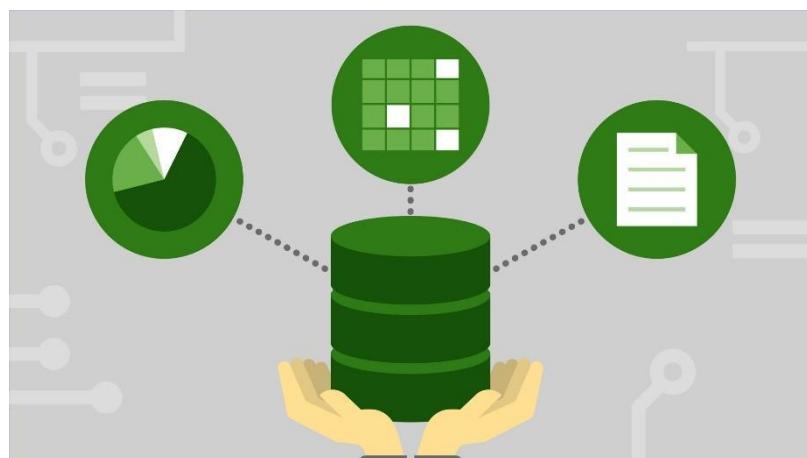


به نام خدا



دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر



آزمایشگاه پایگاه داده

دستور کار شماره 3

حمیدرضا خدادادی

810197499

آبان ماه ۱۴۰۰

گزارش دستور کار انجام شده

در این دستورکار با دیتابیس گراف محور Neo4j کار خواهیم کرد تا با رویکرد گراف محور به ذخیره و بازیابی داده ها بپردازیم. این دستور کار از دو بخش تشکیل شده است:

1. آشنایی با نحوه کار با Neo4j و زبان Cypher
2. کار بر روی یک دیتاست واقعی و انجام چند کوئری

● بخش اول: آموزش سایت Quackit

در زبان Neo4j Query ساختار یافته مختص به خود را دارد که Cypher نامیده می شود. این زبان از سینتکس مشابه با SQL استفاده می کند.

در زبان SQL ما جدول ها و ردیف ها و ستون ها را جستجو می کنیم ولی در Neo4j ما گره ها و برچسب های آن ها و ویژگی های آن ها را جستجو می کنیم. پس Neo4j یک دیتابیس NoSQL است.

برای ساخت یک node یا relationship که به ترتیب همان گره و یال گراف ما خواهد بود از دستور CREATE استفاده خواهیم کرد.

در ابتدا می خواهیم یک پایگاه داده موسیقی بسازیم که نام گروه های موسیقی و آلبوم های آن ها باشد.

با دستور زیر یک گره ایجاد می کنیم که گروه موسیقی اول را می سازد.



The screenshot shows the Neo4j Browser interface. In the top-left corner, there's a sidebar with three icons: a star, a table, and a play button. The main area has a command line at the top with the text "neo4j\$ CREATE (a:Artist { Name : "Strapping Young Ld" })". Below the command line, a message says "Added 1 label, created 1 node, set 1 property, completed after 31 ms." On the right side, there are several tabs: "Table", "Code", "Graph", and "Logs". The "Graph" tab is currently selected, showing a small visualization of the node creation.

راس ها یا گره های گراف می توانند یک label داشته باشند که بدین صورت از راس ها یا گره های دیگر گراف متمایز شوند. در دستور بالا یک متغیر a که گره ما است، ایجاد شده است. این گره label ای برابر با Artist دارد. همچنین گره های گراف می توانند یک سری ویژگی هایی را شامل شوند. برای مثال گره بالا،

دارای یک ویژگی به اسم Name است که این ویژگی Value ای برابر با "Strapping Young Ld" گرفته است.

برای نمایش یگ گره از گراف می توانیم از دستور RETURN استفاده کنیم.

در ادامه ما می خواهیم یک گره دیگر به نام b که label ای برابر با Album دارد را ایجاد کنیم. این گره جدید دو ویژگی Released و Name را با سینتکس بالا نشان خواهیم داد.

```

1 CREATE (b:Album { Name : "Heavy as a Really Heavy Thing", Released : "1995" })
2 RETURN b
  
```

```

1 CREATE (b:Album { Name : "Heavy as a Really Heavy Thing", Released : "1995" })
2 RETURN b
  
```

```

b
  1
    {
      "identity": 1,
      "labels": [
        "Album"
      ],
      "properties": {
        "Released": "1995",
        "Name": "Heavy as a Really Heavy Thing"
      }
    }
  
```

اگر بخواهیم چند گره را به صورت همزمان ایجاد کنیم، آن ها را در سینتکس CREATE با "," از یکدیگر جدا می کنیم.

```

1 CREATE (a:Album { Name: "Killers"}), (b:Album { Name: "Fear of the Dark"})
2 RETURN a,b
  
```

حال ما می خواهیم بین گره ها، یال ایجاد کنیم تا ارتباط بین گره ها را بیان کنیم.

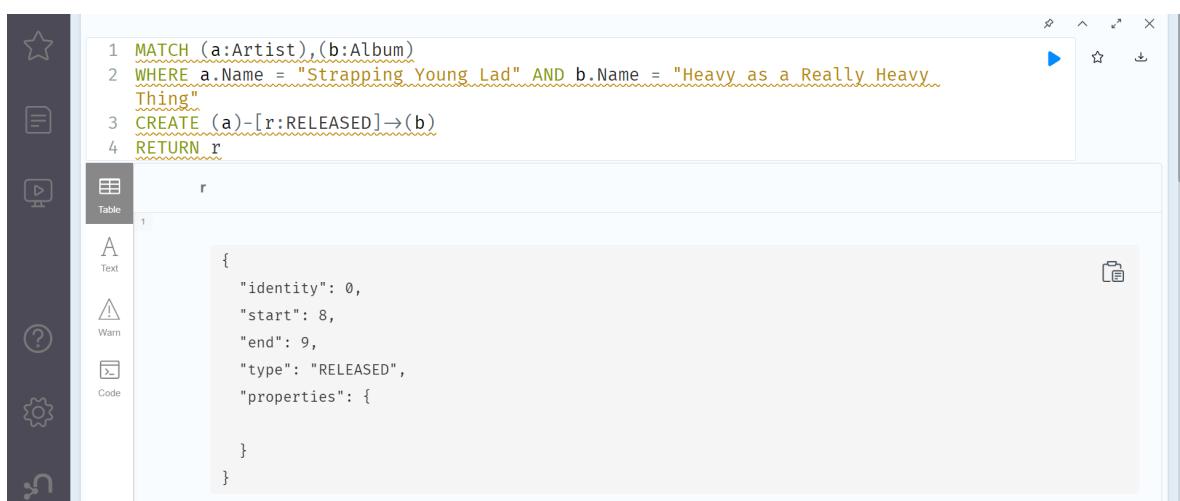
در ادامه بین خواننده و آلبومی که در بالا ساخته شد، یک یال ایجاد می کنیم.

در ابتدا دو گره ای که می خواهیم بین آن ها یال ایجاد کنیم را با سینتکس MATCH مشخص می کنیم. از آن جایی که ممکن است تعداد زیادی گره با label های مختلف داشته باشیم، باید گره های دلخواه خود را فیلتر کیم. ما برای این کار از سینتکس WHERE SQL استفاده می کنیم.

سپس با سینتکس CREATE یال خود را بین دو گره ای که فیلتر کردیم ایجاد می کنیم.

(a)-[r:RELEASED]->(b)

طبق سینتکس خط بالا ما چگونگی ارتباط بین دو گره را مشخص می کنیم. در پرانتز اول گره ای می آید که یال از آن خارج شده و در پرانتز بعد از فلش، گره ای که یال به آن وارد شده است می آید. همچنین داخل کروشه و پس از نام متغیر متناظر با آن یال، label آن یال می آید.



The screenshot shows the Neo4j Studio interface. On the left, there's a sidebar with icons for star, list, and code. The main area has a code editor with the following Cypher query:

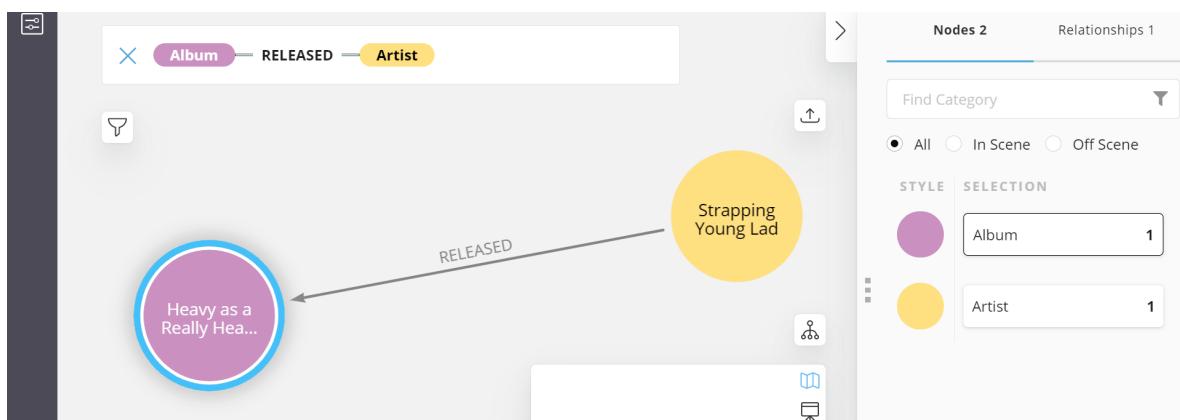
```

1 MATCH (a:Artist),(b:Album)
2 WHERE a.Name = "Strapping Young Lad" AND b.Name = "Heavy as a Really Heavy Thing"
3 CREATE (a)-[r:RELEASED]-(b)
4 RETURN r
    
```

Below the code editor is a results panel titled 'Table' showing one row with the label 'r'. The row contains a JSON object representing the relationship:

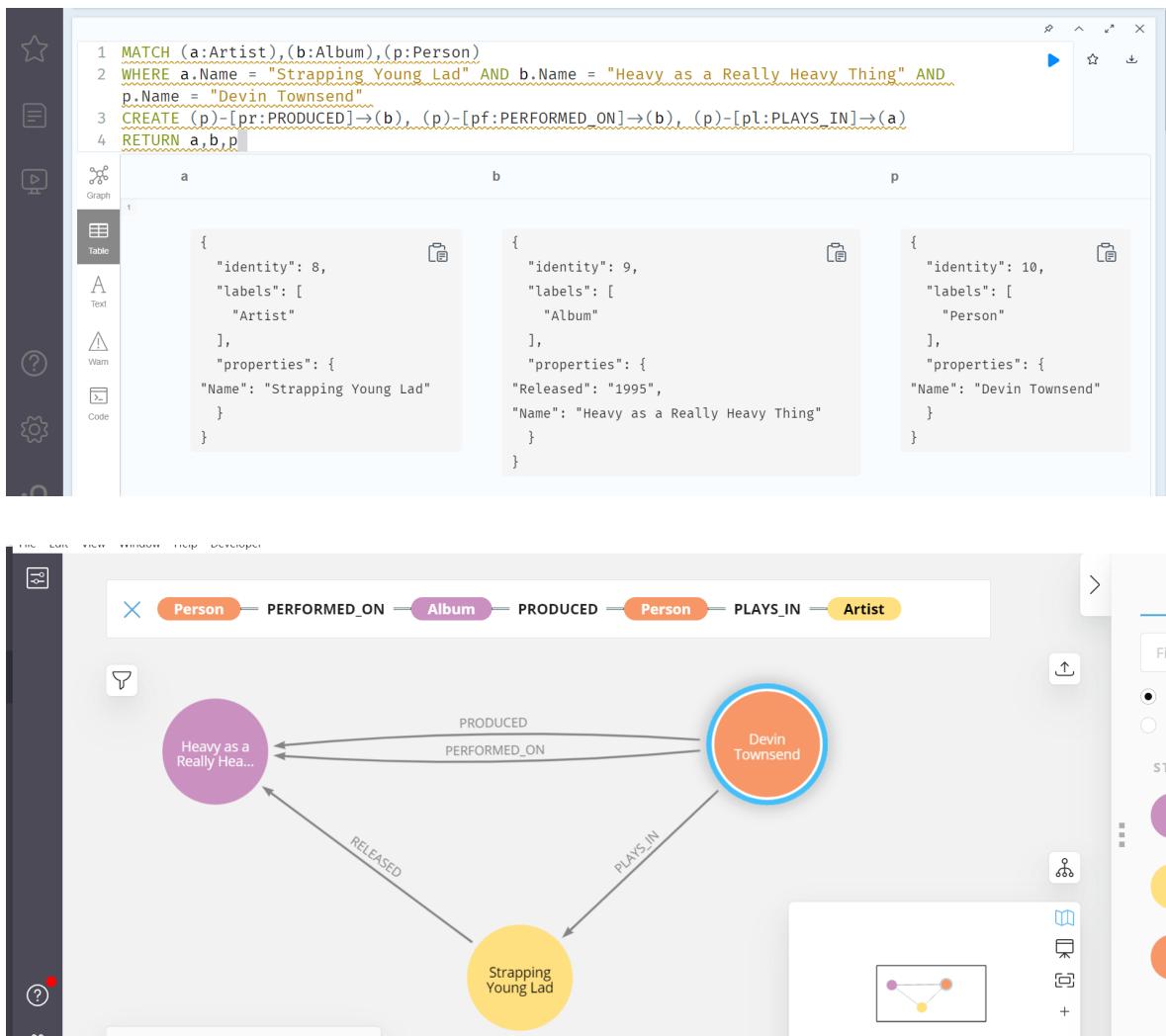
```

{
  "identity": 0,
  "start": 8,
  "end": 9,
  "type": "RELEASED",
  "properties": {}
}
    
```



حال اگر بخواهیم روابط چندگانه بین گره های مختلف را ببینیم مانند قبل عمل می کنیم.

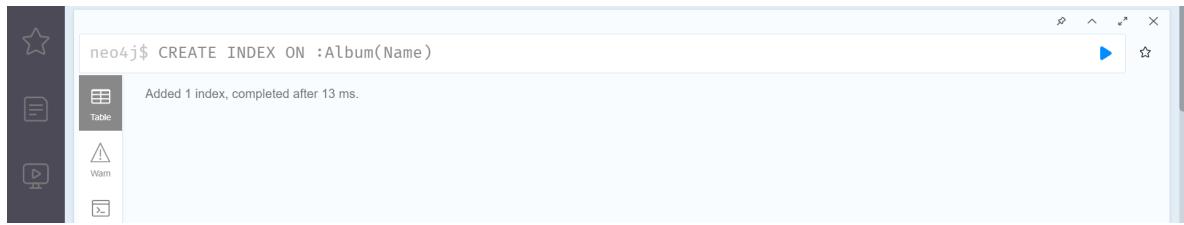
در ابتدا یک گره جدید به نام Person با ویژگی Name ایجاد می کنیم.



در ادامه به ساخت ایندکس روی داده ها می پردازیم تا سرعت عملیات روی داده ها افزایش یابد.

ما می توانیم روی هر گره ای که label ای به آن داده شده است، یک شاخص روی یک ویژگی آن ایجاد کنیم و Neo4j هر زمان که دیتابیس تغییر کند آن ایندکس را آپدیت می کند.

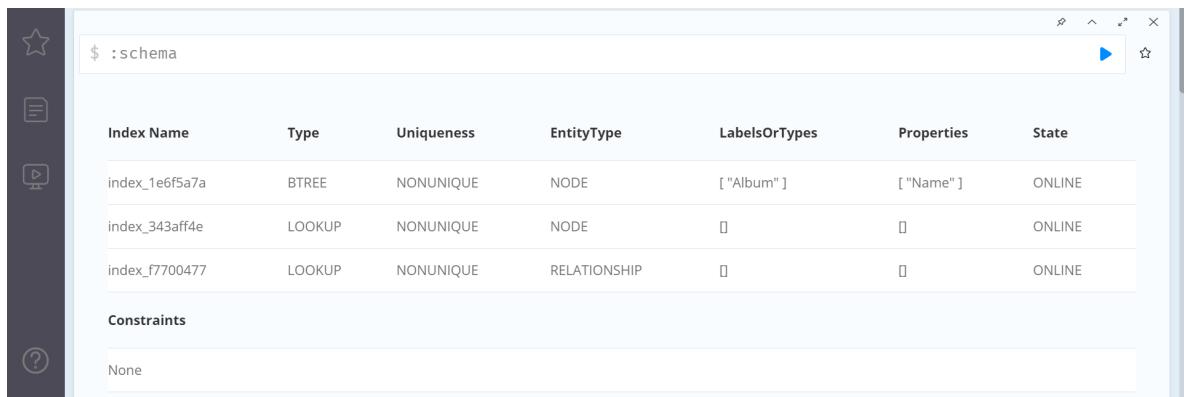
برای ساخت آن از سینتکس CREATE INDEX ON استفاده می کنیم.



```
neo4j$ CREATE INDEX ON :Album(Name)
Added 1 index, completed after 13 ms.
```

در اینجا ما یک ایندکس روی ویژگی Name همه گره های با label Album برابر با label آیجاد کرده ایم.

اگر بخواهیم در پایگاه داده خود، همه ایندکس ها و constraints ها را ببینیم از سینتکس :schema استفاده می کنیم.



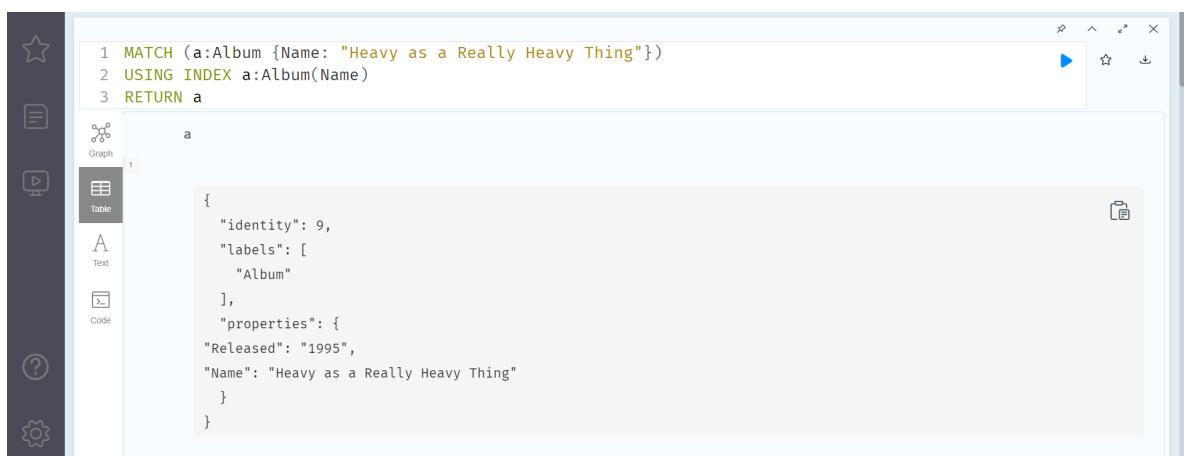
Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
index_1e6f5a7a	BTREE	NONUNIQUE	NODE	["Album"]	["Name"]	ONLINE
index_343aff4e	LOOKUP	NONUNIQUE	NODE			ONLINE
index_f7700477	LOOKUP	NONUNIQUE	RELATIONSHIP			ONLINE

Constraints

None

همچنین ما می توانیم روی داده های خود ایندکس شاخصی تعریف کنیم که در هنگام جستجو به کار رود.

برای این کار از سینتکس USING INDEX استفاده می کنیم.



```
1 MATCH (a:Album {Name: "Heavy as a Really Heavy Thing"})
2 USING INDEX a:Album(Name)
3 RETURN a
```

Graph

a

```
{
  "identity": 9,
  "labels": [
    "Album"
  ],
  "properties": {
    "Released": "1995",
    "Name": "Heavy as a Really Heavy Thing"
  }
}
```

حال می خواهیم روی گره ها و یال ها constraints تعریف کنیم.

به اجرای integrity constraints داده ها کمک می کنند، چون از وارد کردن داده های نادرست توسط کاربران جلوگیری می کنند. یعنی اگر شخصی سعی کند در هنگام اعمال محدودیت، نوع اشتباهی از داده ها را وارد کند، یک پیام خطای دریافت می کند.

یکی از این محدودیت ها، محدودیت یکتاپی داده ها است. یعنی دو گره ای دارای برچسب یکسانی هستند نمی توانند یک مقدار یکسان داشته باشند.

محدودیت دیگری که داریم محدودیت وجود ویژگی است. که طبق این محدودیت یک گره یا یال با label خاص، باید ویژگی خاصی را به طور حتمی شامل شود.

برای ایجاد محدودیت از سینتکس CREATE CONSTRAINT ON استفاده می کنیم.

برای محدودیت وجود یکتاپی از سینتکس IS UNIQUE استفاده می کنیم.

```
neo4j$ CREATE CONSTRAINT ON (a:Artist) ASSERT a.Name IS UNIQUE
Added 1 constraint, completed after 79 ms.
```

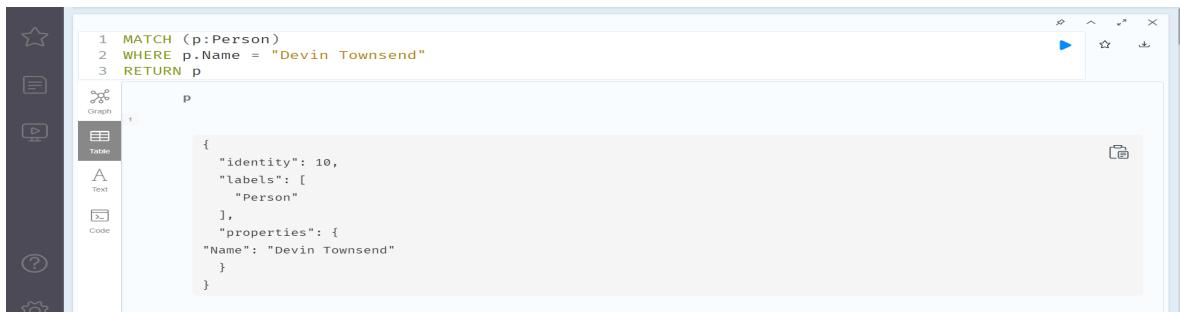
```
neo4j$ CREATE (:Artist {Name: "Strapping Young Lad"})
ERROR Neo.ClientError.Schema.ConstraintValidationFailed
Node(8) already exists with label `Artist` and property `Name` = 'Strapping Young Lad'
```

که مشاهده می شود وقتی می خواهیم داده تکراری وارد دیتابیس کنیم با خطای رو به رو می شویم.

برای محدودیت وجود ویژگی از سینتکس EXIST استفاده می کنیم.

```
neo4j$ CREATE CONSTRAINT ON (a:Artist) ASSERT exists(a.Name)
Added 1 constraint, completed after 5 ms.
```

حال اگر بخواهیم گره ها یا یال هایی را انتخاب کنیم از سینتکس MATCH استفاده می کنیم. این سینتکس داده را طبق معیار دلخواه ما پیدا می کند ولی بر نمی گرداند.



```

1 MATCH (p:Person)
2 WHERE p.Name = "Devin Townsend"
3 RETURN p
  
```

The screenshot shows the Neo4j Browser interface. On the left, there's a sidebar with icons for star, comment, video, graph, table, text, code, and help. The main area has a query editor with the following Cypher code:

```

MATCH (p:Person)
WHERE p.Name = "Devin Townsend"
RETURN p
  
```

Below the query, the results are displayed in a table format. There is one row with the following data:

p
<pre> { "identity": 10, "labels": ["Person"], "properties": { "Name": "Devin Townsend" } } </pre>

اگر بخواهیم از روش دیگری به غیر از استفاده از WHERE داده را پیدا کنیم بدین صورت عمل می کنیم.



```

neo4j$ MATCH (p:Person {Name: "Devin Townsend"}) RETURN p
  
```

This screenshot shows the same Neo4j Browser interface. The sidebar and the table result set are identical to the previous one. The difference is in the query editor, which now includes a WHERE clause:

```

MATCH (p:Person {Name: "Devin Townsend"})
RETURN p
  
```

همچنین برای پیدا کردن یک یال بین دو گره از روش زیر استفاده می کنیم.



```

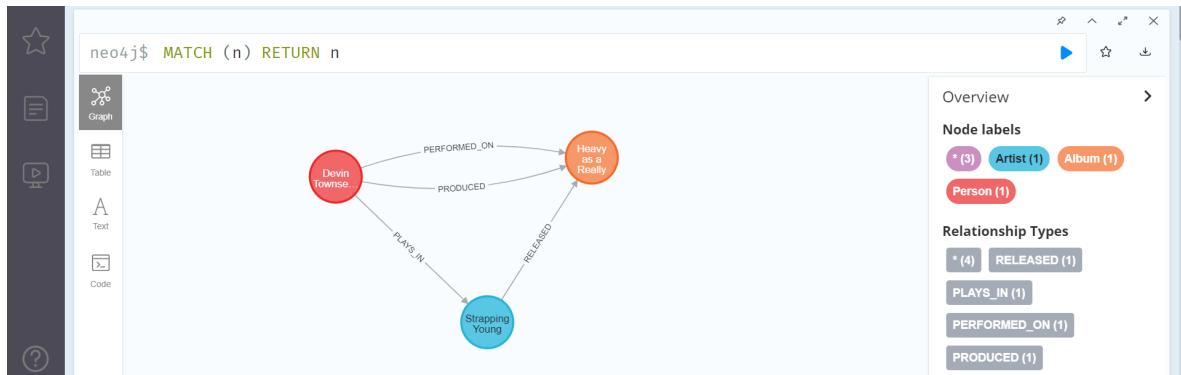
1 MATCH (a:Artist)-[:RELEASED]-(b:Album)
2 WHERE b.Name = "Heavy as a Really Heavy Thing"
3 RETURN a
  
```

This screenshot shows the Neo4j Browser interface again. The sidebar and the table result set are consistent with the previous ones. The query editor contains the following Cypher code:

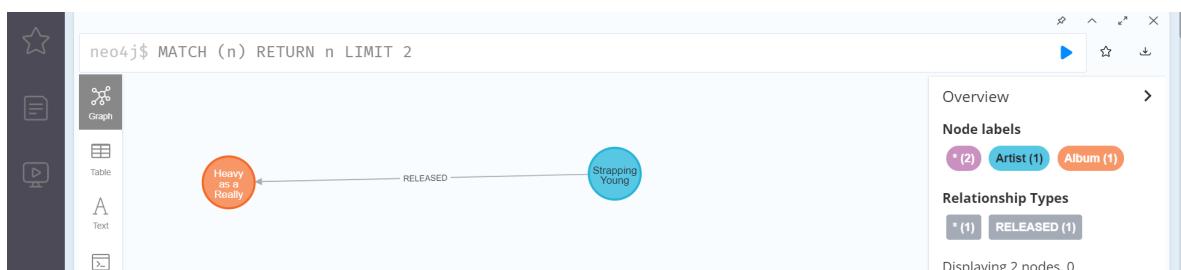
```

MATCH (a:Artist)-[:RELEASED]-(b:Album)
WHERE b.Name = "Heavy as a Really Heavy Thing"
RETURN a
  
```

ما می توانیم با سینتکس زیر همه گره ها و یال های گراف را برگردانیم.

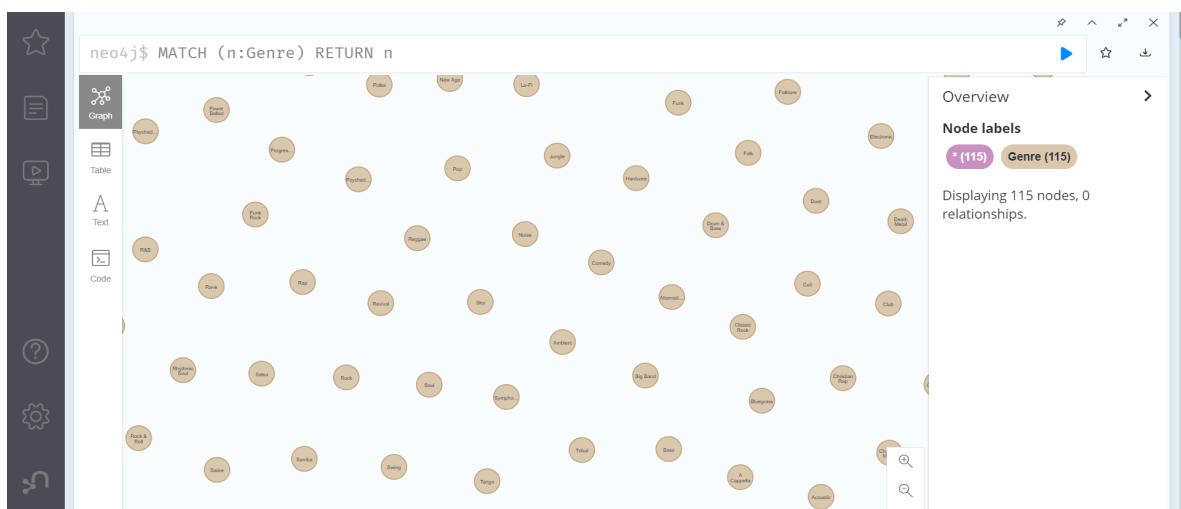


و همچنین با سینتکس LIMIT می توانیم جواب را محدود کیم.



حال می خواهیم که یک پایگاه داده را از فایل load کنیم.

که از سینتکس LOAD CSV FROM استفاده می کنیم.



حال فایلی که دارای header است را طبق این سینتکس load کرده و نمایش می دهیم.

روش های دیگری برای load کردن داده ها از فایل نیز وجود دارد.

حال با سینتکس DROP INDEX ON ایندکسی که روی داده ها ایجاد کرده بودیم را حذف می کنیم.

که در عکس دوم مشاهده می کنید پاک شده است.

Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
constraint_6a938e7e	BTREE	UNIQUE	NODE	["Artist"]	["Name"]	ONLINE
index_343aff4e	LOOKUP	NONUNIQUE	NODE	[]	[]	ONLINE
index_f7700477	LOOKUP	NONUNIQUE	RELATIONSHIP	[]	[]	ONLINE

Constraints

```
ON ( artist:Artist ) ASSERT (artist.Name) IS NOT NULL
ON ( artist:Artist ) ASSERT (artist.Name) IS UNIQUE
```

حال محدودیت ایجاد شده را با سینتکس `DROP CONSTRAINT ON` پاک کرده و نتیجه را مشاهده می کنیم.

```
neo4j$ DROP CONSTRAINT ON (a:Artist) ASSERT a.Name IS UNIQUE
Removed 1 constraint, completed after 3 ms.
```

Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
index_343aff4e	LOOKUP	NONUNIQUE	NODE	□	□	ONLINE
index_f7700477	LOOKUP	NONUNIQUE	RELATIONSHIP	□	□	ONLINE

Constraints

```
ON ( artist:Artist ) ASSERT (artist.Name) IS NOT NULL
```

حال برای حذف گره ها از سینتکس `DELETE` استفاده می کنیم. باید توجه شود که قبل از حذف گره ها باید روابط بین آن ها حذف شود.

```
neo4j$ MATCH (a:Album {Name: "Killers"}) DELETE a
Deleted 1 node, completed in less than 1 ms.
```

همچنین می توانیم چند گره را به یک باره حذف کنیم. که برای یک کار گره های مختلف را با "" در سینتکس `MATCH` جدا می کنیم.

و با سینتکس `MATCH (n) DELETE n` می توانیم همه گره های بدون رابطه را حذف کنیم.

حال برای حذف یک یال از سینتکس زیر استفاده می کنیم.

```
neo4j$ MATCH ()-[r:RELEASED]-() DELETE r
Deleted 1 relationship, completed after 4 ms.
```

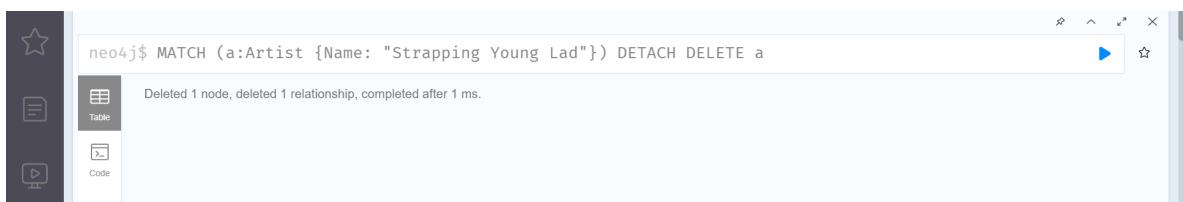
که از سینتکس دقیق تر زیر هم می‌توان استفاده کرد.



```
neo4j$ MATCH (:Artist)-[r:RELEASED]-(:Album) DELETE r
(no changes, no records)
```

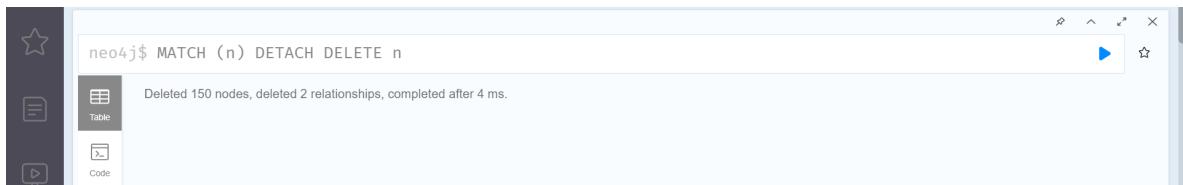
همچنین می‌توانیم چند یال را به یک باره حذف کنیم. که برای یک کاریال‌های مختلف را با "-" در سینتکس MATCH جدا می‌کنیم.

همچنین می‌توانیم با سینتکس DELETE DELETE یک گره و همه روابط آن گره را حذف کنیم.



```
neo4j$ MATCH (a:Artist {Name: "Strapping Young Lad"}) DETACH DELETE a
Deleted 1 node, deleted 1 relationship, completed after 1 ms.
```

در آخر، با سینتکس زیر کل دیتابیس را پاک می‌کنیم.



```
neo4j$ MATCH (n) DETACH DELETE n
Deleted 150 nodes, deleted 2 relationships, completed after 4 ms.
```

● بخش دوم: کار روی دیتاست تحلیل فیلم

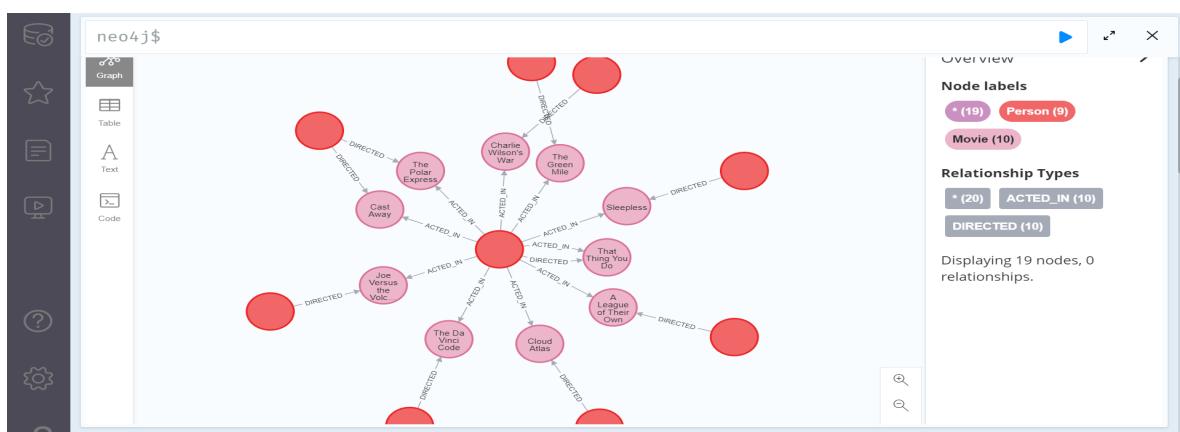
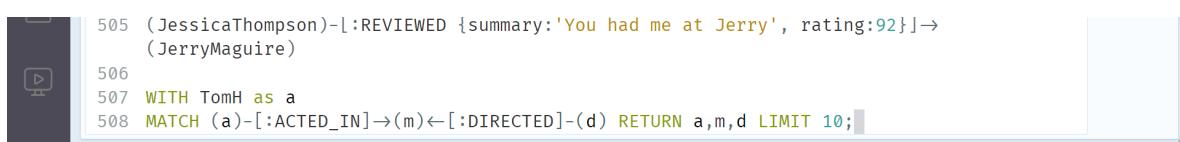
در این قسمت روی دیتاست یک فیلم که رابطه بین بازیگران و فیلم‌ها و کارگردانان فیلم‌ها آورده شده است.

با دستور play movie graph: کد ساخت این دیتابیس در اختیار ما قرار می‌گیرد.

این کد را که گره‌ها و یال‌های گراف ما را می‌سازد اجرا می‌کنیم و خروجی زیر را مشاهده می‌کنیم.



در دو خط آخر این کد، یک زده شده است. در نتیجه خروجی زیر را مشاهده می کنیم.



این گراف 10 تا از فیلم هایی تام هنکس در آن ها به عنوان بازیگر حضور داشته است به همراه کارگردان آن فیلم ها آورده شده است. که این فیلم ها و کارگردانان و بازیگران به صورت گره و رابطه "بازی کردن فیلم" و "کارگردانی فیلم" به صورت یال آمده است.

حال در ادامه می خواهیم که روی دیتابیس Query بزنیم و فیلم ها و بازیگران خاصی را مشاهده کنیم.

در این Query یک گره که نام بازیگر آن تام هنکس است را پیدا کرده ایم.

در ادامه در Query دیگری یک گره که عنوان آن Cloud Atlas است را پیدا کرده ایم.

The screenshot shows the Neo4j browser interface. On the left, there is a sidebar with icons for Star, Graph, Table, and Text. The main area displays two nodes: a red circular node labeled 'Tom Hanks' and a pink circular node labeled 'Cloud Atlas'. The query in the top bar is:

```
neo4j$ MATCH (tom {name: "Tom Hanks"}) RETURN tom
```

To the right of the nodes, there is a 'Node Properties' panel for the 'Person' node. It shows the following properties:

<id>	71
born	1956
name	Tom Hanks


```
neo4j$ MATCH (cloudAtlas {title: "Cloud Atlas"}) RETURN cloudAtlas
```

Below the nodes, there is an 'Overview' panel showing:

- Node labels: * (1) Person, Movie (1)
- Displaying 1 nodes, 0 relationships.

در ادامه یک Query روی همه گره های Person زده ایم و ویژگی name آن ها را انتخاب کرده ایم و با LIMIT ده تا از آن ها را نشان داده ایم.

The screenshot shows the Neo4j browser interface. On the left, there is a sidebar with icons for Star, Graph, Table, Text, Code, Help, Settings, and a question mark. The main area displays a table with the following data:

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

The query in the top bar is:

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

در Query دیگری با استفاده از WHERE فضای جستجو را محدود کرده و فیلم های بازه خاصی را نشان می دهیم. در اینجا روی ویژگی released شرط اعمال کرده ایم.



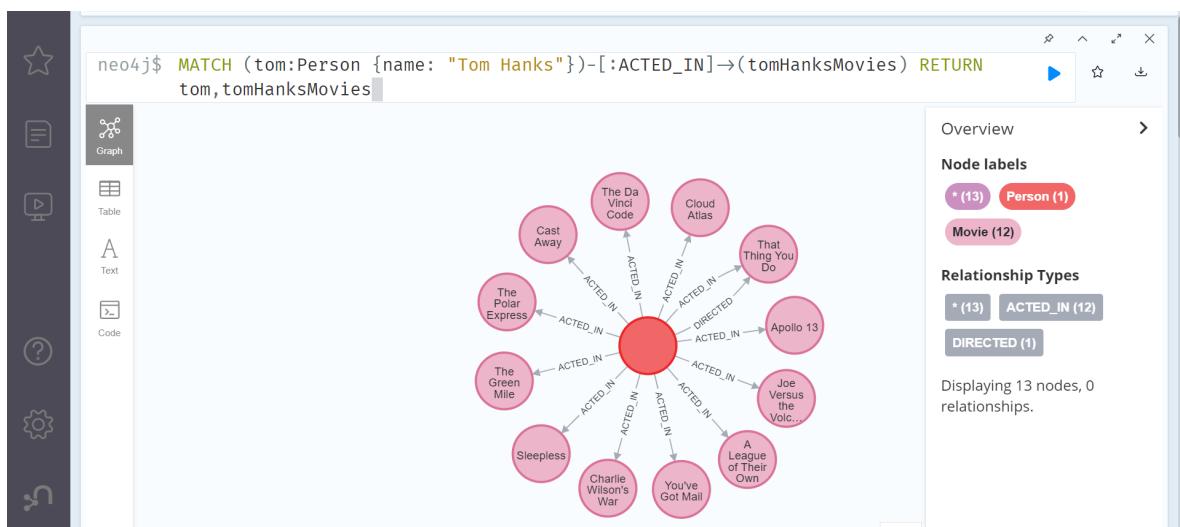
The screenshot shows the Neo4j browser interface with a query results table. The query is:

```
neo4j$ MATCH (nineties:Movie) WHERE nineties.released >= 1990 AND nineties.released < 2000 RETURN nineties.title
```

The results are:

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"

حال در این Query می خواهیم با استفاده از گره ها و یال ها، فیلم هایی که تام هنکس در آن ها بازی کرده است را نشان دهیم.



The screenshot shows the Neo4j browser interface with a graph visualization. The query is:

```
neo4j$ MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]-(tomHanksMovies) RETURN tom,tomHanksMovies
```

The graph displays nodes for movies and their relationships to Tom Hanks. Nodes include: Cast Away, The Da Vinci Code, Cloud Atlas, That Thing You Do, Apollo 13, The Polar Express, The Green Mile, Sleepless, Charlie Wilson's War, You've Got Mail, A League of Their Own, Joe Versus the Volc..., and Cloud Atlas. Relationships are labeled ACTED_IN or DIRECTED.

On the right, there is an Overview panel with the following information:

- Node labels:** * (13), Person (1), Movie (12)
- Relationship Types:** * (13), ACTED_IN (12), DIRECTED (1)
- Displaying 13 nodes, 0 relationships.

در این Query می خواهیم با استفاده از گره ها و یال ها، کارگردان فیلم Cloud Atlas را نشان دهیم.



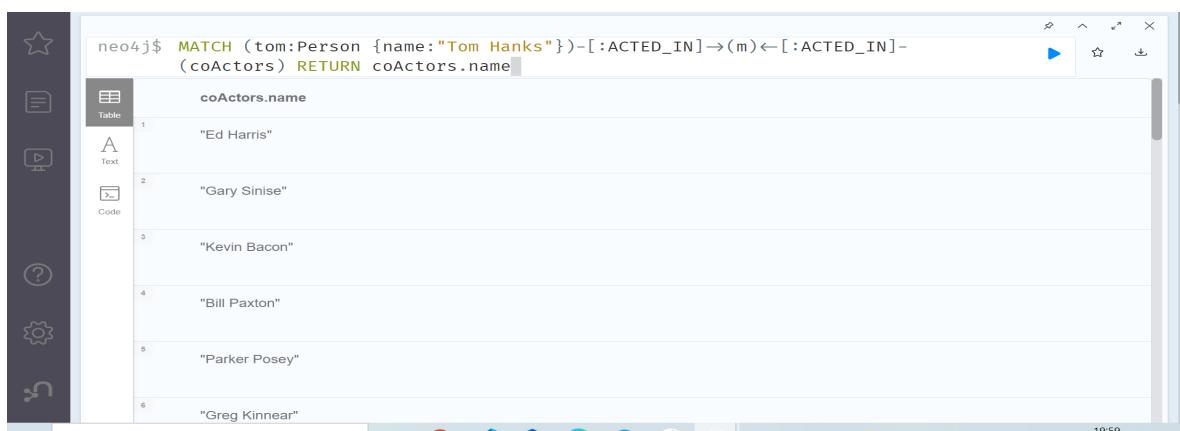
```
neo4j$ MATCH (cloudAtlas {title: "Cloud Atlas"})-[:DIRECTED]-(directors) RETURN directors.name
```

directors.name

1	"Tom Tykwer"
2	"Lilly Wachowski"
3	"Lana Wachowski"

Started streaming 3 records in less than 1 ms and completed in less than 1 ms.

حال یک Query پیچیده تر را می زنیم. تمام فیلم هایی که تام هنکس بازی کرده است (یال ACTED_IN) را پیدا کرده و سپس تمام بازیگران این فیلم ها (یال های ACTED_IN) را هم نشان می دهیم. نود وسط m، نماینده فیلم منتظر در آن مرحله است.

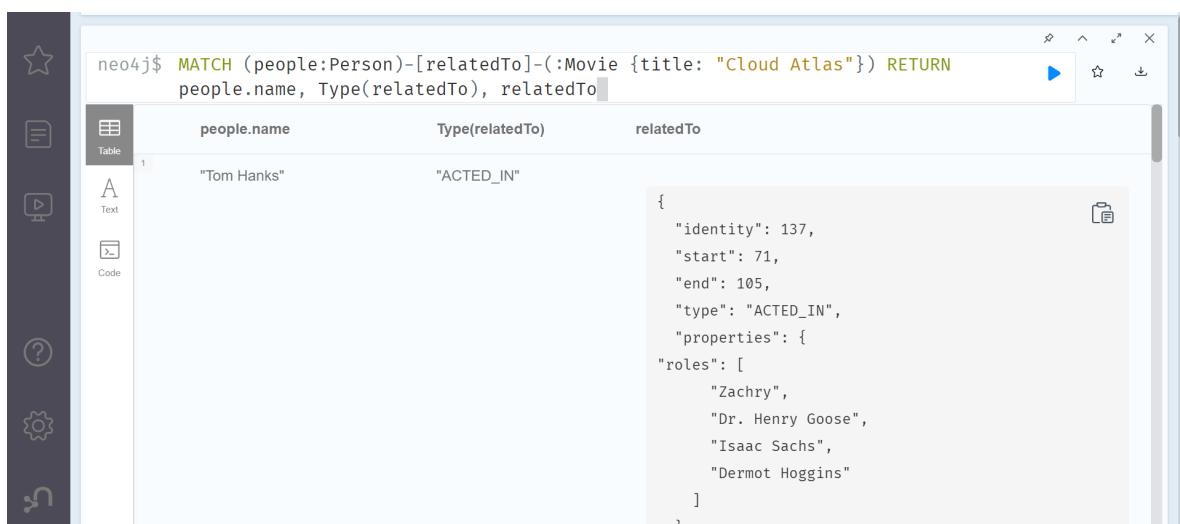


```
neo4j$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]-(m)-[:ACTED_IN]->(coActors) RETURN coActors.name
```

coActors.name

1	"Ed Harris"
2	"Gary Sinise"
3	"Kevin Bacon"
4	"Bill Paxton"
5	"Parker Posey"
6	"Greg Kinnear"

در بعدی نام افرادی که به نحوی با فیلم Cloud Atlas یال دارند به همراه یال مربوط و نوع ارتباط آن یال را نشان داده ایم.



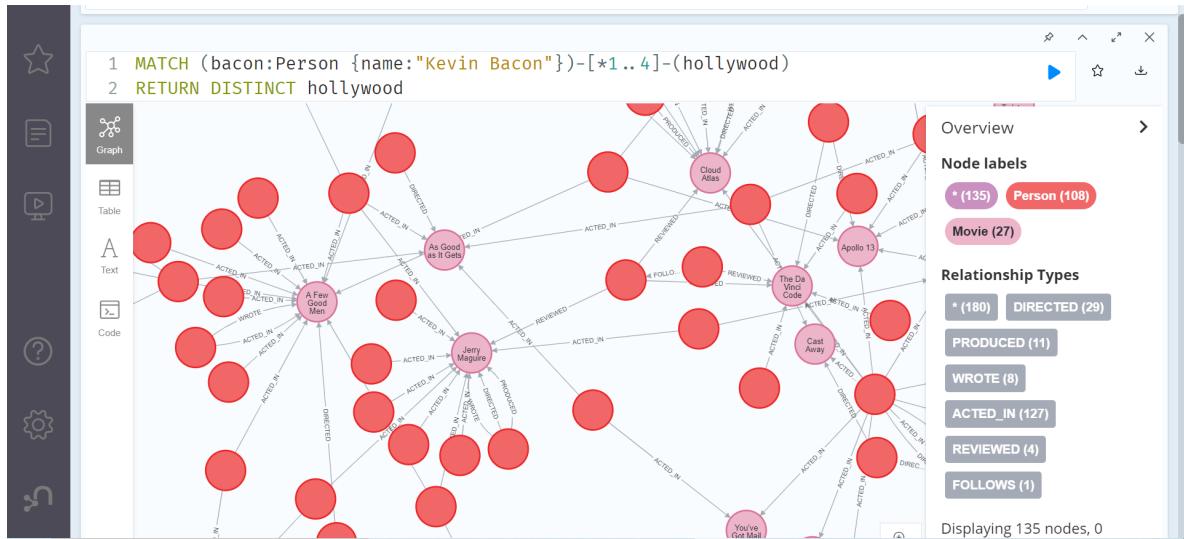
```
neo4j$ MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"}) RETURN people.name, Type(relatedTo), relatedTo
```

people.name Type(relatedTo) relatedTo

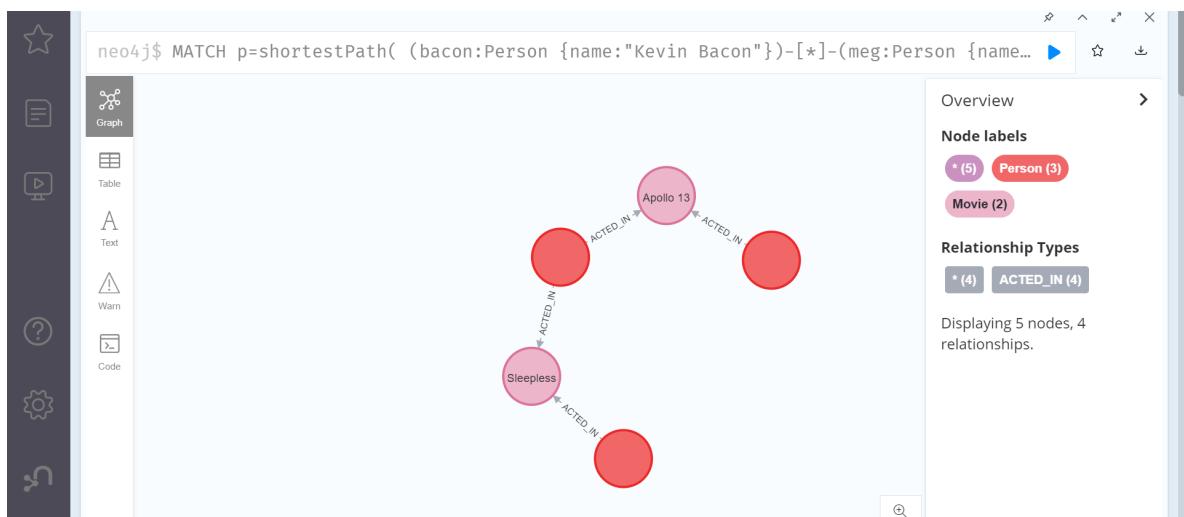
1	"Tom Hanks"	"ACTED_IN"
---	-------------	------------

{
 "identity": 137,
 "start": 71,
 "end": 105,
 "type": "ACTED_IN",
 "properties": {
 "roles": [
 "Zachry",
 "Dr. Henry Goose",
 "Isaac Sachs",
 "Dermot Hoggins"
]
 }
}

حال در Query بعدی همه فیلم‌ها و افرادی که با فرد خاصی، یال‌ها یا همان مسیر به طول ۱ تا ۴ دارند را نشان داده ایم.



حال در این Query می‌خواهیم کوتاه‌ترین مسیر یال‌های بین دو گره افراد را پیدا کنیم.



در این Query می‌خواهیم بازیگرانی که با تام هنکس همکار نبوده‌اند را پیدا کنیم.

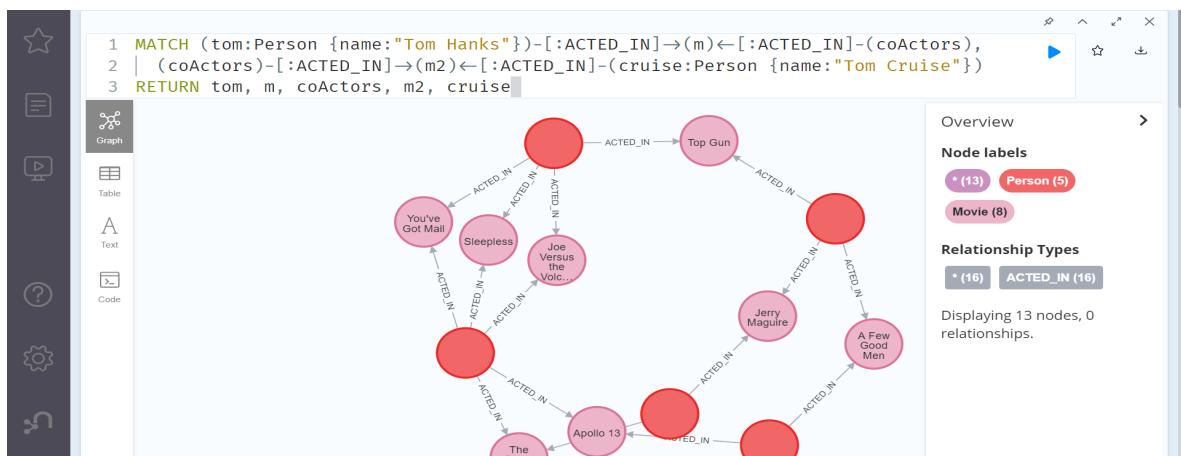
در ابتدا فیلم‌های تام هنکس را پیدا کرده و سپس بازیگران همکار تام هنکس در آن فیلم‌ها را پیدا کرده، سپس بازیگران همکار با بازیگرانی که در فیلم‌هایی با تام هنکس همکاری داشته را پیدا کرده و سپس در یک WHERE شرط دلخواه خود را می‌نویسیم.



The screenshot shows the Neo4j browser interface with a query results table. The table has two columns: 'Recommended' and 'Strength'. The data is as follows:

Recommended	Strength
"Tom Cruise"	5
"Zach Grenier"	5
"Cuba Gooding Jr."	4
"Keanu Reeves"	4
"Jack Nicholson"	3

حال در آخر مانند Query قبلی فیلم هایی را که در آن ها، تام کروز با همکاران تام هنکس، همکاری داشته است را نشان دهیم.



حال کل دیتابیس را پاک می کنیم.



The screenshot shows the Neo4j browser interface with a command-line interface (CLI) window. The user runs the command `DETACH DELETE n`, which deletes 171 nodes and 253 relationships. Below this, another CLI window shows the command `RETURN n` with the message '(no changes, no records)'.

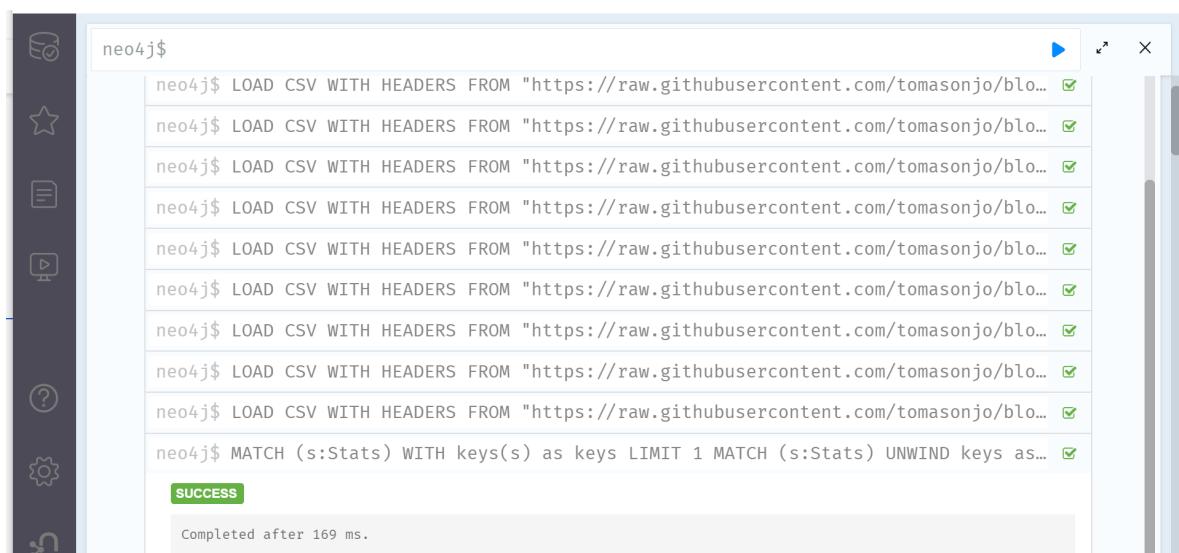
بخش سوم: تحلیل روابط کارکتر های دنیای سینمایی مارول

در این مقاله به تحلیل شخصیت ها و روابط و ویژگی های آن ها در دنیای سینمایی مارول پرداخته شده است. در ابتدا به آنالیز ساده این گراف پرداخته شده است و Query های ساده مختلفی روی آن اجرا شده و نتایجش تحلیل می شود. و سپس الگوریتم های پیچیده تری مثل KNN و Louvain Label Propagation و Modularity بر روی این گراف اجرا می شود و روابط مختلف قهرمان ها نیز بررسی می شود. در ادامه ریز به ریز این عملیات ها به همراه نتیجه خروجی آن ها آورده شده است.

در ابتدا داده ها را با سینتکس LOAD CSV دریافت می کنیم. ارتباطات بین این گره ها و یال ها هم ساخته شده اند.



```
neo4j$ CALL apoc.schema.assert({Character:['name']},{Comic:['id'], Character:['id']})
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blo...
```

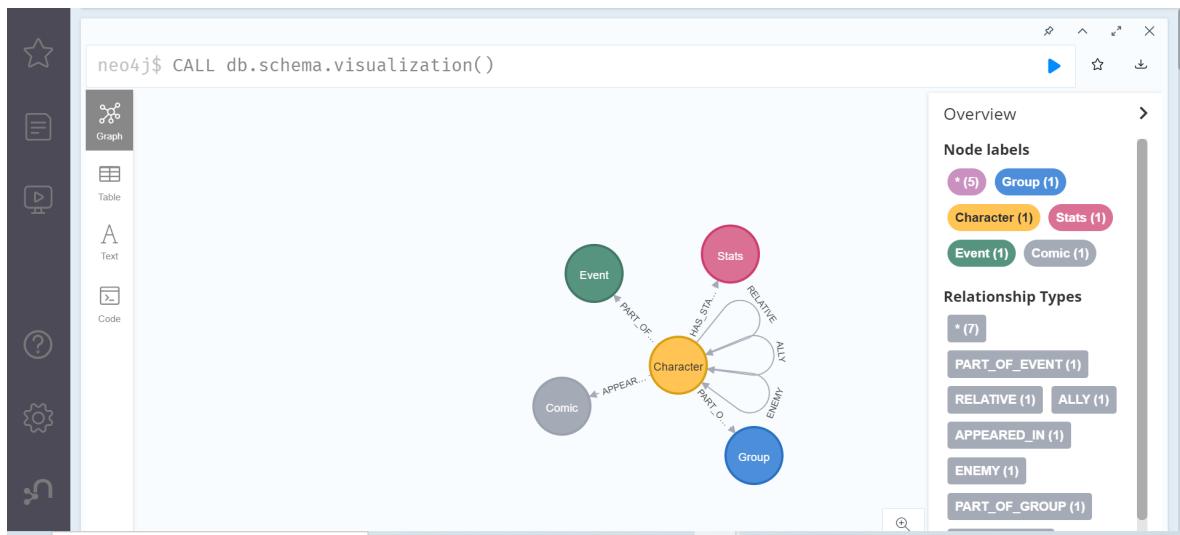
```
neo4j$ LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tomasonjo/blo...
neo4j$ MATCH (s:Stats) WITH keys(s) as keys LIMIT 1 MATCH (s:Stats) UNWIND keys as...
```

SUCCESS

Completed after 169 ms.

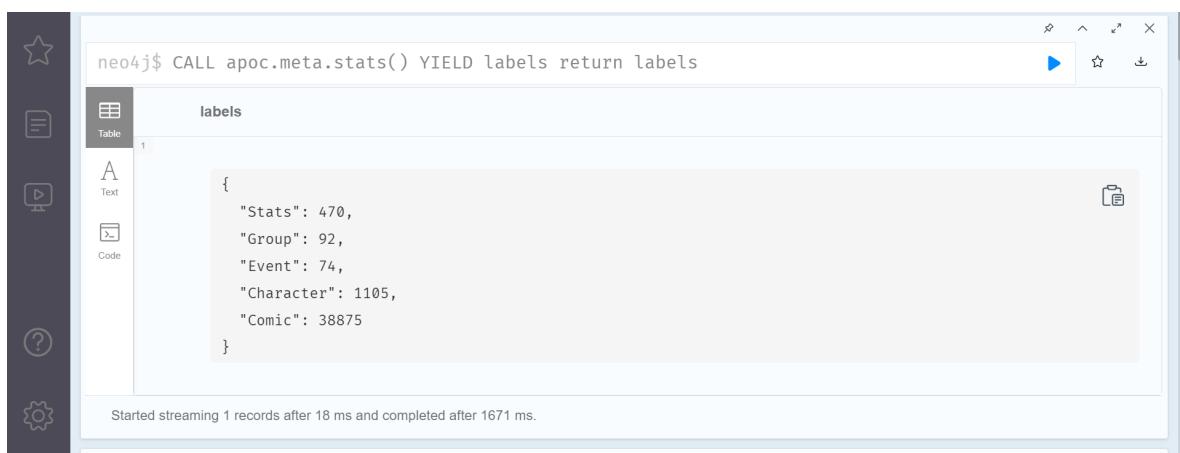
در عکس زیر دیدی کلی از شمای گراف می بینید. ما در اینجا از سینتکس visualization استفاده می کنیم. در مرکز نمودار، شخصیت اصلی یا همان قهرمان فیلم قرار دارد.

آنها می توانند در کمیک های متعدد و رویداد ها و گروه های مختلف ظاهر شوند. برای برخی از شخصیت ها، همچنین آماری از سرعت و مهارت های مبارزه آن ها می دانیم. همچنین، ما بین شخصیت ها پیوندهای اجتماعی داریم که نشان دهنده روابط فامیلی، دوستی یا دشمنی است.



در عکس زیر با استفاده از سینتکس stats، یک دید کلی از گراف خود دریافت خواهیم کرد.

طبق این گزارش، گراف ما 92 تا گروه، 74 تا رویداد، 1105 تا کارکتر و 38875 تا کمیک دارد.



در عکس بعدی، خروجی یک Query را مشاهده می کنید که 5 تا کارکتر پر تکرار در کمیک ها را برگردانده است و نام آن ها به همراه تعداد تکرار حضور آن ها در کمیک ها آمده است. کارکتر اسپایدرمن با تعداد 3357 حضور، بیشترین کارکتر پر تکرار است. به نظر می رسد فیلم های جدید

کمپانی مارول از شخصیت های محبوب بیشترین بهره خود را برده است. سینتکس این Query بدین شکل است که به ازای همه کارکتر ها، تعداد یال های APPEARED_IN که به گره کمیک وصل هستند را پیدا می کند و به صورت نزولی بر اساس تعداد این یال ها مرتب می شود.

با استفاده از LIMIT پنج تا از آن ها را نشان می دهیم.

```

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

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

در Query بعدی، ما رویدادهایی را با بیشترین تعداد قهرمانان شرکت کننده را بررسی خواهیم کرد. همچنین مشخصات آن رویداد نیز آمده است. سینتکس این Query بدین شکل است که به ازای همه رویداد ها، گره هایی که به گره آن رویداد یال PART_OF_EVENT دارد را به همراه توضیحات آن رویداد برگردانده می شود و به صورت نزولی (Desc) و بر اساس تعداد گره ها مرتب می شود.

با استفاده از LIMIT پنج تا از آن ها را نشان می دهیم.

```

1 MATCH (e:Event)
2 RETURN e.title as event,
3      size((e)-[:PART_OF_EVENT]-()) as count_of_heroes,
4      e.start as start,
5      e.end as end,
6      e.description as description
7 ORDER BY count_of_heroes DESC
8 LIMIT 5
    
```

event	count_of_heroes	start	end	description
1 "Fear Itself"	132	"2011-04-16 00:00:00"	"2011-10-18 00:00:00"	"The Serpent, God of Fear and brother to the Adversary, has come to Earth to recruit the super-villains of the world to his cause."
2 "Dark Reign"	128	"2008-12-01 00:00:00"	"2009-12-31 12:59:00"	"Norman Osborn came out the hero of Secret Invasion and became the new Green Goblin, the Dark Reign."
3 "Acts of Vengeance!"	93	"1989-12-10 00:00:00"	"2008-01-04 00:00:00"	"Loki sets about convincing the super-villains of the world to help him overthrow Thor as King of Asgard."
4 "Secret Invasion"	89	"2008-06-02 00:00:00"	"2009-01-25 00:00:00"	"The shape-shifting Skrulls have been infiltrating the Marvel Universe."

حال در Query بعدی می خواهیم گروه های پر جمعیت این گراف به همراه تعداد اعضای آن ها را پیدا کنیم. سینتکس این Query بدین شکل است که به ازای همه گروه ها، تعداد یال های PART_OF_GROUP که به گره اعضا وصل هستند را پیدا می کند و به صورت نزولی (Desc) و بر اساس تعداد گره ها مرتب می شود. با استفاده از LIMIT پنج تا از آن ها را نشان می دهیم.

این خروجی نشان می دهد که گروه x-man با 41 عضو قهرمان، بزرگ ترین گروه است.

```

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

group	members
1 "X-Men"	41
2 "Avengers"	31
3 "Defenders"	26
4 "Next Avengers"	14
5 "Guardians of the Galaxy"	12

در عکس بعدی بررسی کرده ایم که رابطه اعضای یک گروه دشمنانه است یا خیر.

سینتکس این Query بدین شکل است که دو کاراکتر تعریف شده است که هر دو کاراکتر به سمت یک گروه یال PART_OF_GROUP دارند و هم گروه هستند. سپس در WHERE گفته شده است که این دو کاراکتر به سمت هم یالی از نوع ENEMY داشته باشند. و با چک کردن ویژگی id آن ها، از نمایش داده های تکراری جلوگیری می کنیم.

```

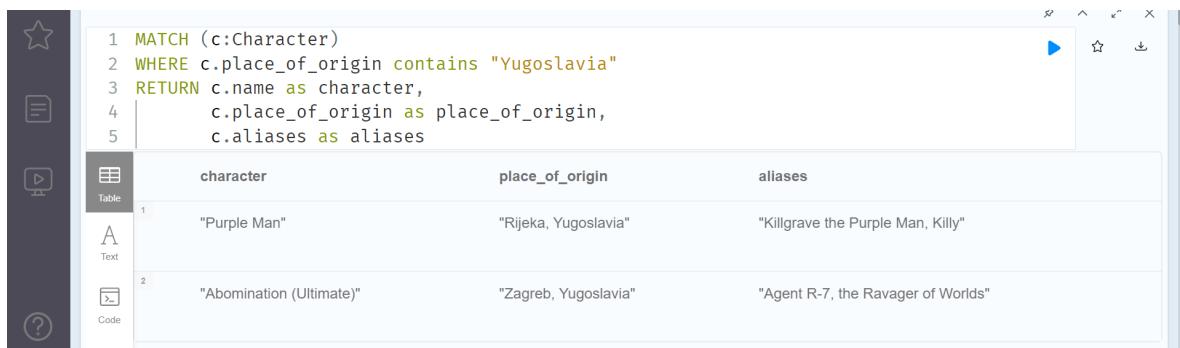
1 MATCH (c1:Character)-[:PART_OF_GROUP]-(g:Group)-[:PART_OF_GROUP]-(c2:Character)
2 WHERE (c1)-[:ENEMY]-(c2) AND id(c1) < id(c2)
3 RETURN c1.name as character1, c2.name as character2, g.name as group
  
```

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

در عکس بعدی بررسی کرده ایم که آیا قهرمانی از کشور یوگوسلاوی وجود دارد یا خیر.

سینتکس این Query بدین شکل است که کشور همه گرهی کارکترها را در شرط WHERE چک می کند که برابر با معادل دلخواه باشد و نام و مکان زندگی و لقب او برگردانده می شود.

دو کارکتر از کشور یوگوسلاوی مشاهده می شود.



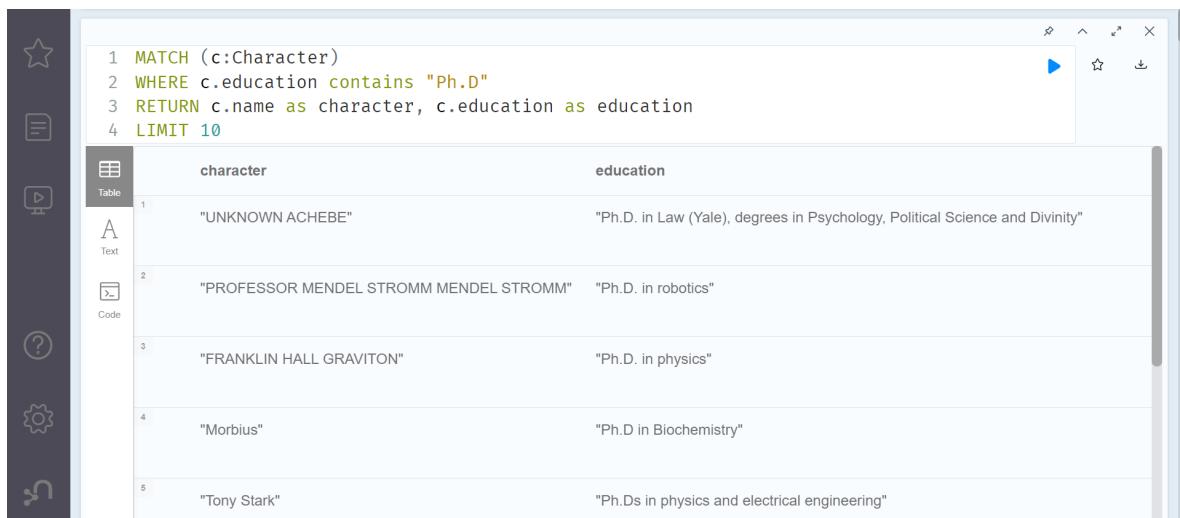
```

1 MATCH (c:Character)
2 WHERE c.place_of_origin contains "Yugoslavia"
3 RETURN c.name AS character,
       c.place_of_origin AS place_of_origin,
       c_aliases AS aliases
    
```

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

در عکس بعدی بررسی کرده ایم که آیا قهرمانی مدرک تحصیلی Ph.D دارد یا خیر.

سینتکس این Query بدین شکل است که همه گرهی کارکترها را در شرط WHERE چک می کند که برابر با معادل دلخواه باشد و نام و مدرک تحصیلی او برگردانده می شود.



```

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

	character	education
1	"UNKNOWN ACHEBE"	"Ph.D. in Law (Yale), degrees in Psychology, Political Science and Divinity"
2	"PROFESSOR MENDEL STROMM MENDEL STROMM"	"Ph.D. in robotics"
3	"FRANKLIN HALL GRAVITON"	"Ph.D. in physics"
4	"Morbius"	"Ph.D. in Biochemistry"
5	"Tony Stark"	"Ph.D.s in physics and electrical engineering"

حال در ادامه به تجزیه و تحلیل جوامع متحдан و خویشاوندان می پردازیم.

در اولین Query تعداد دوستان و دشمنان و فامیل های هر قهرمان آورده شده است.

سینتکس این Query بدین شکل است درجه هر گره کارکتر گراف که معادل با تعداد یال های آن است را محاسبه می کنیم.

این گره ها را به صورت نزولی (Desc) و بر اساس تعداد یال هایشان مرتب می شود. با استفاده از LIMIT پنج تا از آن ها را نشان می دهیم.

به نظر می رسد که اسکارلت ویچ بیشترین ترین دشمنان را دارد. ولورین بیشترین متهدان را دارد اما دشمنان زیادی نیز دارد. به نظر می رسد تریتون یک خانواده بزرگ با 17 رابطه فامیلی دارد.

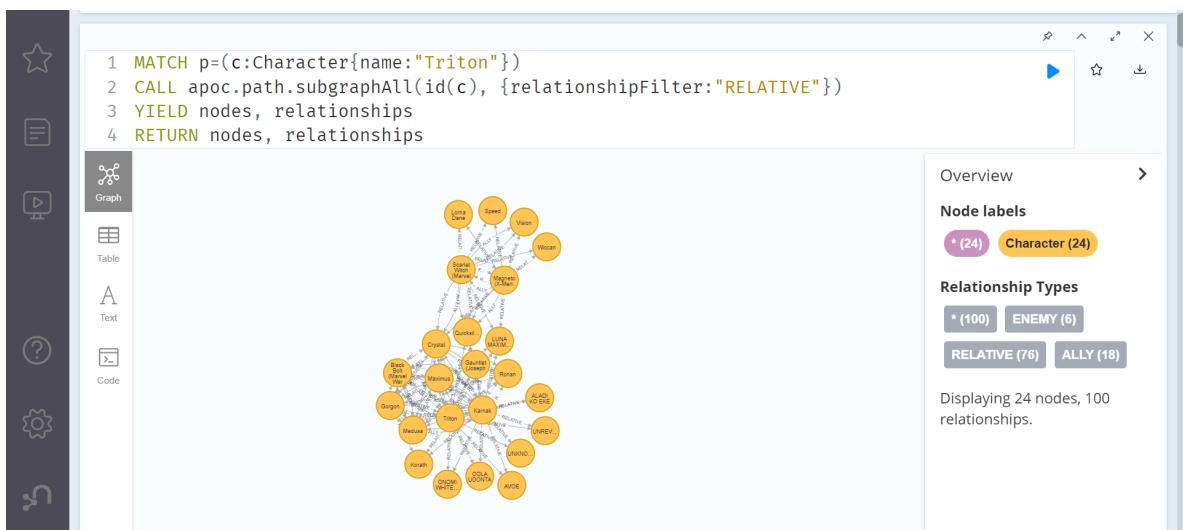
```

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

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

در Query بعدی زیر گراف رابطه فامیلی برای کارکتر Triton را بررسی می کنیم.

سینتکس این Query بدین شکل است که path.subgraphAll دو آرگومان شناسه گره و نوع یال آن را می گیرد و سپس همه یال های میان گره های دیگر را با گره داده شده، برمی گرداند.



```

neo4j$ MATCH p=(c:Character{name:'Triton'}) CALL apoc.path.subgraphAll(id(c), {relatio...

```

nodes	relationships
<pre>{ "identity": 442, "labels": ["Character"], "properties": { "aliases": "None", "education": "null", "identity": "Secret", "name": "Triton", "id": "e8a83f57-6717-403f-84e9-56ecded817c7", "place_of_origin": "null" } }</pre>	<pre>{ "identity": 65096, "start": 442, "end": 259, "type": "ENEMY", "properties": {} }</pre>

در Query بعدی قسمت ها یا جزایر جدای گراف را بررسی می کنیم. (یعنی مجموعه ای از گره ها در گراف جهت دار به گره های دیگر گراف یال ندارند).

سینتکس این Query بدین شکل است که از gds.wcc استفاده شده است و این متده این گروه گره ها را برای ما پیدا می کند. این متده دو تا آرگومان گره ها (در اینجا کارکتر ها) و یال ها (در اینجا یال ALLY) را می گیرد. در شرط WHERE گفته ایم که تک گره ها را جزو جستجو قرار ندهد. این گره ها را به صورت نزولی (Desc) و بر اساس گره های اعضا مرتب می شود. با استفاده از LIMIT پنج تا از آن ها را نشان می دهیم. بزرگترین جزیره گره ها 195 عضو دارد. سپس ما چند جزیره از گره های کوچک داریم که تنها چند عضو دارند.

```

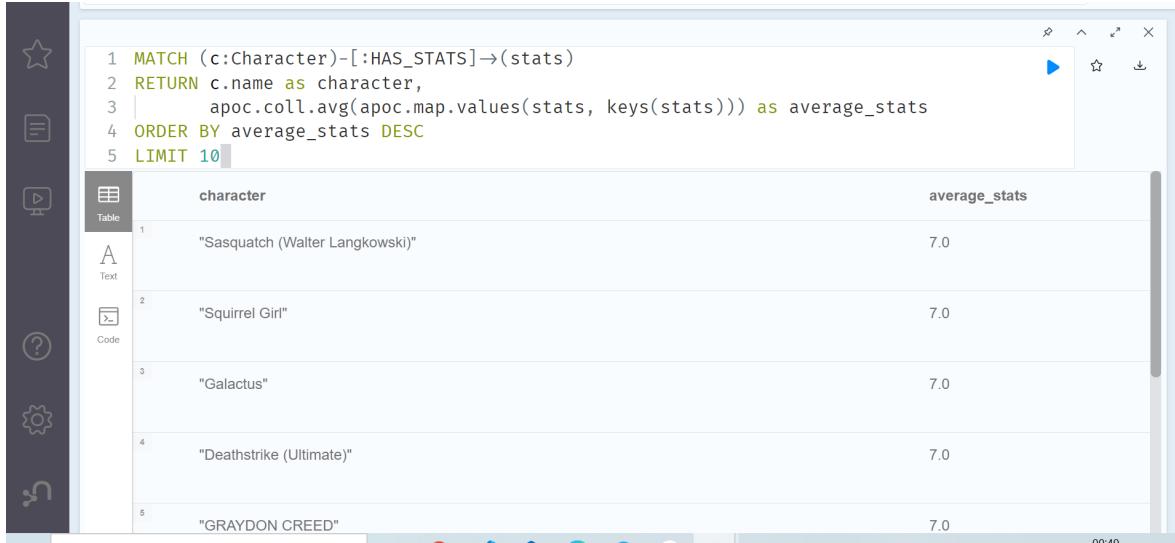
1 CALL gds.wcc.stream({
2   nodeProjection:'Character',
3   relationshipProjection:'ALLY'}
4 YIELD nodeId, componentId
5 WITH componentId, count(*) as members
6 WHERE members > 1
7 RETURN componentId, members
8 ORDER BY members DESC
9 LIMIT 5

```

componentId	members
0	195
26	4
245	3

حال می خواهیم یک Query بزنیم و میانگین یک ویژگی را برای یک گره پیدا کنیم.

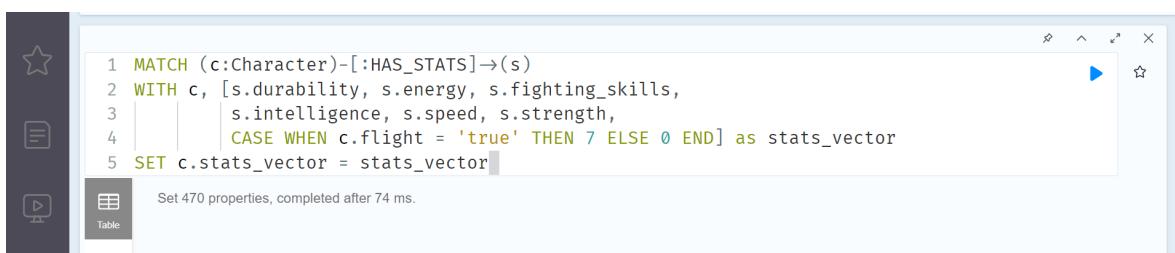
سینتکس این Query بدین شکل است که گره های کارکترها و هر گره ای که با یال HAS_STATS به آن کارکترها متصل هستند را پیدا می کنیم. سپس میانگین stats های آن گرهها را پیدا می کنیم. این گرهها را به صورت نزولی (Desc) و بر اساس امتیازهای گرهها مرتب می شود. با استفاده از LIMIT پنج تا از آنها را نشان می دهیم.



The screenshot shows the Neo4j Studio interface with a query results table. The table has two columns: 'character' and 'average_stats'. The data is as follows:

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

حال می خواهیم یک Query بزنیم و الگوریتم KNN را اجرا کنیم. در ابتدا یک بردار از آرایه اعداد به نام stats_vector تعریف شده است که در آن وضعیت شخصیت‌ها را ذخیره می‌کنیم تا برای مقایسه قهرمان‌ها با هم استفاده شود. ما از آمار کارکترها و همچنین توانایی آن‌ها در پرواز برای پر کردن بردار استفاده خواهیم کرد. از آنجایی که همه آمار‌ها محدوده یکسانی بین صفر تا هفت دارند، نیازی به نرمال‌سازی نیست. ما فقط باید ویژگی پرواز را به گونه‌ای رمزگذاری کنیم که بین صفر تا هفت باشد. شخصیت‌هایی که می‌توانند پرواز کنند، ارزش هفت را خواهند داشت، در حالی که آن‌هایی که نمی‌توانند پرواز کنند، ارزش صفر را خواهند داشت. برای رمزگذاری‌های مختلف از سینتکس CASE استفاده کرده‌ایم.



The screenshot shows the Neo4j Studio interface with a query results table. The table has one column: 'character'. Below the table, a message says "Set 470 properties, completed after 74 ms." The character listed is "Sasquatch (Walter Langkowski)".

در ادامه کارکترهایی را که دارای بردار آماری هستند را فیلتر می‌کنیم.



```
neo4j$ MATCH (c:Character) WHERE exists (c.stats_vector) SET c:CharacterStats
Added 470 labels, completed after 19 ms.
```

The screenshot shows the Neo4j browser interface with a query executed in the main panel. The query is: `MATCH (c:Character) WHERE exists (c.stats_vector) SET c:CharacterStats`. The result message indicates that 470 labels were added and the operation completed in 19 ms. The sidebar on the left contains icons for Star, Table, Warn, and Code.

یک گراف جدید با استفاده از تغییراتی (اضافه کردن بردار ویژگی و فیلتر کردن) که در گراف قبل دادیم، ایجاد می کنیم.



nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
<pre> 1 CALL gds.graph.create['marvel', 'CharacterStats', 2 '*', {nodeProperties:'stats_vector'}] </pre>	<pre> { "CharacterStats": { "properties": { "stats_vector": { "property": "stats_vector", "defaultValue": null } }, "label": "CharacterStats" } } </pre>	"marvel"	470	515	42

The screenshot shows the Neo4j browser interface with the results of the `gds.graph.create` command. The table displays the configuration for the graph named "marvel". The "nodeProjection" column shows the Cypher code used to define the node properties, specifically setting the `stats_vector` property. The "relationshipProjection" column shows the configuration for relationships, including orientation, aggregation, and type. The table also includes columns for `graphName`, `nodeCount`, `relationshipCount`, and `createMillis`.

حال در عکس زیر این الگوریتم را روی دیتابیس با داده های تغییر یافته اجرا می کنیم.

این الگوریتم چند ویژگی دارد که می توان مقدار داد.

:تعداد همسایه هایی که باید برای هر گره پیدا کرد. topK

:نرخ نمونه برای محدود کردن تعداد مقایسه ها در هر گره. SampleRate

:مقداری به درصد برای تعیین early stop. اگر بهروزرسانی ها کمتر از مقدار این پارامتر اتفاق بیفتند، الگوریتم متوقف می شود. deltaThreshold

:بین هر iteration، چند تلاش برای اتصال همسایگان گره جدید بر اساس انتخاب تصادفی انجام می شود. randomJoins

حال الگوریتم را با تعیین مقدار پارامتر ها انجام می دهیم.

```
neo4j$ CALL gds.beta.knn.mutate('marvel', {nodeWeightProperty:'stats_vector', sampleRate:0.8, topK:15, mutateProperty:'score', mutateRelationshipType:'SIMILAR'})
```

	createMillis	computeMillis	mutateMillis	postProcessingMillis	nodesCompared	relationshipsWritten	similarityDistribution
1	0	168	38	-1	470	7050	<pre>{ "p1": 0.250000953674, "max": 1.000006675720, "p5": 0.33333015441, "p90": 0.500002861022, "p50": 0.500002861022, "p95": }</pre>

حال می خواهیم یک Query بزنیم و الگوریتم Louvain Modularity را اجرا کنیم.
سینتکس این Query بدین شکل است که gds.louvain.write آرگومان هایی مثل تایپ یال ها و وزن آن ها را می گیرد و یک سری ویژگی هایی را محاسبه و نمایش می دهد. با استفاده از MATCH کارکتر هایی که ویژگی آمار stats را دارند، پیدا می کنیم.
و میانگین هر یک از ویژگی های کارکتر ها که در stats قرار دارد بر می گردانیم.

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

	writeMillis	nodePropertiesWritten	modularity	modularities	ranLevels	communityCount
1	23	470	0.6275247215494375	[0.539756178422499, 0.6275247215494375]	2	8

```
neo4j$
```

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

	community	members	fighting_skills	durability	energy	intelligence	speed
1	67	61	4.491803278688525	6.114754098360656	5.4262295081967205	4.40983606557377	5.55737704918
2	287	110	2.990909090909091	2.5181818181818185	1.8818181818181816	2.863636363636364	2.36363636363
3	18	54	4.203703703703704	3.425925925925926	2.555555555555555	3.1666666666666667	3.12962962962
4	259	38	4.421052631578945	5.026315789473682	4.052631578947367	4.236842105263159	3.47368421052

در آخر الگوریتم Label Propagation Query را انجام می دهیم. این الگوریتم بر اساس label هایی که موجود است سعی می کند که برای راس های label را تخمین بزند. سینتکس این Query بدین شکل است که باید آرگومان هایی مثل تایپ یال ها و وزن آن ها را می گیرد و یک سری ویژگی هایی را محاسبه و نمایش می دهد. با استفاده از MATCH کارکتر هایی که ویژگی آمار را دارند، پیدا می کنیم.

و میانگین هر یک از ویژگی های کارکتر ها که در stats قرار دارد بر می گردانیم.

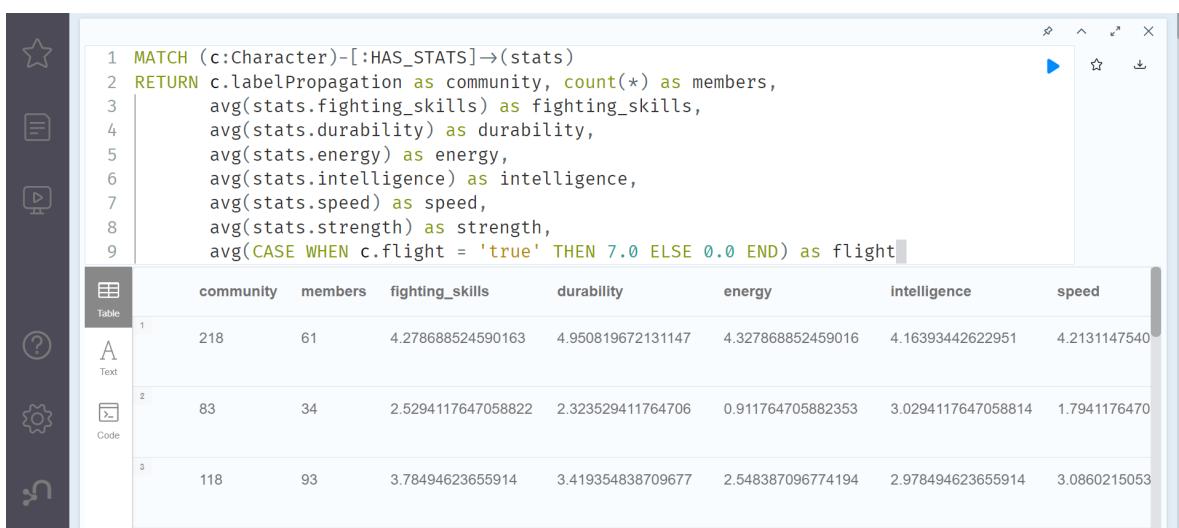


```

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

```

	writeMillis	nodePropertiesWritten	ranIterations	didConverge	communityCount	communityDistribution	postProcessingMillis
1	3	470	7	true	16	{ "p99": 93, "min": 3, "max": 93, "mean": 29.375, "p90": 61, "p50": 16, "p999": 93, "p95": 72, "p75": 46 }	5



```

1 MATCH (c:Character)-[:HAS_STATS]-(stats)
2 RETURN c.labelPropagation as community, count(*) as members,
3       avg(stats.fighting_skills) as fighting_skills,
4       avg(stats.durability) as durability,
5       avg(stats.energy) as energy,
6       avg(stats.intelligence) as intelligence,
7       avg(stats.speed) as speed,
8       avg(stats.strength) as strength,
9       avg(CASE WHEN c.flight = 'true' THEN 7.0 ELSE 0.0 END) as flight

```

	community	members	fighting_skills	durability	energy	intelligence	speed
1	218	61	4.278688524590163	4.950819672131147	4.327868852459016	4.16393442622951	4.2131147540
2	83	34	2.5294117647058822	2.323529411764706	0.911764705882353	3.0294117647058814	1.7941176470
3	118	93	3.78494623655914	3.419354838709677	2.548387096774194	2.978494623655914	3.0860215053

مشکلات و توضیحات تکمیلی

در کل مشکل خاصی نبود. ولی حجم این دستور کار خیلی زیاد و بعضی قسمت های آن طاقت فرسا بود.

آنچه آموختم / پیشنهادات

من برایم کار با دیتابیس جدیدی به غیر از SQL جذاب بود. به نظرم یادگیری خوبی هم داشت.