
**دانشگاه اصفهان**

**دانشکده مهندسی کامپیوتر**


# گزارش فنی انجام تمرین اول (گراف دانش)

**درس: مفاهیم پیشرفته در نرم‌افزار۲**

**استاد: دکتر محمدعلی نعمت‌بخش**

**دستیار آموزشی: زهرا زندیه شیرازی**


**دانشجو: رضا برزگر نوذری**

**شماره دانشجویی: ۴۰۳۳۶۱۵۰۰۵**


**لینک GitHub:**

https://github.com/RezaBN/DB403-Knowledge-Graph

**آبان ۱۴۰۳**

# فهرست مطالب:

## ۱. کتابخانه‌ها

کتابخانه‌هایی همچون pandas، spacy و networkx به ترتیب جهت پردازش داده، انجام وظایف پردازش زبان طبیعی (*NLP*) و ساخت گراف مورد استفاده واقع شدند. کتابخانه matplotlib.pyplot برای مصورسازی و نمایش گراف استفاده شد، در حالی که tqdm جهت نشان دادن میزان پیشرفت پردازش و اجرای کد بکار گرفته شد. در نهایت، کتابخانه neo4j برای تعامل با پایگاه داده گراف Neo4j مورد استفاده واقع شد. تصویر ۱ دستورات وارد کردن کتابخانه‌های ذکر شده را در Jupyter Notebook نشان میدهد.

Importing Libraries

```
In [4]: import pandas as pd

        import spacy
        from spacy import displacy
        from spacy.matcher import Matcher
        from spacy.tokens import Span

        import networkx as nx
        from tqdm import tqdm
        import matplotlib.pyplot as plt

        from neo4j import GraphDatabase
```

***تصویر ۱. وارد کردن کتابخانه‌ها مورد نیاز در Jupyter Notebook.***

## ۲. بارگذاری مجموعه داده

مجموعه داده *"wiki_sentences_v2.csv"* توسط دستور ***read_csv()*** از کتابخانه ***pandas*** بارگذاری می‌شود. این مجموعه داده شامل جملات انگلیسی استخراج شده از *Wikipedia* می‌باشد. دستور ***data.head()***، چند سطر اول از دیتاست، به عنوان یک نمای اولیه از ساختار داده‌ها، را نمایش می‌دهد. این مراحل در تصویر۲ قابل مشاهده می‌باشد.

Load the dataset

```
In [5]: data = pd.read_csv("Data/wiki_sentences_v2.csv")
        print(data.head())  # Displays the first 5 rows of the DataFrame

                                            sentence
        0  confused and frustrated, connie decides to lea...
        1  later, a woman's scream is heard in the distance.
        2          christian is then paralyzed by an elder.
        3                      the temple is set on fire.
        4                outside, the cult wails with him.
```

***تصویر ۲. بارگذاری دیتاست.***

## ۳. استخراج موجودیت‌ها و روابط

جهت استخراج موجودیت‌ها و روابط از کتابخانه ***spaCy*** استفاده شده است. *SpaCy* یک کتابخانه منبع باز برای پردازش پیشرفته زبان طبیعی در ***Python*** است. SpaCy به طور خاص به ساخت برنامه‌هایی با هدف پردازش و *"درک"* حجم زیادی از متون کمک می‌کند. به طور کلی، می‌توان از آن برای استخراج اطلاعات یا درک زبان طبیعی یا برای پیش پردازش متن در راستای یادگیری عمیق استفاده کرد.

استخراج موجودیت‌ها و روابط شامل مراحل مختلفی است و در ادامه به جزئیات هر مرحله پرداخته‌ایم.

### ۱.۳. بارگذاری مدل زبانی

ابتدا مدل زبان انگلیسی ***en_core_web_sm*** توسط دستور ***load()*** از spacy بارگذاری می‌شود. این مدل شامل ابزارهای تجزیه و تحلیل دستوری، تشخیص موجودیت‌ها و تجزیه وابستگی‌ها است. با بارگذاری این مدل، می‌توان جملات را به اجزای کوچکتر (توکن‌ها) تقسیم کرد و نقش‌های دستوری آن‌ها را شناسایی کرد، که در مراحل بعدی برای استخراج فاعل، مفعول و روابط بین آن‌ها استفاده می‌شود. کد مربوط به بارگذاری مدل زبانی انگلیسی ***en_core_web_sm*** در تصویر۳ نشان داده شده است.

*Initialize spaCy*

In [6]:
```
# Load the spaCy English language model
nlp = spacy.load("en_core_web_sm")
```

*تصویر ۳. بارگذاری مدل زبان انگلیسی en_core_web_sm.*

### ۲.۳. استخراج موجودیت‌ها

تابع ***"get_entities"*** برای شناسایی **موجودیت‌های اصلی** (فاعل و مفعول) در هر جمله طراحی شده است. این تابع از **تجزیه وابستگی دستوری** برای شناسایی نقش‌های کلمات در جمله استفاده می‌کند. کد پیاده‌سازی این تابع در تصویر۴ نشان داده شده است.

*Defining function "get_entities" to extract entities*

In [7]:
```
# The function to extract entities
def get_entities(sent):
    ent1 = ""
    ent2 = ""

    prefix = ""
    modifier = ""
    prv_tok_dep = ""    # Dependency tag of previous token in the sentence
    prv_tok_text = ""   # Previous token in the sentence

    for tok in nlp(sent):
        if tok.dep_ != "punct":
            # Check if the token is a compound word or modifier
            if tok.dep_ in ["compound", "amod"]:
                prefix += tok.text + " "
            elif tok.dep_.endswith("mod"):
                modifier += tok.text + " "

            # Identify the subject
            if tok.dep_ in ["nsubj", "nsubjpass"]:
                ent1 = modifier + prefix + tok.text
                prefix, modifier = "", ""

            # Identify the object
            elif tok.dep_ in ["dobj", "pobj"]:
                ent2 = modifier + prefix + tok.text

            # Update variables
            prv_tok_dep = tok.dep_
            prv_tok_text = tok.text

    return [ent1.strip(), ent2.strip()]

# Example usage
sentence = "Reza has a cars."
print(get_entities(sentence))
```

```
['Reza', 'cars']
```

*تصویر ۴. کد تابع "get_entities".*

**مراحل عملکرد *get_entities*:**

۱) **تعریف متغیرها:** متغیرهای ***ent1*** و ***ent2*** برای ذخیره فاعل و مفعول در جمله استفاده می‌شوند. همچنین ***prefix*** و ***modifier*** برای ذخیره عبارت‌های مرکب (مثل *"New York Times"*) و صفت‌ها (مثل *"high-speed"*) به کار می‌روند.

۲) **پردازش جمله با spaCy:** در هر جمله، کلمات (توکن‌ها) به ترتیب بررسی می‌شوند تا نقش دستوری آن‌ها شناسایی شود. برای مثال، اگر کلمه‌ای نقش دستوری ***nsubj*** یا ***nsubjpass*** داشته باشد، به عنوان فاعل(*ent1*) در نظر گرفته می‌شود و اگر نقش ***dobj*** یا ***pobj*** داشته باشد، به عنوان مفعول (*ent2*) تعیین می‌شود.

۳) **تشخیص فاعل و مفعول:** کلماتی با نقش ***compound*** یا ***amod*** به *prefix* اضافه می‌شوند که ترکیب‌ها یا صفت‌ها را شکل می‌دهند.

در نهایت، فاعل‌ها (*nsubj*، *nsubjpass*) و مفعول‌ها (*dobj*، *pobj*) شناسایی و در متغیرهای *ent1* و *ent2* ذخیره می‌شوند. به عنوان یک نمونه که در تصویر۴ مشاهده نیز می‌شود، با دادن جمله *"Reza has a car"* به تابع *get_entities*، *Reza* و *car* را به عنوان موجودیت به درستی استخراج می‌کند.

### ۳.۳. استخراج روابط

تابع ***"get_relation"*** برای شناسایی **روابط** بین موجودیت‌های فاعل و مفعول طراحی شده است. این تابع از ابزار ***Matcher*** در ***spaCy*** برای تعریف الگوهای خاص استفاده می‌کند. در اینجا، روابط معمولاً افعال یا عبارات کلیدی هستند که نشان‌دهنده ارتباط بین فاعل و مفعول‌اند. تصویر۵ کد پیاده‌سازی این تابع را نشان می‌دهد.

**مراحل عملکرد *get_relation*:**

۱) **تعریف الگوهای روابط:** الگوهایی برای یافتن روابط ایجاد می‌شود که معمولاً شامل کلماتی با نقش دستوری ***ROOT*** (مانند فعل‌های اصلی جمله) هستند. الگوها به شکل زیر تعریف می‌شوند:

- **الگوی اول:** فعل اصلی (با نقش *ROOT*)، به دنبال آن یک حرف اضافه (اختیاری)، یک عامل (اختیاری)، و یک صفت (اختیاری) قرار دارد. این الگو برای روابطی مانند *"composed by"* یا *"directed by"* مناسب است.
- **الگوی دوم:** فعل اصلی (با نقش *ROOT*)، به دنبال آن یک جزء کوچک یا فعل دیگر (اختیاری). این الگو برای روابطی مانند *"started"* یا *"has launched"* مناسب است.

۲) **تطبیق الگوها** :با استفاده از *matcher*، این الگوها بر روی جمله اعمال می‌شوند. اگر تطبیقی یافت شود، بخش تطبیق‌یافته به عنوان رابطه بین فاعل و مفعول انتخاب می‌شود.

به عنوان یک نمونه که در تصویر۵ نیز مشاهده می‌شود، با دادن جمله *"Reza has 3 cars"* به تابع *get_relation*، *has* را به عنوان رابطه به درستی استخراج می‌کند.

*Defining function "get_relation" to extract relations*

In [9]:
```python
# The function to extract relations
def get_relation(sent):
    doc = nlp(sent)
    matcher = Matcher(nlp.vocab)

    # Define patterns for extracting relationships
    patterns = [
        [{'DEP': 'ROOT'}, {'DEP': 'prep', 'OP': "?"}, {'DEP': 'agent', 'OP': "?"}, {'POS': 'ADJ', 'OP': "?"}],  # Basic root-prep structure
        [{'DEP': 'ROOT'}, {'POS': 'PART', 'OP': "?"}, {'POS': 'VERB', 'OP': "?"}],  # Verb-based pattern
    ]

    for i, pattern in enumerate(patterns):
        matcher.add(f"pattern_{i+1}", [pattern])

    matches = matcher(doc)

    # Check if any matches are found
    if matches:
        # Get the last match
        span = doc[matches[-1][1]:matches[-1][2]]
        return span.text
    else:
        return "No relation found"

# Example usage
print(get_relation("Reza has 3 cars."))
```

```
has
```

*تصویر ۵. کد تابع "get_relation".*

### ۴.۳. جمع‌بندی

تابع ***get_entities*** به کمک تجزیه وابستگی دستوری، فاعل و مفعول اصلی هر جمله را به عنوان موجودیت استخراج می‌کند. تابع ***get_relation*** با استفاده از الگوهای spaCy، فعل اصلی و روابط بین موجودیت‌ها را شناسایی می‌کند. این روابط، نشان‌دهنده ارتباط بین فاعل و مفعول است که در مراحل بعدی به عنوان یال‌های گراف استفاده می‌شود. این دو تابع با هم به ما امکان می‌دهند تا موجودیت‌ها و روابط بین آن‌ها را از جملات استخراج کرده و برای ساخت یک گراف دانش آماده کنیم.

برای استخراج موجودیت‌ها و روابط از مجموعه‌داده *wiki_sentences_v2*، جملات موجود در آن را در یک حلقه for به هر یک از توابع *get_entities* و *get_relation* به عنوان ورودی می‌دهیم و خروجی را در آرایه‌هایی با نام‌های *entity_pairs* و *relations* قرار می‌دهیم. تصویر۶ کد مربوطه را به همراه نمونه‌هایی از موجودیت‌ها و روابط استخراج شده را نشان می‌دهد.

*Extracting entities using 'get_entities' function*

In [8]:
```python
# Extracting entities
entity_pairs = []
for i in tqdm(data["sentence"]):
    entity_pairs.append(get_entities(i))

print(entity_pairs[0:10])
```

```
100%|██████████| 4318/4318 [00:37<00:00, 114.51it/s]
[['connie', 'own'], ['later scream', 'distance'], ['christian', 'then elder'], ['temple', 'fire'], ['', 'outside cult him'], ['it', 'religious awakening'], ['c. mackenzie', 'craig cast'], ['later craig di francia', 'action cast'], ['sebastian maniscalco', 'later paul ben cast'], ['we', 'just film']]
```

*Extracting relations using 'get_relation' function*

In [10]:
```python
# Extracting relations
relations = [get_relation(i) for i in tqdm(data["sentence"])]
print(relations[0:10])
```

```
100%|██████████| 4318/4318 [00:41<00:00, 105.10it/s]
['decides to leave', 'heard in', 'paralyzed by', 'set on', 'wails with', "'s", 'joined', 'revealed to', 'revealed as', 'tried to make']
```

*تصویر ۶. کد حلقه مربوط به استخراج موجودیت‌ها و روابط آنها توسط توابع مربوطه.*

## ۴. ساخت گراف دانش[1]

هدف ساخت یک **گراف دانش جهت‌دار** با استفاده از موجودیت‌ها (فاعل‌ها و مفعول‌ها) و روابط استخراج‌شده از جملات است. این گراف، اطلاعاتی درباره‌ی چگونگی ارتباط موجودیت‌ها با یکدیگر را به‌صورت ساختاریافته نمایش می‌دهد. در ادامه، هر مرحله از ساخت گراف توضیح داده شده است.

### ۱.۴. مرحله اول: ایجاد یک *"DataFrame"*

ابتدا، داده‌های استخراج‌شده شامل **فاعل‌ها**، **مفعول‌ها** و **روابط** آن‌ها به‌صورت یک DataFrame با نام *"kg_df"* ساختاربندی می‌شوند. هر ردیف از *kg_df* نشان‌دهنده یک ارتباط بین دو موجودیت (فاعل و مفعول) با یک نوع رابطه مشخص است.

**ساختار *kg_df*** شامل سه ستون است:

۱) ***source*** (مبدا): فاعل یا موجودیت اصلی جمله که با get_entities استخراج شده است.

۲) ***target*** (مقصد): مفعول یا موجودیت ثانویه که با get_entities استخراج شده است.

۳) ***edge*** (یال): رابطه بین موجودیت‌ها که با get_relation استخراج شده است.

این ساختار به ما کمک می‌کند که به‌سادگی اطلاعات موردنیاز برای ساخت یک گراف جهت‌دار را در اختیار داشته باشیم.

### ۲.۴. مرحله دوم: ایجاد گراف جهت‌دار با استفاده از *"networkx"*

بعد از ساخت *kg_df*، از کتابخانه ***networkx*** برای ساخت **گراف دانش جهت‌دار (*G*)** استفاده می‌کنیم. گراف جهت‌دار به ما کمک می‌کند که مسیر و جهت ارتباطات بین موجودیت‌ها را نشان دهیم. در این راستا، ما از تابع *"from_pandas_edgelist"* (از کتابخانه *networkx*) برای انتقال مستقیم داده‌ها از *kg_df* به گراف استفاده می‌شود. تابع *"MultiDiGraph"* (از کتابخانه *networkx*) برای ایجاد ساختار گراف استفاده می‌شود که امکان داشتن چندین یال بین یک جفت گره را فراهم می‌کند.

کد ساخت گراف در تصویر۷ نشان داده شده است.

Constructing a Directed Knowledge Graph with Extracted Entities and Relations

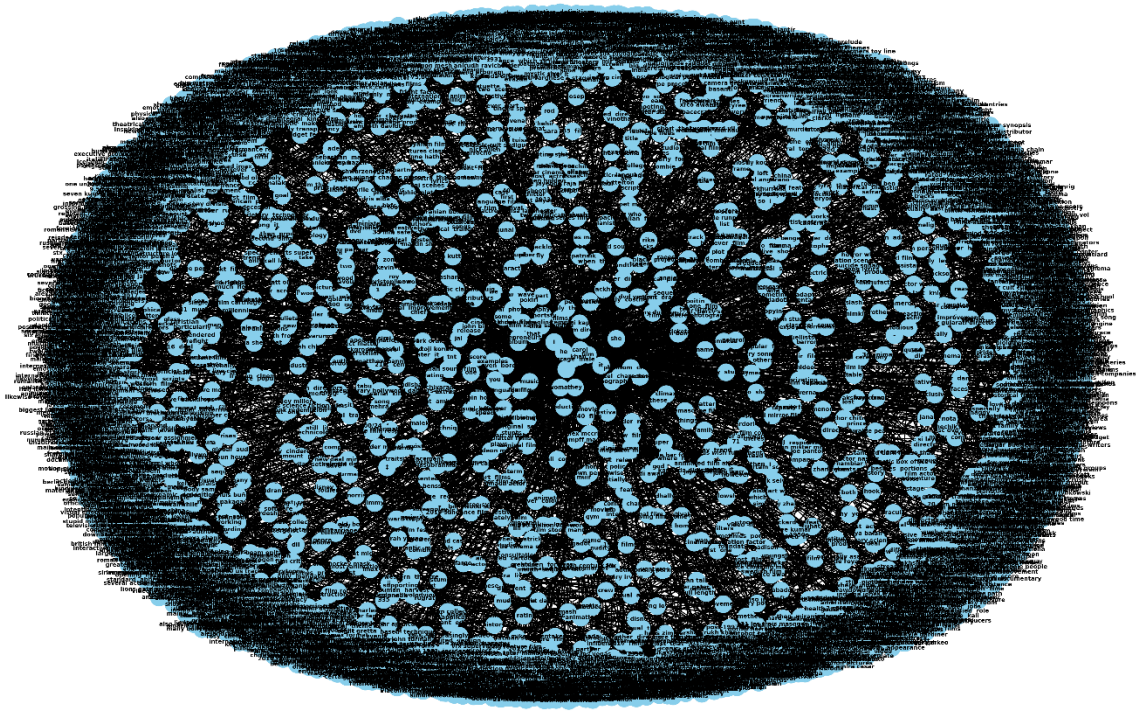
```
In [12]: # extract subject
source = [i[0] for i in entity_pairs]

# extract object
target = [i[1] for i in entity_pairs]

kg_df = pd.DataFrame({'source':source, 'target':target, 'edge':relations})

# create a directed-graph from the created dataframe "kg_df".
G=nx.from_pandas_edgelist(kg_df, "source", "target",
                          edge_attr=True, create_using=nx.MultiDiGraph())
```

*تصویر ۷. کد ساخت گراف دانش G.*

[1] Knowledge Graph

## ۵. رسم و نمایش گراف

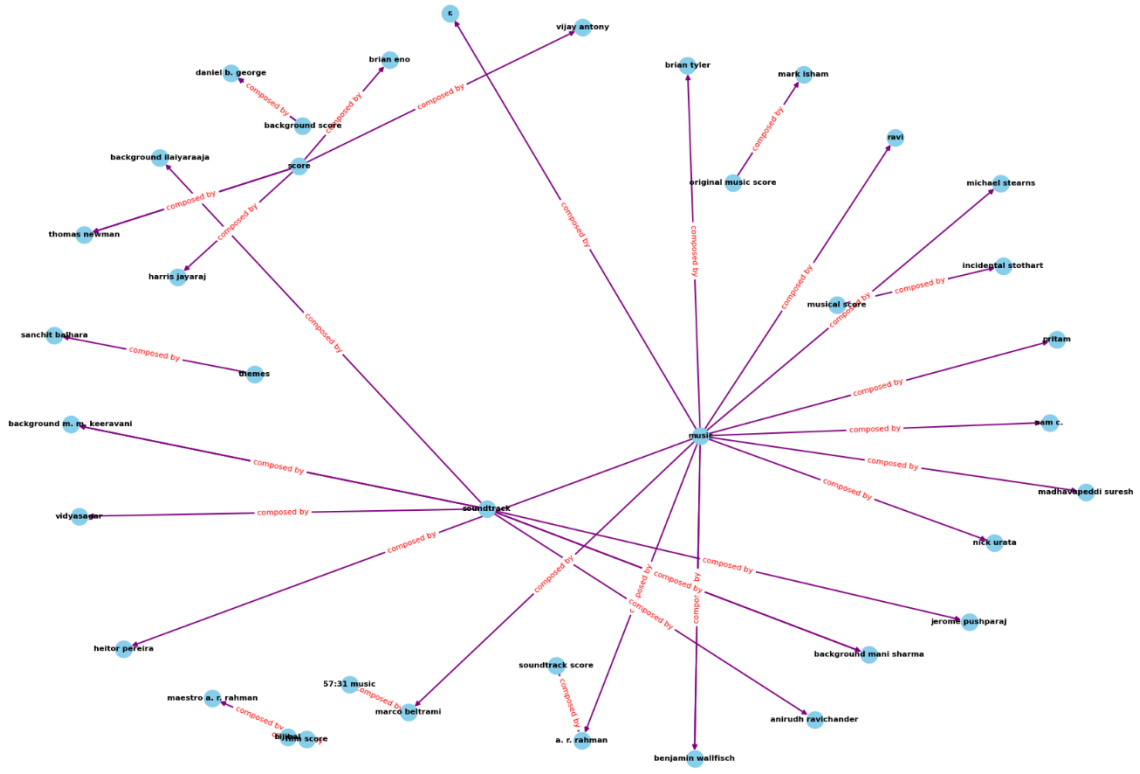
در این بخش، به مصورسازی و نمایش گرافیکی گراف می‌پردازیم. اکثر اوقات، نمایش همه موجودیت‌ها و روابط بین آنها خوب نیست زیرا گراف مصور شده از آنها به سختی قابل خواندن و درک است. این معمولا برای متن‌های بزرگ، به دلیل تعداد زیاد موجودیت و رابطه اتفاق می‌افتد. گراف (***G***) کامل ایجاد شده بالا از مجموعه داده‌های *wiki_sentences_v2* نیز بیش از حد شلوغ است که شکل نمایش آن در تصویر۸ آورده شده است. از این رو، ما در ادامه یک نمایش از گراف فیلتر شده (***filtered_G***) توسط رابطه خاص ***"composed by"***، با دو روش مختلف نمایش می‌دهیم.



***تصویر ۸ تصویر گراف (G) کامل ایجاد شده از مجموعه داده‌های wiki_sentences_v2 .***

### ۱.۵. فیلتر کردن گراف (*filtered_G*) و نمایش آن با استفاده از *networkx* و *matplotlib*

در این مرحله، از *networkx* و *matplotlib* برای فیلتر و نمایش گراف استفاده می‌کنیم. این روش به ما امکان می‌دهد که گراف را با فیلتر کردن روابط خاص (مانند*"composed by"*) نمایش دهیم و ویژگی‌های ظاهری گره‌ها و یال‌ها را تنظیم کنیم تا نمودار بصری واضح‌تر و خواناتری داشته باشیم. گراف نمایش داده شده از این روش در تصویر۹ قابل مشاهده می‌باشد.




*تصویر ۹. تصویر گراف فیلتر شده توسط matplotlib.*

**مراحل ساخت و نمایش *filtered_G*:**

- **ایجاد گراف فیلتر شده** : مشابه با ایجاد گراف *G* که در بخش ۲.۴ توضیح داده شد انجام می‌شود، با این تفاوت که اینبار روابط را محدود به "*composed by*" می‌کنیم. بدین ترتیب، گراف *filtered_G* که روابط بین موجودیت‌هایی که توسط رابطه *composed by* با هم مرتبط هستند ایجاد می‌شود. می‌توانیم گراف را محدود به یک یا چندین رابطه، فیلتر کنیم. این کار به ما امکان می‌دهد که روی روابط مهم و پرتکرار تمرکز کنیم.
- **تنظیم ویژگی‌های ظاهری گره‌ها و یال‌ها:** برای این منظور، از "***spring_layout***" برای تنظیم فاصله متناسب بین گره‌ها استفاده شد تا یال‌ها کمتر روی هم بیفتند و ساختار بهتری از گراف نمایش داده شود. برای تمایز بیشتر، گره‌ها با رنگ آبی آسمانی (*skyblue*) و یال‌ها با رنگ بنفش (*purple*) نمایش داده می‌شوند. اندازه فونت‌ها، ضخامت خطوط و وزن فونت‌ها تنظیم می‌شوند تا وضوح نمودار افزایش یابد. برچسب‌های یال‌ها به رنگ قرمز نمایش داده می‌شوند تا روابط بین گره‌ها مشخص‌تر شوند.
- **نمایش گراف:** در نهایت با استفاده از دستور ***plt.show()*** گراف، با نمایش تنظیم شده، رسم و نمایش داده می‌شود.

تصویر ۱۰ کد این مراحل را برای ایجاد گراف فیلتر شده و نمایش آن براساس تنظیمات مذکور نشان می‌دهد.

Visualize a sample graph with 'composed by' relationship

```
In [13]: # Filter edges with 'composed by' relationship and create a directed multigraph
filtered_G = nx.from_pandas_edgelist(kg_df[kg_df['edge'] == "composed by"], "source", "target",
                                     edge_attr=True, create_using=nx.MultiDiGraph())

# Visualize the graph
plt.figure(figsize=(18, 13))
pos = nx.spring_layout(filtered_G, k=0.3)  # Adjust k to control node spacing
nx.draw(filtered_G, pos, with_labels=True, node_color="skyblue", font_size=8, font_weight="bold", edge_color="purple", width=1.5)

# Extract edge labels from 'edge' attribute or any specific attribute in kg_df
edge_labels = nx.get_edge_attributes(filtered_G, 'edge')  # Adjust to the correct attribute name
nx.draw_networkx_edge_labels(filtered_G, pos, edge_labels=edge_labels, font_size=8, font_color="red")

# Save and display the figure
plt.savefig("knowledge_graph_Newfiltered995.png", format="PNG")
plt.show()
```

***تصویر ۱۰. فیلتر کردن گراف (filtered_G) و نمایش آن با استفاده از networkx و matplotlib.***

### ۲.۵. نمایش گراف "filtered_G" با استفاده از "pyvis"

روش دوم نمایش گراف، استفاده از **کتابخانه *pyvis*** است. این کتابخانه امکان نمایش گراف را به‌صورت **تعاملی** فراهم می‌کند که می‌تواند در مرورگر مشاهده شود. *pyvis* به ما اجازه می‌دهد که با کلیک و حرکت گره‌ها و یال‌ها، بهتر ساختار گراف را بررسی کنیم و جزئیات بیشتری را مشاهده نماییم.

**مراحل ساخت و نمایش گراف تعاملی با *pyvis*:**

۱) **ایجاد گراف *pyvis*** : ابتدا یک شبکه *pyvis* با استفاده از تابع *Network* از کتابخانه *pyvis* ایجاد می‌کنیم و گره‌ها و یال‌های گراف فیلتر شده (filtered_G) را به آن اضافه می‌کنیم.

۲) **سفارشی‌سازی گره‌ها و یال‌ها:** مشابه به روش پیشین می‌توانیم رنگ و اندازه گره‌ها را تنظیم کنیم. برای مثال، گره‌ها را با رنگ آبی نمایش می‌دهیم و اندازه آن‌ها را بر اساس اهمیت یا تعداد یال‌های متصل تنظیم می‌کنیم. برای نمایش جهت روابط، روی یال‌ها فلش قرار می‌دهیم.

۳) **تنظیمات ظاهری گراف:** با استفاده از تنظیمات *options*، ویژگی‌های دیگری مانند اندازه فونت‌ها و رنگ یال‌ها را می‌توانیم سفارشی کنیم.

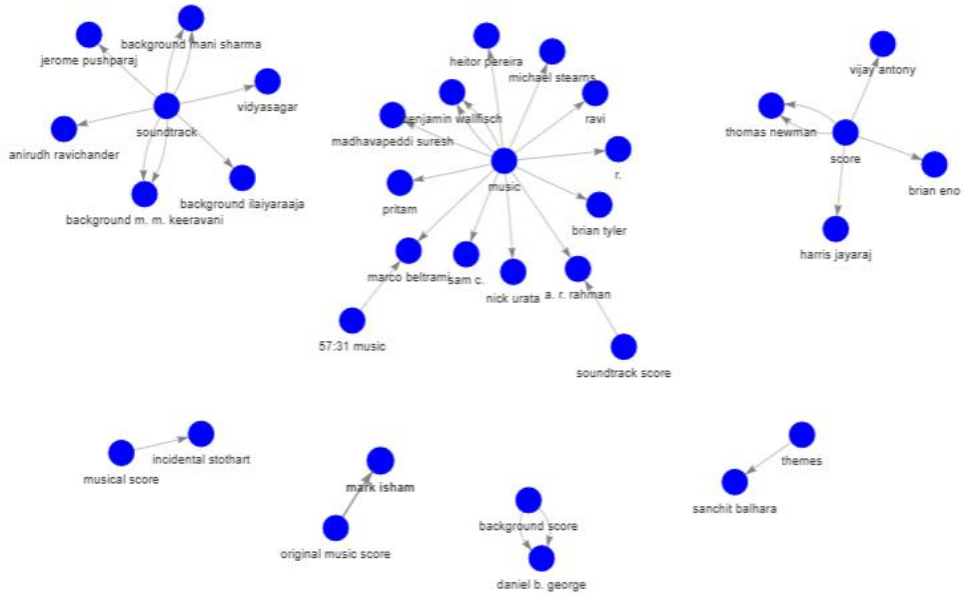
۴) **ذخیره و نمایش گراف:** گراف را به عنوان یک فایل *HTML* ذخیره می‌کنیم و سپس می‌توانیم آن را در یک مرورگر باز کنیم.

کد برای ساخت و نمایش گراف با pyvis در تصویر ۱۱ نشان داده شده است. همچنین، گراف نمایش داده شده از این روش در تصویر ۱۲ قابل مشاهده می‌باشد.

***Using pyvis to advanced visualization***

In [18]:
```
from pyvis.network import Network
# Initialize a NetworkX directed graph and pyvis Network object
net = Network(notebook=True, directed=True)

# Customize node appearance
for node in filtered_G.nodes():
    net.add_node(node, label=node, color="blue", size=19)

# Add edges with arrows to indicate direction
for edge in filtered_G.edges():
    net.add_edge(edge[0], edge[1], arrowStrikethrough=True)

# Generate the graph and display it inline
net.set_options("""
var options = {
  "nodes": {
    "font": {
      "size": 19
    }
  },
  "edges": {
    "arrows": {
      "to": {
        "enabled": true
      }
    },
    "color": {
      "color": "gray"
    }
  }
}
""")
net.show("directed_graph.html")
```

***تصویر ۱۱. کد ساخت و نمایش گراف با pyvis.***




***تصویر ۱۲. تصویر گراف "filtered_G" با استفاده از "pyvis".***

### ۳.۵. مقایسه نمایش با *matplotlib* و *pyvis*

- ***networkx + matplotlib***: این روش یک نمودار ایستا و ساده را فراهم می‌کند که برای نمایش سریع و ایجاد تصاویر برای گزارش‌ها مفید است. ویژگی‌های ظاهری را به‌خوبی می‌توان تنظیم کرد، اما تعامل با گراف محدود است.

- ***Pyvis***: این روش یک نمایش تعاملی از گراف ارائه می‌دهد که برای تحلیل‌های عمیق و مشاهده‌ی جزئیات به کار می‌آید. امکان جابجایی گره‌ها، زوم و تعامل با گراف، بررسی ساختار گراف را آسان‌تر می‌کند. با این حال، برای نمایش‌های ثابت یا درج در گزارش‌ها، مناسب نیست.

با این دو روش، می‌توانید گراف دانش خود را هم به‌صورت ایستا و هم به‌صورت تعاملی نمایش دهید و از هرکدام بسته به نیاز استفاده کنید.

## ۶. اتصال به *Neo4j* و بارگذاری گراف دانش در آن

اتصال به *Neo4j* و بارگذاری گراف دانش در آن شامل چندین مرحله است. در این بخش، با استفاده از اطلاعات دسترسی به پایگاه داده و ایجاد اتصالات لازم، موجودیت‌ها (گره‌ها) و روابط (یال‌ها) را در *Neo4j* ذخیره می‌کنیم. هدف از این مراحل، ساخت یک **گراف دانش پایدار** در *Neo4j* است که امکان اجرای کوئری‌ها و تحلیل‌های بیشتر روی گراف دانش را فراهم می‌کند. کد مربوط به اتصال و انتقال گراف دانش به *Neo4j* در تصویر۱۳ نشان داده شده است. در ادامه هر یک از این دو مرحله توضیح داده می‌شود.

### ۱.۶. اتصال به*Neo4j*

برای اتصال به پایگاه داده‌ی*Neo4j* ، ابتدا باید **آدرس سرور**، **نام کاربری** و **رمز عبور** مشخص شود. این اطلاعات به کمک کلاس ***GraphDatabase*** در *Neo4j* استفاده می‌شوند تا یک اتصال برقرار شود.

- ***Uri***: آدرس سرور Neo4j (معمولاً به صورت bolt://localhost:7687 برای پایگاه داده محلی).
- ***username*** و ***password***: نام کاربری و رمز عبور برای دسترسی به پایگاه داده.

**Connect to Neo4j and Create Graph in Neo4j**

In [19]:
```
# Connect to Neo4j
uri = "bolt://localhost:7687"
username = "neo4j"
password = "Reza1371"

driver = GraphDatabase.driver(uri, auth=(username, password))
session = driver.session()
```

In [20]:
```
# Create index on the name property for optimization
session.run("CREATE INDEX IF NOT EXISTS FOR (n:Entity) ON (n.name)")

# Function to create nodes and relationships in Neo4j
def create_graph_in_neo4j(G):
    # First, create all nodes
    for node in G.nodes():
        session.run("MERGE (n:Entity {name: $name})", {"name": node})

    # Next, create all relationships without Cartesian product
    for source, target, data in G.edges(data=True):
        session.run(
            "MATCH (a:Entity {name: $source}) "
            "MATCH (b:Entity {name: $target}) "
            "MERGE (a)-[r:RELATED_TO {type: $label}]->(b)",
            {"source": source, "target": target, "label": data['edge']}
        )

create_graph_in_neo4j(G)
```

*تصویر ۱۳. کد اتصال و انتقال گراف دانش به Neo4j.*

این اطلاعات به تابع ***driver()*** از کلاس *GraphDatabase* داده می‌شود و سپس یک **شیء اتصال *(driver)*** ایجاد می‌شود که امکان تعامل با پایگاه داده را فراهم می‌کند. کد این بخش در تصویر۱۳ در سلول۱۹ نشان داده شده است.

**نکته** :باید مطمئن شوید که سرویس *Neo4j* در حال اجراست و اطلاعات دسترسی به درستی تنظیم شده‌اند.

### ۲.۶. ایجاد یا انتقال گراف به *Neo4j*

پس از برقراری اتصال، باید گراف را به *Neo4j* منتقل کنیم. برای این کار، یک تابع به نام *create_graph_in_neo4j* ایجاد می‌کنیم که موجودیت‌ها و روابط را از گراف *networkx* گرفته و آن‌ها را در *Neo4j* وارد می‌کند. هدف این تابع این است که **گره‌ها و یال‌ها را در *Neo4j* ایجاد کند** و از تکرار موارد جلوگیری نماید.

**جزئیات تابع *create_graph_in_neo4j*:**

۱) **ایجاد گره‌ها (موجودیت‌ها)**: تابع ابتدا تمامی **گره‌ها** را از گراف *networkx* دریافت می‌کند. با استفاده از دستور ***MERGE***، هر گره به پایگاه داده اضافه می‌شود. دستور *MERGE* از ایجاد موارد تکراری جلوگیری می‌کند. یعنی اگر گره‌ای با نام مشابه در پایگاه داده وجود داشته باشد، دوباره ایجاد نمی‌شود. این ویژگی از اضافه شدن چندباره‌ی گره‌ها جلوگیری می‌کند و باعث بهینه‌سازی گراف می‌شود.

۲) **ایجاد یال‌ها (روابط)**: پس از ایجاد گره‌ها، تابع به سراغ **روابط** (یال‌ها) می‌رود. برای هر رابطه، ابتدا گره‌های مبدا و مقصد (*source* و *target*) انتخاب می‌شوند. سپس، رابطه‌ی بین این دو گره با دستور *MERGE* ایجاد می‌شود. رابطه دارای یک **ویژگی *(attribute)*** به نام type است که نوع رابطه (مانند *"directed by"* یا *"produced by"*) را نشان می‌دهد.

**توضیح دستورات Cypher در تابع:**

۱) ***MERGE (n:Entity {name: $name})***: این دستور یک گره با برچسب Entity و ویژگی name ایجاد می‌کند. اگر گره‌ای با همین نام وجود داشته باشد، دوباره ایجاد نمی‌شود.

۲)
```
MATCH (a:Entity {name: $source})
MATCH (b:Entity {name: $target})
MERGE (a)-[r:RELATED_TO {type: $label}]->(b)
```

این دستور ابتدا گره‌های مبدا و مقصد را پیدا می‌کند (با استفاده از ***MATCH***)، سپس یک رابطه جهت‌دار *(RELATED_TO)* بین آن‌ها با ویژگی *type* ایجاد می‌کند. *type* همان نوع رابطه است که از *data['edge']* گرفته می‌شود.

## ۷. اجرای هفت کوئری(*Query*) *Cypher* از *Neo4j*

اجرای کوئری‌های ***Cypher*** در *Neo4j* به ما این امکان را می‌دهد که اطلاعات گراف دانش را با جزئیات بیشتر تحلیل کنیم. در این بخش، به هر کوئری و هدف آن پرداخته و نحوه استفاده از آن برای استخراج اطلاعات کلیدی را توضیح خواهیم داد.

### ۱.۷. شمارش کل موجودیت‌ها

این کوئری، تعداد کل گره‌هایی که به‌عنوان موجودیت (با برچسب *Entity*) ذخیره شده‌اند را به دست می‌آورد. این اطلاعات به ما کمک می‌کند تا اندازه و مقیاس گراف دانش خود را درک کنیم و بررسی کنیم که آیا تمامی موجودیت‌ها به‌درستی وارد پایگاه داده شده‌اند یا خیر. این کوئری در تصویر۱۴ به همراه خروجی در سلول ۲۱ قابل مشاهده است.

**توضیح:**

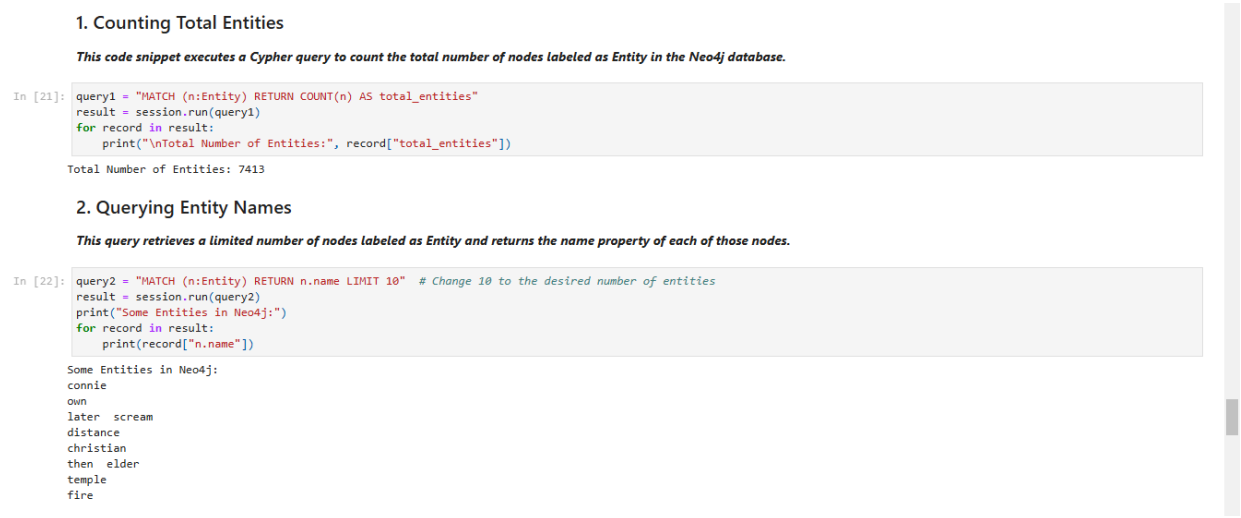
- دستور *MATCH (n:Entity)* به دنبال تمام گره‌هایی با برچسب *Entity* می‌گردد.
- دستور *COUNT(n)* تعداد این گره‌ها را برمی‌گرداند و نتیجه را به نام *total_entities* (که مقدار۷۴۱۳ می‌باشد) نمایش می‌دهد. این نشان میدهد که گراف ایجاد شده شامل ۷۴۱۳ موجودیت یا گره است.

### ۲.۷. کوئری نام موجودیت‌ها

این کوئری، نام تعداد مشخصی از موجودیت‌ها را بازیابی می‌کند. این کار برای بررسی داده‌ها و آشنایی با ساختار کلی گراف مفید است که در تصویر۱۴ به همراه خروجی در سلول۲۲ قابل مشاهده است.

**توضیح:**

- دستور *MATCH (n:Entity)* همه گره‌های دارای برچسب *Entity* را انتخاب می‌کند.




*تصویر ۱۴. کوئری ۱ (تعداد کل موجودیت‌ها) و کوئری ۲ (نام ۱۰ تا ازموجودیت‌ها).*

- با استفاده از *RETURN n.name LIMIT 10*، نام ۱۰ گره برگردانده می‌شود که به بررسی سریع و دیدن نام موجودیت‌ها کمک می‌کند. به عنوان نمونه، اجرای این دستور، *connie* و *Christian* را به عنوان موجودیت نشان می‌دهد.

### ۳.۷. شمارش کل روابط

این کوئری تعداد کل روابط موجود در گراف را محاسبه می‌کند. دانستن تعداد روابط، ایده‌ای از تراکم گراف و تعداد ارتباطات بین موجودیت‌ها می‌دهد. این کوئری در سلول۲۳ از تصویر۱۵ نشان داده شده است.

**توضیح:**

- دستور *MATCH ()-[r]->()* به دنبال تمام یال‌ها (روابط) در گراف می‌گردد.
- *COUNT(r)* تعداد کل روابط را برمی‌گرداند و نتیجه را به عنوان *total_relationships* نمایش می‌دهد. خروجی ۵۸۶۵ بیانگر تعداد کل روابط موجود بین تمام موجودیت‌ها می‌باشد.

### ۴.۷. کوئری روابط موجودیت‌ها

این کوئری نمونه‌ای از روابط بین موجودیت‌ها را برمی‌گرداند، به طوری که مبدا، نوع رابطه، و مقصد هر رابطه مشخص شود. این کار برای بررسی روابط موجود بین موجودیت‌ها و مشاهده جزئیات آن‌ها مفید است. این کوئری در سلول۲۴ از تصویر۱۵ نشان داده شده است.

3. Counting Total Relationships

***This query counts the total number of relationships.***

In [23]:
```
# Query to count total number of relationships
query3 = "MATCH ()-[r]->() RETURN COUNT(r) AS total_relationships"
result = session.run(query3)
for record in result:
    print("Total Number of Relationships:", record["total_relationships"])
```

```
Total Number of Relationships: 5865
```

4. Querying Entity Relationships

***This Cypher query retrieves and displays 10 random sample of relationships between entities.***

In [24]:
```
query4 = "MATCH (a:Entity)-[r:RELATED_TO]->(b:Entity) RETURN a.name AS source, r.type AS relationship, b.name AS target ORDER BY rand() LIMIT 10"
result = session.run(query4)
print("\nRelationships in Neo4j:")
for record in result:
    print(record["source"], record["relationship"], record["target"])
```

```
Relationships in Neo4j:
 pattinson vs. cam gigandet
zak penn rewrote then it
steven derek dana françois matthew robert catrini cast in undisclosed undisclosed roles
it rewritten with different  angle
traits are common live action
it was animated  3d
awards continued to least  1994
it was just sync
bryan hirota served as vfx supervisor
it sung by gagan himself
```

***تصویر۱۵. کوئری ۳ (تعداد کل روابط) و کوئری ۴ (روابط موجودیت‌ها).***

**توضیح:**

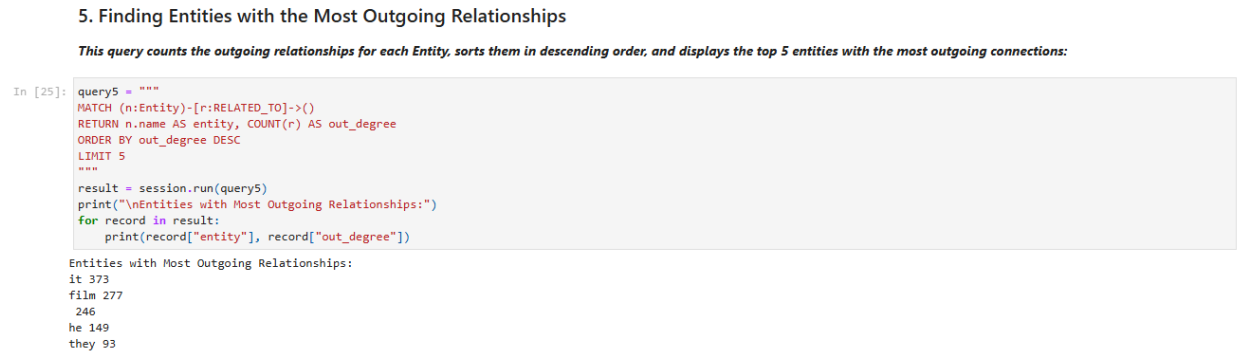
- دستور *MATCH (a:Entity)-[r:RELATED_TO]->(b:Entity)* روابط بین گره‌های دارای برچسب ***Entity*** را که از نوع *RELATED_TO* هستند پیدا می‌کند.
- با استفاده از *ORDER BY rand() LIMIT 10*، ده رابطه تصادفی برگردانده می‌شود. به عنوان مثال از خروجی: ***connie*** به عنوان موجودیت مبدا به موجودیت مقصد ***town*** توسط رابطه ***decides to leave*** متصل می‌باشد. از این رابطه ما نتیجه می‌گیریم که این شخص تصمیم به ترک شهر دارد.

### ۵.۷. کوئری استخراج موجودیت‌هایی با بیشترین روابط

این کوئری موجودیت‌هایی که بیشترین روابط را دارند شناسایی می‌کند. چنین موجودیت‌هایی معمولاً نقش کلیدی و مرکزی در گراف دارند و ممکن است نمایانگر مفاهیم اصلی و پرتکرار باشند. تصویر۱۶ کد این کوئری را به همراه نتیجه نشان می‌دهد.

**توضیح:**

- *MATCH (n:Entity)-[r:RELATED_TO]->()* موجودیت‌هایی با برچسب ***Entity*** را که دارای روابط خروجی هستند، پیدا می‌کند.
- *COUNT(r)* تعداد روابط خروجی هر موجودیت را محاسبه می‌کند و نتیجه به ترتیب نزولی توسط دستور *ORDER BY out_degree DESC* مرتب می‌شود.
- در نهایت با دستور *LIMIT 5* این کوئری ۵ موجودیت با بیشترین تعداد روابط خروجی را برمی‌گرداند.




*تصویر ۱۶. کوئری۵: استخراج موجودیت‌هایی با بیشترین روابط.*

### ۶.۷. کوئری کوتاه‌ترین مسیر بین دو موجودیت

این کوئری کوتاه‌ترین مسیر بین دو موجودیت مشخص را در گراف پیدا می‌کند. استفاده از این کوئری در تحلیل ارتباطات بین دو موجودیت یا بررسی وجود ارتباط بین آن‌ها مفید است. تصویر۱۷ کد این کوئری را به همراه نتیجه نشان می‌دهد.

### 6. Find the Shortest Path Between Two Entities

***This query finds the shortest path (if any) between two specific entities:***

In [26]:
```
query6 = """
MATCH (start:Entity {name: $start_name}), (end:Entity {name: $end_name}),
      p = shortestPath((start)-[*]-(end))
RETURN [n IN nodes(p) | n.name] AS path
"""

SourceEntityName = "mani sharma"
TargetEntityName = "nick urata"
params = {"start_name": SourceEntityName, "end_name": TargetEntityName}
result = session.run(query6, params)
print("\nShortest Path Between Two Entities:")
for record in result:
    print(record["path"])
```

```
Received notification from DBMS server: {severity: WARNING} {code: Neo.ClientNotification.Statement.UnboundedVariableLengthPatternWarning} {category: } {title: The provided pattern is unbounded, consider adding an upper limit to the number of node hops.} {description: Using shortest path with an unbounded pattern will likely result in long execution times. It is recommended to use an upper limit to the number of node hops in your pattern.} {position: line: 3, column: 24, offset: 98} for query: '\nMATCH (start:Entity {name: $start_name}), (end:Entity {name: $end_name}),\n      p = shortestPath((start)-[*]-(end))\nRETURN [n IN nodes(p) | n.name] AS path\n'
Received notification from DBMS server: {severity: WARNING} {code: Neo.ClientNotification.Statement.CartesianProductWarning} {category: } {title: This query builds a cartesian product between disconnected patterns.} {description: If a part of a query contains multiple disconnected patterns, this will build a cartesian product between all those parts. This may produce a large amount of data and slow down query processing. While occasionally intended, it may often be possible to reformulate the query that avoids the use of this cross product, perhaps by adding a relationship between the different parts or by using OPTIONAL MATCH (identifier is: (end))} {position: line: 2, column: 1, offset: 1} for query: '\nMATCH (start:Entity {name: $start_name}), (end:Entity {name: $end_name}),\n      p = shortestPath((start)-[*]-(end))\nRETURN [n IN nodes(p) | n.name] AS path\n'
Shortest Path Between Two Entities:
['mani sharma', 'soundtrack', 'nick urata']
```

***تصویر ۱۷. کوئری۶: کوتاه‌ترین مسیر بین دو موجودیت.***

**توضیح:**

- *MATCH (start:Entity {name: $start_name}), (end:Entity {name: $end_name})* دو موجودیت مشخص را پیدا می‌کند.
- *shortestPath((start)-[*]-(end))* کوتاه‌ترین مسیر بین این دو موجودیت را محاسبه می‌کند.
- *RETURN [n IN nodes(p) | n.name] AS path* مسیر به‌دست‌آمده را به صورت لیستی از نام گره‌ها برمی‌گرداند.

به عنوان مثال، خروجی *['mani sharma', 'soundtrack', 'nick urata']* نشان می‌دهد که ***mani*** و ***nick*** از طریق موجودیت ***'soundtrack'*** با هم مرتبط هستند.

### ۷.۷. کوئری جفت‌های موجودیت با فراوانی روابط

این کوئری، جفت موجودیت‌ها را بر اساس نوع رابطه شناسایی می‌کند و تعداد دفعات هر رابطه را نمایش می‌دهد. این تحلیل به ما کمک می‌کند تا متداول‌ترین جفت‌های مرتبط و نوع روابط بین آن‌ها را شناسایی کنیم. تصویر۱۸ این کد این کوئری را به همراه نتیجه نشان می‌دهد.

**توضیح:**

- *MATCH (e1)-[r]->(e2)* همه جفت‌های موجودیت و روابط آن‌ها را پیدا می‌کند.
- *COUNT(*) AS frequency* تعداد دفعات هر نوع رابطه بین جفت‌های موجودیت را محاسبه می‌کند.
- *ORDER BY relationship, frequency DESC LIMIT 10* نتایج را بر اساس نوع رابطه و فراوانی به ترتیب نزولی مرتب کرده و ۱۰ جفت متداول را برمی‌گرداند.

### 7. Top Entity Pairs by Relationship Type with Frequency Count

***This query can reveal which entities are frequently connected and the types of relationships that are dominant between them.***

```
In [27]: # Advanced query to find the most common entity pairs per relationship type
query = """
MATCH (e1)-[r]->(e2)
RETURN type(r) AS relationship, e1.name AS entity1, e2.name AS entity2, COUNT(*) AS frequency
ORDER BY relationship, frequency DESC
LIMIT 10
"""
result = session.run(query)

print("\nMost Common Entity Pairs per Relationship Type:")
for record in result:
    print(f"Relationship: {record['relationship']}, Entity1: {record['entity1']}, Entity2: {record['entity2']}, Frequency: {record['frequency']}")
```

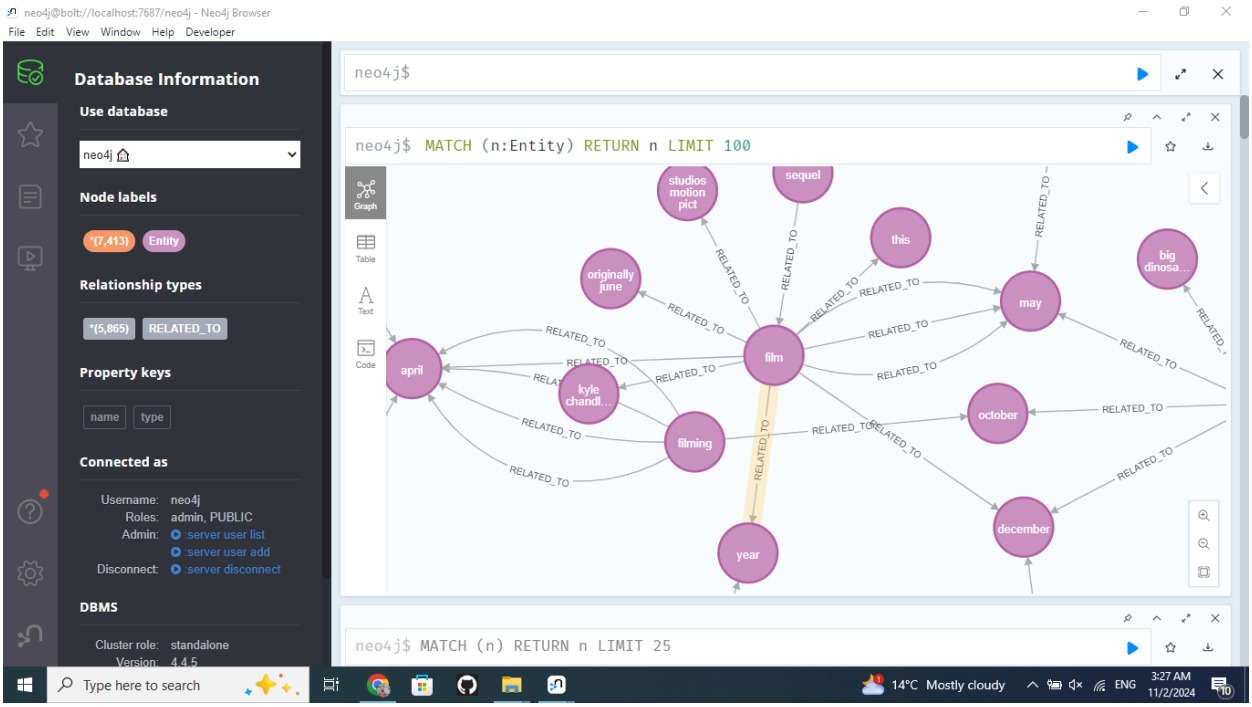
```
Most Common Entity Pairs per Relationship Type:
Relationship: RELATED_TO, Entity1: , Entity2: , Frequency: 33
Relationship: RELATED_TO, Entity1: i, Entity2: it, Frequency: 9
Relationship: RELATED_TO, Entity1: you, Entity2: me, Frequency: 5
Relationship: RELATED_TO, Entity1: film, Entity2: march, Frequency: 5
Relationship: RELATED_TO, Entity1: she, Entity2: him, Frequency: 4
Relationship: RELATED_TO, Entity1: principal photography, Entity2: august, Frequency: 4
Relationship: RELATED_TO, Entity1: filming, Entity2: april, Frequency: 4
Relationship: RELATED_TO, Entity1: principal  photography, Entity2: august, Frequency: 4
Relationship: RELATED_TO, Entity1: film, Entity2: may, Frequency: 3
Relationship: RELATED_TO, Entity1: film, Entity2: , Frequency: 3
```

```
In [ ]:
```

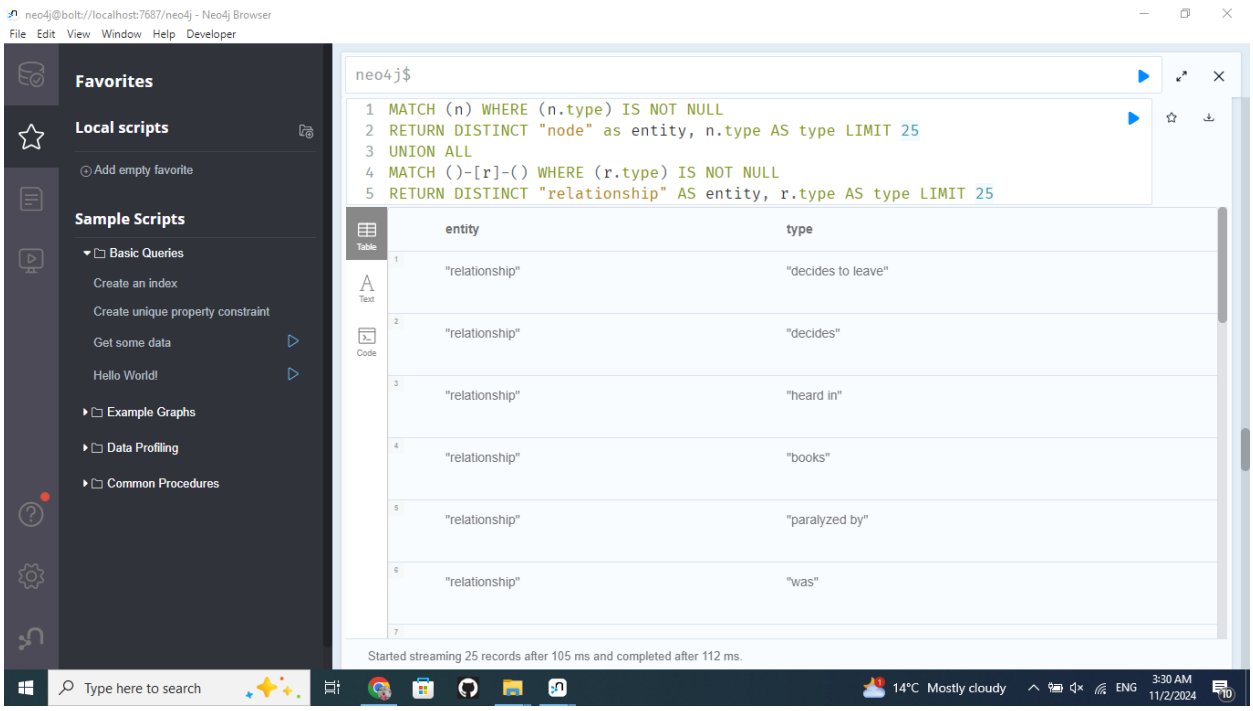
*تصویر ۱۸. کوئری۷: جفت‌های موجودیت با فراوانی روابط.*

## ۸. مشاهده گراف و انجام کوئری در *Neo4j Browser*

ما در این تمرین از طریق *Jupyter Notebook* گراف را مشاهده نمودیم و با اتصال آن به *Neo4j*، کوئری‌هایی بر روی آن انجام دادیم. ولی میتوان پس از انتقال گراف دانش به *Neo4j* به صورت مستقیم از طریق خود آن گراف ایجاد شده را مشاهده کرد و کوئری ها را در آن انجام داد و نتایج کوئری را هم به صورت گراف و هم به صورت نتیجه جدولی مشاهده نمود. در ادامه تصاویری از مشاهده گراف و کوئری در *Neo4j Browser* قرار داده شده است.




*تصویر 19. تصویری از نمایش گراف محدود شده به تعداد ۱۰۰ موجودیت در Neo4j.*




*تصویر ۲۰. تصویری از اجرای یک کوئری در Neo4j برای استخراج انواع روابط با محدودیت نمایش ۲۵ نوع.*

## ۹. جمع‌بندی و نتیجه‌گیری

در این تمرین، هدف ایجاد یک **گراف دانش** از مجموعه داده‌ای متنی و تحلیل ساختار آن با استفاده از ابزارهای پردازش زبان طبیعی و گرافی بود. **کتابخانه‌های مورد نیاز** مانند *spaCy* و *networkx* برای استخراج موجودیت‌ها و روابط و همچنین ساخت گراف مورد استفاده قرار گرفتند. **موجودیت‌ها و روابط** از داده‌های متنی استخراج و در قالب **گراف جهت‌دار** ساختاربندی شدند. برای نمایش گراف از *matplotlib* و *pyvis* استفاده شد که امکان مشاهده گراف به صورت ایستا و تعاملی را فراهم کردند.

سپس، گراف دانش در **پایگاه داده *Neo4j*** بارگذاری شد. به کمک کوئری‌های *Cypher*، امکان اجرای تحلیل‌های مختلف فراهم شد، مانند شمارش موجودیت‌ها و روابط، یافتن موجودیت‌های کلیدی و شناسایی مسیرهای ارتباطی بین موجودیت‌ها. این کوئری‌ها اطلاعات ارزشمندی از ساختار و نحوه ارتباط مفاهیم با یکدیگر ارائه دادند و نشان دادند که کدام مفاهیم به صورت پرتکرار و محوری در گراف دانش ظاهر می‌شوند.

این فرآیند به ما نشان داد که **گراف دانش** ابزاری مؤثر برای سازماندهی و تجزیه و تحلیل داده‌های متنی است. استفاده از *Neo4j* و *Cypher* به ما امکان داد تا گراف را به صورت پایدار ذخیره کرده و تحلیل‌های پیشرفته‌تری بر روی آن انجام دهیم. این رویکرد نه تنها ساختار دانش را به شکلی واضح نمایش می‌دهد، بلکه می‌تواند به عنوان مرجعی برای استخراج روابط و الگوهای نهفته در داده‌ها مورد استفاده قرار گیرد.

این پروژه می‌تواند در آینده با استفاده از داده‌های پیچیده‌تر و تکنیک‌های پردازش پیشرفته‌تر توسعه یابد تا الگوهای پنهان بیشتری کشف و تحلیل شوند.

## منابع و مراجع

برای انجام این تمرین و توسعه گراف دانش از منابع زیر استفاده شده است:


### ۱. کتابخانه‌های اصلی:

**Pandas**: McKinney, W. (2010). *Data Structures for Statistical Computing in Python*. Proceedings of the 9th Python in Science Conference. DOI: https://doi.org/10.25080/Majora-92bf1922-00a, Available at: https://pandas.pydata.org/

**spaCy**: Explosion AI (Matthew Honnibal and Ines Montani). *spaCy: Industrial-Strength Natural Language Processing*. Available at: https://spacy.io/

**networkx**: Hagberg, A., Schult, D., & Swart, P. (2008). *Exploring Network Structure, Dynamics, and Function using NetworkX*. Proceedings of the 7th Python in Science Conference. DOI: https://doi.org/10.25080/TCWV9851, Available at: https://networkx.github.io/

**matplotlib**: Hunter, J. D. (2007). *Matplotlib: A 2D Graphics Environment*. Computing in Science & Engineering, 9(3), 90–95. DOI**:** 10.1109/MCSE.2007.55, Available at: https://matplotlib.org/

**pyvis**: Pyvis Network Visualization Library. Documentation available at: https://pyvis.readthedocs.io/

**neo4j**: Neo4j Inc. *Neo4j Graph Database Platform*. Available at: https://neo4j.com/

### ۲. مستندات نرم‌افزار و کوئری‌های Neo4j:

Neo4j Documentation. (2023). *Cypher Query Language*. Available at: https://neo4j.com/docs/cypher-manual/current/

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph Databases*. O'Reilly Media, Inc.

### ۳. دیگر منابع کمک آموزشی

Hami Ismail. Relationship Extraction from Any Web Articles using spaCy and Jupyter Notebook in 6 Steps. Medium. Available at: https://hami-asmai.medium.com/relationship-extraction-from-any-web-articles-using-spacy-and-jupyter-notebook-in-6-steps-4444ee68763f