DIRECTED را دارند برگردانده میشود. در اصل کارگردان‌های این فیلم بازگردانده میشوند.

کوئری هفتم: پیدا کردن هم‌بازی‌های تام هنکس

```
1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]→(m)←[:ACTED_IN]-(coActors)
2 RETURN tom, m, coActors
```



ابتدا فیلم‌هایی که تام هنکس در آنها بازی کرده است (رابطه ACTED_IN با تام هنکس دارند) پیدا می‌شود. سپس بازیگرانی که در این فیلم‌ها بازی کرده‌اند (رابطه ACTED_IN با این فیلم دارند) پیدا میشود (coActors) در نهایت نیز خروجی به گونه‌ای بازگردانده میشود تا نتیجه به شکل یک گراف قابل مشاهده باشد.

کوئری هشتم: هر کس چه نقشی در فیلم cloud atlas دارد

```
1 MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
2 RETURN people.name, type(relatedTo), relatedTo
```

people.name	type(relatedTo)	relatedTo
"Tom Hanks"	"ACTED_IN"	{ "identity": 137, "start": 71, "end": 105, "type": "ACTED_IN", "properties": { "roles": ["Zachry", "Dr. Henry Goose", "Isaac Sachs", "Dermot Hoggins"] } }

Started streaming 10 records after 15 ms and completed after 18 ms.

با اجرای این کوئری با قابلیت دستور type آشنا میشویم که نوع رابطه هر نود person با فیلم cloud atlas را نمایش میدهد. (در صورتی که رابطه ای وجود داشته باشد) رابطه بین این دو نود related_to نامیده شده که type آن نمایش داده میشود.

کوئری نهم: یافتن فیلم‌ها و بازیگران با فاصله حداکثر ۳ از کوین بیکن

```
1 MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..3]-(hollywood)
2 RETURN DISTINCT bacon, hollywood
```

Overview

Node labels

- Person
- Movie

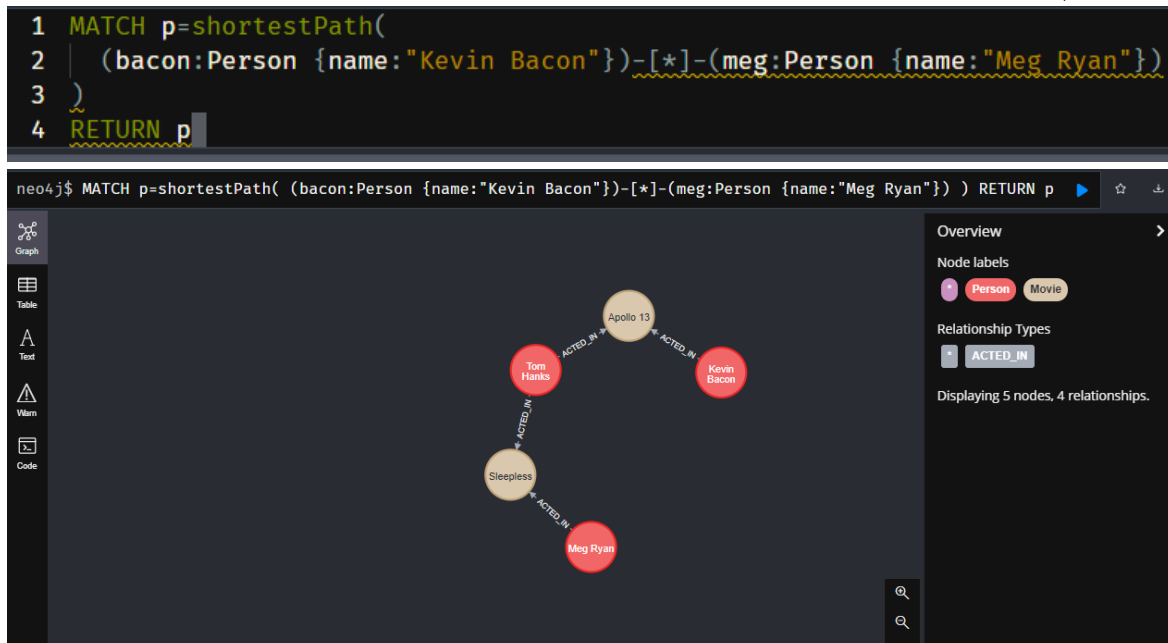
Relationship Types

- ACTED_IN
- DIRECTED
- WROTE
- PRODUCED

Displaying 49 nodes, 0 relationships.

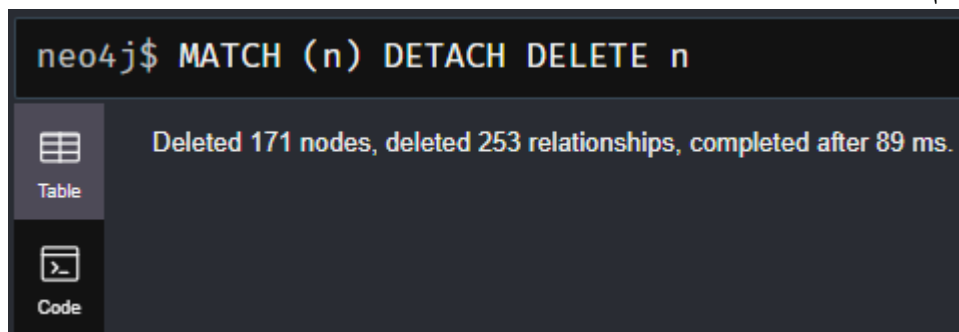
در این کوئری ابتدا کوین بیکن را پیدا کردیم و با استفاده از [*1..3] تمام نودهایی را پیدا کردیم که با کوین بیکن فاصله‌ای بین ۱ تا ۳ دارند. در نهایت نیز از دستور distinct استفاده کردیم تا تکراری نداشته باشیم و به گونه‌ای داده‌ها را برگرداندیم تا به صورت گراف نمایش داده شود.

کوئری دهم: یافتن مسیر بیکن تا مگ رایان



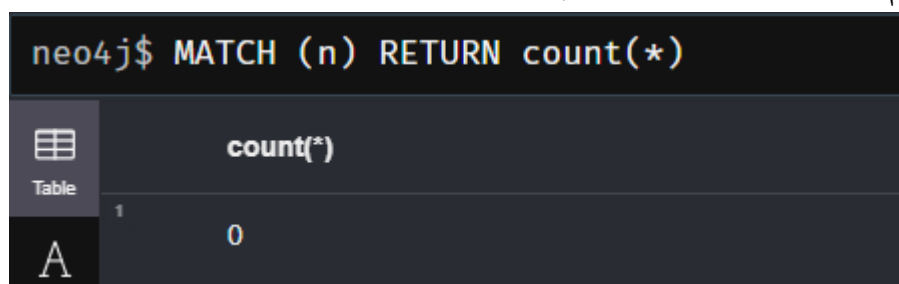
نودهای مربوط به بیکن و رایان پیدا میشود و بعد از آن با استفاده از [*] فواصل بین این دو نود پیدا میشود و بین این فواصل shortestPath گرفته میشود. توجه کنید عملیات [*] میتواند بسیار زمانبر باشد اما گراف مورد بررسی ما کوچک است پس مشکلی پیش نمی‌آید.

کوئری یازدهم: پاکسازی



با استفاده از این دستور تمام نودها و روابط بین آنها پاک میشود.

کوئری دوازدهم: مطمئن شدن از اینکه همه چیز پاک شده است



تعداد نودهای موجود در دیتابیس نمایش داده میشود. مشاهده میشود این مقدار صفر است یعنی تمام نودها به درستی پاک شده‌اند.

بررسی روابط کاراکترهای دنیای سینمایی مارول:

دیتاست مورد نظر با نام marvel-superheroes در سایت Kaggle موجود است. دیتاستی که ما از آن استفاده میکنیم شامل کاراکترها و کامیک‌بوک‌ها و اتفاقات و توانایی‌ها و ... است که روی این مسائل تحلیل‌های مختلفی انجام می‌دهیم. ابتدا با کدی که در سایت موجود است دیتاست را لود میکنیم:

```
1 CALL apoc.schema.assert({Character:['name']},{Comic:['id'], Character:['id'], Event:['id'], Group:
  ['id']});
2
3 LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-
  datasets/main/Marvel/heroes.csv" as row
4 CREATE (c:Character)
5 SET c += row;
6
7 LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-
  datasets/main/Marvel/groups.csv" as row
8 CREATE (c:Group)
9 SET c += row;
```

که نتیجه آن به شکل زیر است:

```
neo4j$ CALL apoc.schema.assert({Character:['name']},{Comic:['id'], Character:['id'], Event:['id'], Group...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blog-datasets/main/Marvel...
neo4j$ MATCH (s:Stats) WITH keys(s) as keys LIMIT 1 MATCH (s:Stats) UNWIND keys as key CALL apoc.create...
```

در هنگام اجرای این مرحله نیاز به دانلود پلاگین apoc بود و قبل از دانلود کردن آن به ارور برمیخوردم. پس از دانلود آن و قرار دادن در پوشه plugins، نیاز بود در فایل کانفیک neo4j.conf تغییراتی اعمال کنم که به شکل زیر است:

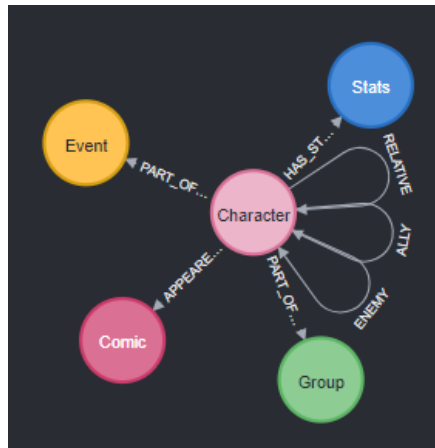
```
dbms.directories.plugins=c:/neo4j/plugins
dbms.security.procedures.unrestricted=apoc.*
dbms.security.procedures.whitelist=apoc.*
```

در اینجا برخی تنظیمات امنیتی و همچنین آدرس پوشه پلاگین‌ها قرار داده شده است. بعد از آن با ری استارت کردن neo4j توانستم از توابع مورد نظر برای لود کردن استفاده کنم.

حال با استفاده از دستور زیر به بررسی این دیتاست می‌پردازیم:

```
neo4j$ CALL db.schema.visualization()
```

نتیجه آن به شکل زیر است:



همانطور که مشاهده میشود کاراکترها وجود دارند که میتوانند در چندین کامیک بوک (comic) حضور داشته باشند، جزئی از یک اتفاق (event) باشند یا به یک گروه (group) تعلق داشته باشند. برای برخی کاراکترها معیارهایی مانند سرعت یا توانایی مبارزه نگه داشته میشود. همچنین بین کاراکترهای مختلف سه نوع ارتباط وجود دارد: مرتبط (relative)، هم‌پیمان (ally) یا دشمن (enemy)

برای اینکه نسبت به سائز گراف حس پیدا کنیم دستور زیر را اجرا میکنیم:

```
neo4j$ CALL apoc.meta.stats() YIELD labels return labels
```

که خروجی آن به شکل زیر است:

```
{
  "Stats": 470,
  "Group": 92,
  "Event": 74,
  "Character": 1105,
  "Comic": 38875
}
```

میتوانیم ببینیم به ازای هر نوع نود چه تعداد از آن وجود دارد. مثلاً ۱۱۰۵ کاراکتر داریم.

حال به بررسی مقادیر نودها میپردازیم تا با گراف آشنایی بیشتری پیدا کنیم. برای شروع از کوئری زیر کمک میگیریم تا بیشترین حضور کاراکترها در کامیک بوک‌ها را بیابیم:

```
MATCH (c:Character)
RETURN c.name as character,
       size((c)-[:APPEARED_IN]->()) as comics
ORDER BY comics DESC
LIMIT 5
```

همانطور که مشاهده میشود به ازای هر کاراکتر، تعداد روابط APPEARED_IN که یک نود آن کاراکتر مورد نظر است شمرده میشود. در نهایت این مقادیر به صورت نزولی مرتب شده و پنج مقدار اول آن بازگردانده میشود. نتیجه این کوئری به شکل زیر است:

character	comics
1 "Spider-Man (1602)"	3357
2 "Tony Stark"	2354
3 "Logan"	2098
4 "Steve Rogers"	2019
5 "Thor (Marvel: Avengers Alliance)"	1547

به نظر اسپایدرمن از همه محبوب تر است. بعد از او آبرون من و ولورین بیشترین حضور را در کامیک بوک ها دارند.

حال اندکی به بررسی event ها میپردازیم. با استفاده از کوئری زیر، event هایی که بیشترین تعداد کاراکترها در آن حضور داشتند را میبینیم:

```
MATCH (e:Event)
RETURN e.title as event,
       size((e)-[:PART_OF_EVENT]-()) as count_of_heroes,
       e.start as start,
       e.end as end,
       e.description as description
ORDER BY count_of_heroes DESC
LIMIT 5
```

در این کوئری به ازای هر event، تعداد روابط PART_OF_EVENT که نود event آن همان event مورد نظر است شمرده میشود. علاوه بر این تعداد زمان شروع و زمان پایان و توضیحات آن نیز برگردانده میشود و بر اساس تعداد قهرمانان به صورت نزولی مرتب میشود و پنج مقدار اول آن نمایش داده میشود. نتیجه به شکل زیر خواهد بود (توضیحات به دلیل طولانی بودن به خوبی نیفتاده است):

event	count_of_heroes	start	end	description
"Fear Itself"	132	"2011-04-16 00:00:00"	"2011-10-18 00:00:00"	"The Serpent, Go
"Dark Reign"	128	"2008-12-01 00:00:00"	"2009-12-31 12:59:00"	"Norman Osborn
"Acts of Vengeance!"	93	"1989-12-10 00:00:00"	"2008-01-04 00:00:00"	"Loki sets about c
"Secret Invasion"	89	"2008-06-02 00:00:00"	"2009-01-25 00:00:00"	"The shape-shifter
"Civil War"	86	"2006-07-01 00:00:00"	"2007-01-29 00:00:00"	"After a horrific tr

از آنجایی که من بیشتر فیلم های کامیک بوکی دیدم تا خود کامیک بوک ها را بخوانم فقط مورد آخر را میشناسم!

حال به بررسی بزرگترین گروه‌ها میپردازیم:

```
MATCH (g:Group)
RETURN g.name as group,
       size((g)←[:PART_OF_GROUP]-()) as members
ORDER BY members DESC LIMIT 5
```

به ازای هر گروه تعداد روابط PART_OF_GROUP که یک سر آن گروه مورد نظر است را می‌شماریم. با این کار به ازای هر گروه تعداد کاراکترهای آن را می‌یابیم. سپس این اطلاعات را بر اساس تعداد کاراکتر به صورت نزولی مرتب می‌کنیم و پنج مقدار اول را بازگردانیم. نتیجه به شکل زیر میشود:

group	members
"X-Men"	41
"Avengers"	31
"Defenders"	26
"Next Avengers"	14
"Guardians of the Galaxy"	12

همانطور که مشاهده میشود گروه مردان ایکس بیشترین عضو را دارد و بعد از آن avengers قرار دارد.

حال میخواهیم ببینیم کدام کاراکترها در عین اینکه در یک تیم قرار دارند با یکدیگر دشمن نیز هستند:

```
MATCH (c1:Character)-[:PART_OF_GROUP]→(g:Group)←[:PART_OF_GROUP]-(c2:Character)
WHERE (c1)-[:ENEMY]-(c2) and id(c1) < id(c2)
RETURN c1.name as character1, c2.name as character2, g.name as group
```

در قسمت MATCH به ازای هر کاراکتر که عضو گروهی است سایر اعضای آن گروه را می‌یابیم. در قسمت WHERE چک میکنیم آیا این دو کاراکتر دشمن هستند یا خیر. شرط id برای این است که رکورد مورد نظر فقط یکبار بیاید (یعنی اگر دو کاراکتر این شرایط را داشته باشند یکبار c1 کاراکتر اول میشود و یکبار کاراکتر دوم پس در جدول نهایی این داده دو بار می‌آید) در نهایت نیز نام دو کاراکتر به همراه گروه آن بازگردانده میشود.

character1	character2	group
"Logan"	"Sabretooth (House of M)"	"X-Men"
"Logan"	"Mystique (House of M)"	"X-Men"
"CAIN MARKO JUGGERNAUT"	"Logan"	"X-Men"
"CAIN MARKO JUGGERNAUT"	"Storm (Marvel Heroes)"	"X-Men"
"Rogue (X-Men: Battle of the Atom)"	"Warren Worthington III"	"X-Men"

ظاهراً ولورین با هم تیمی هایش مشکل دارد ((

در ادامه چون دیدیم یک کاراکتر از یوگسلاوی است میخواهیم ببینیم آیا کاراکترهای دیگری از یوگسلاوی وجود دارند یا خیر:

```
MATCH (c:Character)
WHERE c.place_of_origin contains "Yugoslavia"
RETURN c.name as character,
       c.place_of_origin as place_of_origin,
       c.aliases as aliases
```

به ازای هر کاراکتر با استفاده از دستور contains چک میکنیم که آیا کلمه Yugoslavia جزو محل تولد آن کاراکتر هست یا خیر. اگر بود مقادیر نام و محل تولد و نام های مستعار آن کاراکتر را بازمیگردانیم:

character	place_of_origin	aliases
"Purple Man"	"Rijeka, Yugoslavia"	"Killgrave the Purple Man, Killy"
"Abomination (Ultimate)"	"Zagreb, Yugoslavia"	"Agent R-7, the Ravager of Worlds"

حال به بررسی کاراکترهایی که دکترا دارند میپردازیم:

```
1 MATCH (c:Character)
2 WHERE c.education contains "Ph.D"
3 RETURN c.name as character, c.education as education
4 LIMIT 10
5
```

به ازای هر کاراکتر با استفاده از دستور contains چک میکنیم که آیا کلمه Ph.D جزو تحصیلات آنها هست یا خیر. اگر بود مقادیر نام و تحصیلات آن را برمیگردانیم. توجه کنید ۱۰ مقدار اول را نمایش میدهیم:

character	education
"UNKNOWN ACHEBE"	"Ph.D. in Law (Yale), degrees in Psychology, Political Science and Divinity"
"PROFESSOR MENDEL STROMM MENDEL STROMM"	"Ph.D. in robotics"
"FRANKLIN HALL GRAVITON"	"Ph.D. in physics"
"Morbis"	"Ph.D in Biochemistry"
"Tony Stark"	"Ph.Ds in physics and electrical engineering"
"Hulk-dok"	"Ph.D in nuclear physics and two other fields"

حال اندکی به بررسی روابط میپردازیم. در این قسمت میخواهیم بدانیم هر کاراکتر با چه تعداد از کاراکترهای دیگر رابطه دوستی یا دشمنی یا رابطه خانوادگی دارد. از کوئری زیر استفاده میکنیم:

```
MATCH (c:Character)
RETURN c.name as name,
       size((c)-[:ALLY]→()) as allies,
       size((c)-[:ENEMY]→()) as enemies,
       size((c)-[:RELATIVE]→()) as relative
ORDER BY allies + enemies + relative DESC
LIMIT 5
```

به ازای هر کاراکتر سه مقدار بازمیگردانیم:

- تعداد روابط ALLY که یک سر آن کاراکتر مورد نظر است را تحت عنوان allies
- تعداد روابط ENEMY که یک سر آن کاراکتر مورد نظر است را تحت عنوان enemy
- تعداد روابط RELATIVE که یک سر آن کاراکتر مورد نظر است را تحت عنوان relative

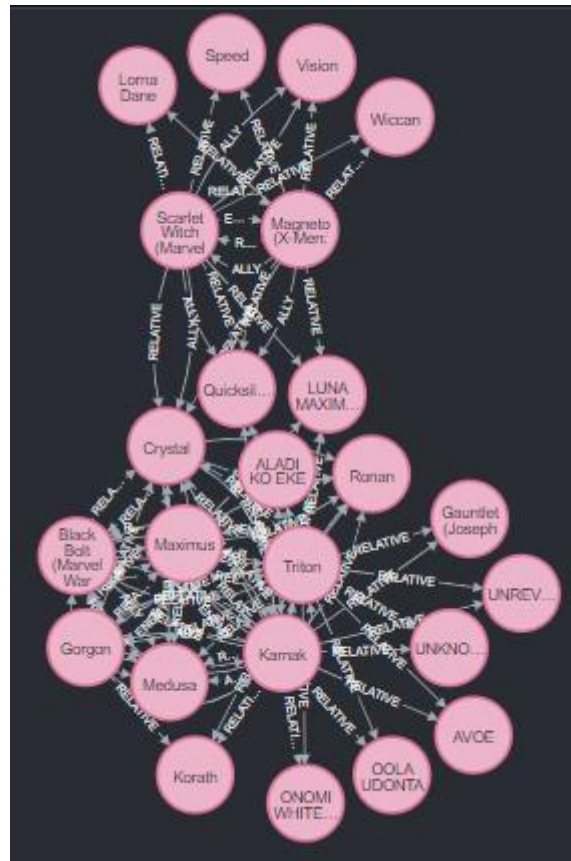
بر اساس مجموع این مقادیر داده ها را به صورت نزولی مرتب میکنیم پنج مقدار اول را نشان میدهیم:

name	allies	enemies	relative
"Scarlet Witch (Marvel Heroes)"	16	14	8
"Thor (Marvel: Avengers Alliance)"	9	14	10
"Invisible Woman (Marvel: Avengers Alliance)"	13	10	7
"Logan"	14	10	5
"Karnak"	6	2	17

اسکارلت ویچ نسبت به سایر کاراکترها روابط بیشتری دارد. همچنین اسکارلت ویچ و ثور از همه بیشتر دشمن دارند. تریتون هم از همه بیشتر فامیل دارد. برای اینکه انجمن خانوادگی تریتون را ببینیم از تابع زیر استفاده میکنیم:

```
1 MATCH p=(c:Character{name:"Triton"})
2 CALL apoc.path.subgraphAll(id(c), {relationshipFilter:"RELATIVE"})
3 YIELD nodes, relationships
4 RETURN nodes, relationships
```

به ازای کاراکتر تریتون زیرگرافی که شامل تمام روابط RELATIVE با این کاراکتر هست را پیدا میکنیم (از طریق صدا کردن تابع subgraphAll) و نتیجه را در nodes و relationships میریزیم و این دو مقدار را بازمیگردانیم. نتیجه به صورت زیر است:



پس توانستیم گراف خانوادگی تربیتیون را ببینیم.

حال الگوریتم Weakly Connected Components algorithm را اجرا میکنیم تا بزرگترین کامپوننت هایی را پیدا کنیم که در آن گراف رابطه دوستی وجود دارد (یافتن جزایر در گراف):

```
1 CALL gds.wcc.stream({
2   nodeProjection:'Character',
3   relationshipProjection:'ALLY'})
4 YIELD nodeId, componentId
5 WITH componentId, count(*) as members
6 WHERE members > 1
7 RETURN componentId, members
8 ORDER BY members DESC
9 LIMIT 5
```

از تابع `gds.wcc.stream` استفاده میکنیم تا projection روی رابطه `ALLY` کاراکترها انجام دهیم و از این تابع اینکه هر نود در کدام کامپوننت است را دریافت میکنیم. سپس به ازای هر کامپوننت تعداد نودهای آن را می‌شماریم. این تعداد باید از یک بزرگتر باشد. در نهایت آیدی کامپوننت به همراه تعداد نودها را بازمیگردانیم. خروجی را بر اساس تعداد نود مرتب کرده و پنج مقدار اول را نمایش میدهیم:

members	componentid
195	0
4	26
3	245
2	6
2	2

همانطور که مشاهده میشود بیشترین تعداد دوست بودن برای کامپوننت با آیدی صفر است که ۱۹۵ عضو دارد.

حال به بررسی توانایی های کاراکترها میپردازیم. با استفاده از کوئری زیر بیشترین میانگین stats های کاراکترها را به دست می آوریم:

```
1 MATCH (c:Character)-[:HAS_STATS]-(stats)
2 RETURN c.name as character,
3        apoc.coll.avg(apoc.map.values(stats, keys(stats))) as average_stats
4 ORDER BY average_stats DESC
5 LIMIT 10
```

به ازای هر کاراکتر stats های مربوط به آن (نودهایی که کاراکتر با آنها رابطه HAS_STATS دارد) را به دست می آوریم. با استفاده از تابع apoc.map_values این مقادیر را در کنار یکدیگر قرار میدهیم و با استفاده از تابع apoc.coll.avg این مقادیر کنار هم قرار داده شده میانگین میگیریم. نام کاراکتر و میانگین به دست آمده را باز میگردانیم و بر اساس میانگین به صورت نزولی مرتب میکنیم و ده مقدار اول را نمایش میدهیم:

character	average_stats
"Sasquatch (Walter Langkowski)"	7.0
"Squirrel Girl"	7.0
"Galactus"	7.0
"Deathstrike (Ultimate)"	7.0
"GRAYDON CREED"	7.0
"CHTHON"	7.0

برخی نتایج به دست آمده را مشاهده میکنید. همانطور که مشاهده میشود مقادیر به دست آمده بیشترین میزانی است که یک کاراکتر میتواند داشته باشد.

حال می‌خواهیم الگوریتم knn را پیاده سازی کنیم. ابتدا می‌خواهیم به ازای هر نود کاراکتر یک وکتور درون آن نود داشته باشیم تا stat‌های آن کاراکتر همراه آن ذخیره شود. برای این کار از دستور زیر استفاده می‌کنیم:

```
MATCH (c:Character)-[:HAS_STATS]-(s)
WITH c, [s.durability, s.energy, s.fighting_skills,
         s.intelligence, s.speed, s.strength,
         CASE WHEN c.flight = 'true' THEN 7 ELSE 0 END] as stats_vector
SET c.stats_vector = stats_vector
```

در قسمت MATCH به ازای هر کاراکتر تمام نودهایی که با آن کاراکتر رابطه HAS_STATS دارند را به دست می‌آوریم. در قسمت WITH این stat‌ها را (که اعداد بین ۰ تا ۷ هستند) در یک وکتور با نام stats_vector میریزیم. توجه کنید قابلیت پرواز امری صفر و یکی است و اگر کاراکتری قابلیت پرواز داشت به او مقدار ۷ و در غیر این صورت مقدار صفر نسبت داده میشود. در نهایت نیز در قسمت SET یک لیبل به کاراکتر اضافه میشود که این وکتور در آن قرار میگیرد. خروجی آن به شکل زیر است:

Set 470 properties, completed after 354 ms.

در گام بعد می‌خواهیم هر نود کاراکتری که stats_vector را دارد تگ کنیم. برای این کار از دستور زیر استفاده می‌کنیم:

```
MATCH (c:Character)
WHERE exists(c.stats_vector)
SET c:CharacterStats
```

به ازای هر کاراکتر با استفاده از دستور exists چک می‌کنیم که آیا stats_vector را دارد یا خیر. اگر داشت یک لیبل به نام CharacterStats در کاراکتر مورد نظر SET می‌کنیم. خروجی این قسمت به شکل زیر است:

Added 470 labels, completed after 107 ms.

در گام بعد یک گراف برای اجرای الگوریتم تولید می‌کنیم:

```
CALL gds.graph.create('marvel', 'CharacterStats',
                     '*', {nodeProperties:'stats_vector'})
```

با صدا کردن تابع gds.graph.create یک گراف تولید می‌کنیم. نام آن را marvel، node projection آن را characterStats (که در قسمت قبل تولید کردیم)، relation projection آن را تمام روابط و از configuration آن مقدار nodeProperties را برابر state_vector می‌گذاریم. خروجی به شکل زیر است:

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre>{ "CharacterStats": { "properties": { "stats_vector": { "property": "stats_vector", "defaultValue": null } }, "label": "CharacterStats" } }</pre>	<pre>{ "_ALL_": { "orientation": "NATURAL", "aggregation": "DEFAULT", "type": "*", "properties": { } } }</pre>	"marvel"	470	515	169

در نهایت تنها کافی است تابع مربوط به اجرای الگوریتم را صدا کنیم.

```
CALL gds.beta.knn.mutate('marvel', {nodeWeightProperty:'stats_vector', sampleRate:0.8, topK:15, mutateProperty:'score', mutateRelationshipType:'SIMILAR'})
```

تابع مربوطه `gds.beta.knn.mutate` نام دارد و روی گرافی که تولید کردیم اجرا میشود. هاپیر پارامترهایی که برای اجرای الگوریتم تعیین کردیم مشخص است و بقیه پارامترها به صورت پیش فرض تعیین شده است. خروجی این تابع اطلاعات مفیدی به ما نشان نمیدهد فلذا از قرار دادن خروجی آن صرف نظر کردیم. توجه کنید استفاده از `mutate` نتیجه را روی گرافی از قبل تعیین شده قرار میدهد.

حال میخواهیم الگوریتم Louvain Modularity را اجرا کنیم.

```
CALL gds.louvain.write('marvel',
  {relationshipTypes:['SIMILAR'],
   relationshipWeightProperty:'score',
   writeProperty:'louvain'});
```

تابع `gds.louvain.write` را صدا میکنیم تا الگوریتم را اجرا کند. (روی همان گراف `marvel` که قبلاً ساختیم) توجه کنید از `write` استفاده کردیم تا داده ها روی همان گراف نوشته شوند. سایر پارامترهای این الگوریتم هم قابل مشاهده است. خروجی این تابع اطلاعات مفیدی به ما نشان نمیدهد فلذا از قرار دادن خروجی آن صرف نظر کردیم.

حال به دنبال مشاهده نتیجه ها خواهیم رفت.

```
MATCH (c:Character)-[:HAS_STATS]-(stats)
RETURN c.louvain as community, count(*) as members,
  avg(stats.fighting_skills) as fighting_skills,
  avg(stats.durability) as durability,
  avg(stats.energy) as energy,
  avg(stats.intelligence) as intelligence,
  avg(stats.speed) as speed,
  avg(stats.strength) as strength,
  avg(CASE WHEN c.flight = 'true' THEN 7.0 ELSE 0.0 END) as flight
```

توابع قبل یک `label` با نام `Louvain` به کاراکتر اضافه کردند. این لیبل اضافه شده به همراه تعداد کاراکترهایی که این لیبل را دارند به همراه میانگین `stats`های کاراکترهایی که این لیبل را دارند بازگردانده میشوند. نتیجه به شکل زیر است:

"community"	"members"	"fighting_skill ls"	"durability"	"energy"	"intelligence"	"speed"	"strength"	"flight"
348	73	4.575342465753 426	5.890410958904 109	5.315068493150 682	4.342465753424 657	5.506849315068 493	5.191780821917 808	2.109589041095 8913
334	42	2.238095238095 238	2.023809523809 5237	0.880952380952 3807	3.023809523809 5237	1.809523809523 8093	1.761904761904 7619	0.166666666666 66669
393	95	3.473684210526 317	2.863157894736 8417	2.273684210526 3158	2.915789473684 2094	2.473684210526 315	2.526315789473 684	0.663157894736 8424
131	15	4.866666666666 667	5.266666666666 667	4.399999999999 9995	4.533333333333 333	4.4	4.666666666666 667	0.0
323	30	3.899999999999 9995	4.699999999999 999	3.099999999999 9996	3.4	2.566666666666 6673	4.266666666666 666	0.233333333333 33342
132	50	3.999999999999 999	3.280000000000 0002	2.460000000000 0004	3.019999999999 9996	3.14	3.260000000000 0002	0.42
298	66	3.924242424242 424	3.909090909090 9087	3.090909090909 09	3.090909090909 0917	3.318181818181 8183	3.803030303030 303	1.060606060606 0608

البته این عکس فقط بخشی از خروجی است و در ادامه چند کامیونیتی دیگر نیز وجود دارند.

حال الگوریتم Label Propagation را اجرا میکنیم.

```
CALL gds.labelPropagation.write('marvel',
{relationshipTypes:['SIMILAR'],
relationshipWeightProperty:'score',
writeProperty:'labelPropagation'})
```

تابع `gds.labelPropagation.write` را صدا میکنیم تا الگوریتم را اجرا کند. (روی همان گراف `marvel` که قبلا ساختیم) توجه کنید از `write` استفاده کردیم تا داده ها روی همان گراف نوشته شوند. سایر پارامترهای این الگوریتم هم قابل مشاهده است. خروجی این تابع اطلاعات مفیدی به ما نشان نمیدهد فلذا از قرار دادن خروجی آن صرف نظر کردم. حال به دنبال مشاهده نتیجه ها خواهیم رفت.

```
MATCH (c:Character)-[:HAS_STATS]→(stats)
RETURN c.labelPropagation as community, count(*) as members,
avg(stats.fighting_skills) as fighting_skills,
avg(stats.durability) as durability,
avg(stats.energy) as energy,
avg(stats.intelligence) as intelligence,
avg(stats.speed) as speed,
avg(stats.strength) as strength,
avg(CASE WHEN c.flight = 'true' THEN 7.0 ELSE 0.0 END) as flight
```

توابع قبل یک `label` با نام `labelPropagation` به کاراکتر اضافه کردند. این لیبل اضافه شده به همراه تعداد کاراکترهایی که این لیبل را دارند به همراه میانگین `stats`های کاراکترهایی که این لیبل را دارند بازگردانده میشوند. نتیجه به شکل زیر است:

"community"	"members"	"fighting_skill"	"durability"	"energy"	"intelligence"	"speed"	"strength"	"flight"
444	9	4.333333333333333	5.555555555555555	4.666666666666666	4.444444444444444	5.333333333333333	4.777777777777777	0.0
602	19	1.368421052631579	1.6842105263157894	0.7368421052631578	2.6315789473684217	1.6842105263157892	1.4736842105263156	0.368421052631579
118	112	3.9553571428571423	3.8125	2.5089285714285707	2.964285714285714	3.062500000000000	3.3839285714285716	1.0625
270	15	4.866666666666666	5.266666666666666	4.399999999999999	4.533333333333333	4.4	4.666666666666666	0.0
215	16	4.5	3.499999999999999	2.374999999999999	3.3125	3.125	4.125	0.0
218	58	4.172413793103449	4.568965517241379	3.896551724137932	3.827586206896551	3.775862068965518	4.1206896551724155	0.8448275862068966
675	6	2.333333333333333	1.166666666666666	1.833333333333333	3.333333333333333	0.333333333333333	1.0	0.0

البته این عکس فقط بخشی از خروجی است و در ادامه چند کامیونیتی دیگر نیز وجود دارند.

مشکلات و توضیحات تکمیلی

پروژه‌ای بسیار عالی همراه مباحث جدید و جذاب بود. البته در انتها کمی خسته کننده شده بود زیرا از مباحث دیتابیس به سمت مباحث دیتا رفته بود اما در کل پروژه ای عالی بود.

آنچه آموختم / پیشنهادات

به عنوان کسی که قبلاً با دیتابیس‌های گراف‌ی کار نکرده بودم تجربه‌ای بسیار عالی بود. از نظر من اگر صورت پروژه به گونه‌ای بود که مجبور شویم خودمان چند کوئری ساده بنویسیم بهتر میشد. همچنین برخی دستورهای مورد استفاده deprecated شده بود (در تصاویر مربوطه زیر آنها خط کشیده شده است) شاید بهتر باشد از دستورهای جدید استفاده کنیم.