Datenjournalismus (Teil 2)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

9. Oktober 2020

Datenjournalismus

2. Was sind die Datenmanagement und -analyseprobleme dahinter?

letzte Woche:

Frage 1

Wie verwalten wir graphische Daten?

relationales Modell

Frage 2

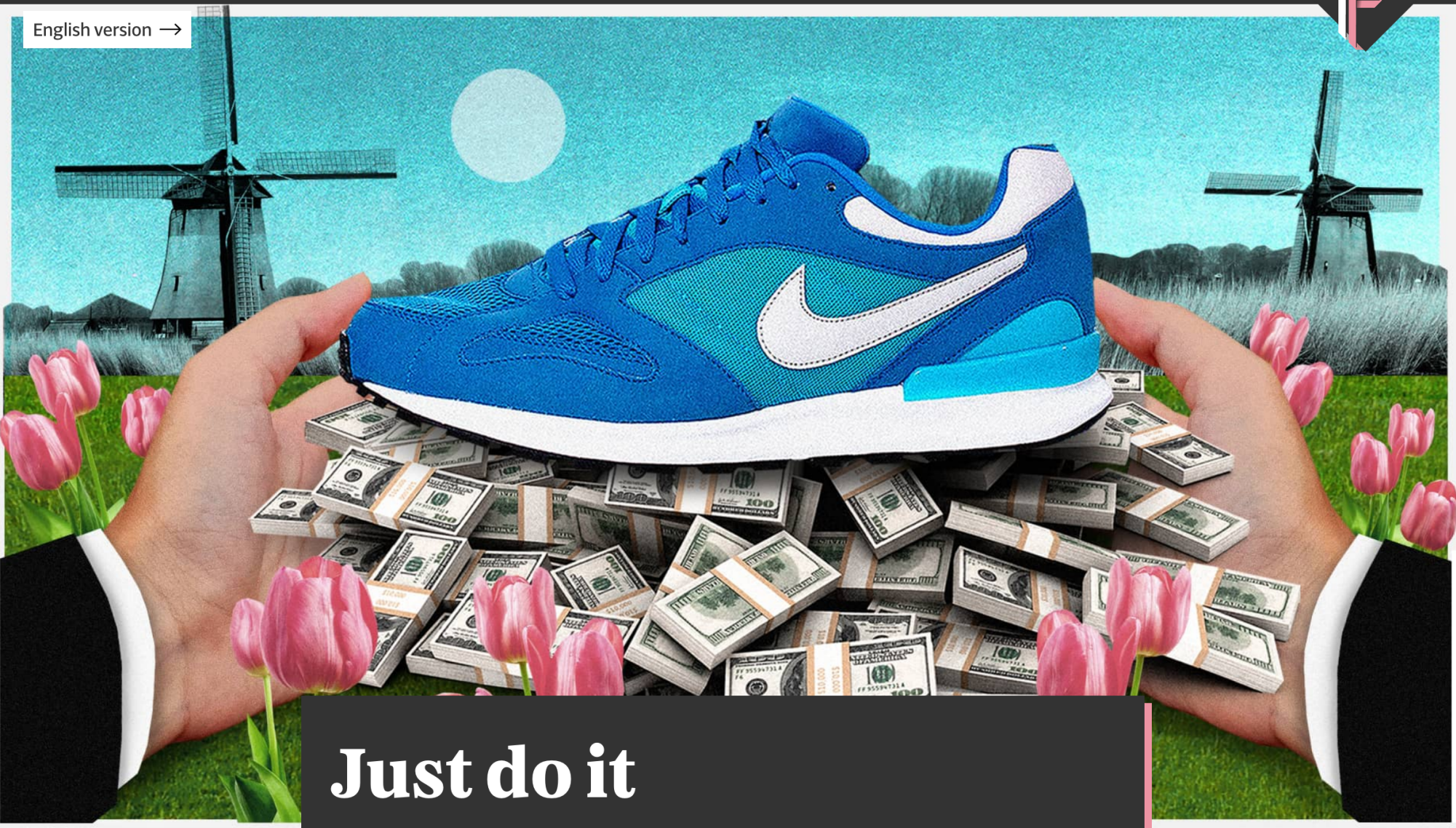
Wie stellen wir Anfragen an diese Daten?

Cypher

Datenjournalismus

4. Transfer der Grundlagen auf die konkrete Anwendung

English version →



Just do it

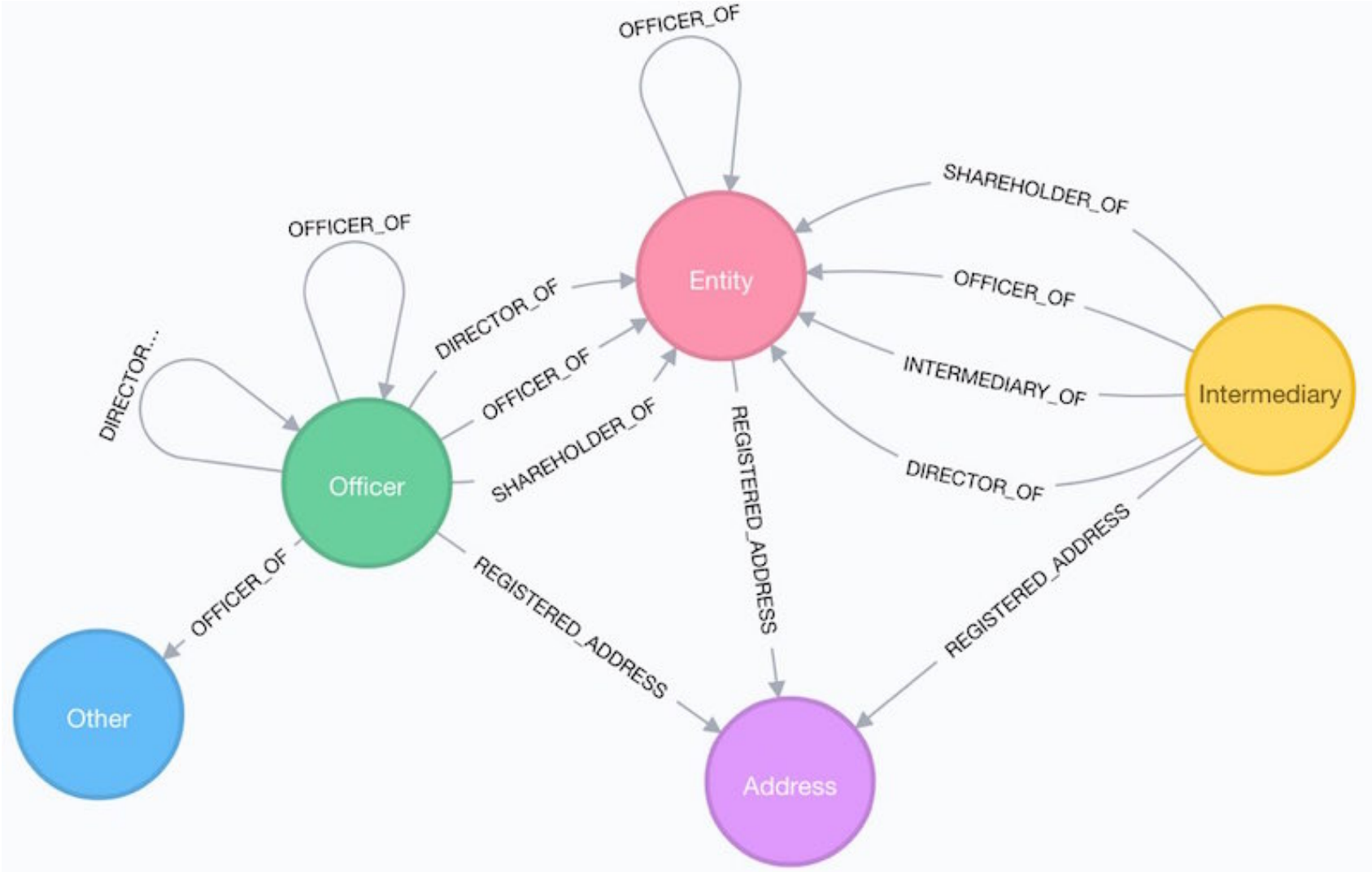
Nike ist bei sportlichen Wettkämpfen auf der ganzen Welt präsent. In einer Disziplin aber ist das Unternehmen selbst kaum zu schlagen - im Vermeiden

https://projekte.sueddeutsche.de/paradisepapers/wirtschaft/nike-und-die-niederlande-prellen-den-deutschen-staat-e116625/

Die Paradise Papers

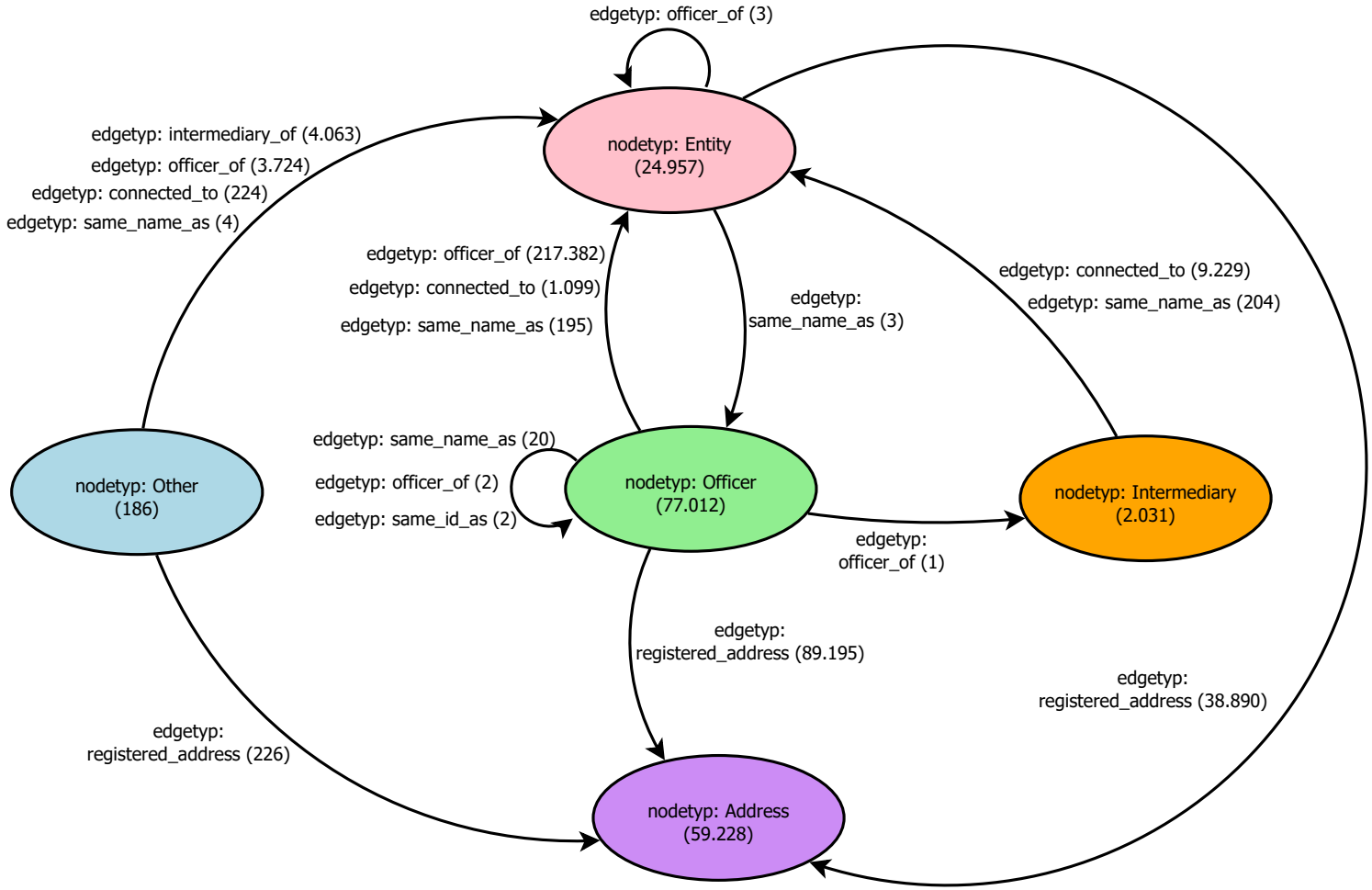
- Unterlagen der Anwaltskanzlei Appleby und des Treuhandunternehmens Asiaciti Trust
- durch Datenleck (angeblich zumindest teilweise durch SQL Injection, dazu gleich mehr)
- enthält u.a. Daten zu mehr als 120.000 Personen und Firmen
- aufbereitet durch ein weltweites Team von Journalisten: https://www.icij.org/
- die Journalisten werteten 13,4 Millionen Dokumente (ca. 1,4 TB) aus
- und stellten eine aufbereitete Version als Graphdatenbank zur Verfügung, um Recherchen zu vereinfachen
- https://de.wikipedia.org/wiki/Paradise_Papers

Das Graph„schema“



Quelle: https://offshoreleaks-data.icij.org/offshoreleaks/neo4j/guide/datashape.html

Das Graph„schema“ reverse-engineered



Achtung

Wir vermuten, dass in der Neo4j-Sandbox beim Laden des Schemas die Knotentypen Intermediary und Other vertauscht wurden.

Knotentypen

- Entity (legale Entität), z.B. Firma, Treuhandgesellschaft, Stiftung, etc.
- Officer (Funktionär): eine Person oder Firma, die in Beziehung steht zu Entity, z.B. ein Begünstigter, Direktor, Aktionär
- Intermediary (Vermittler): vermittelt z.B: Kontakt zu einer Offshore-Firma— oder einem Offshore-„Service“
- Address (Adresse)
- Other (Andere): andere Entitäten

Anmerkung zu dieser Modellierung

Es gibt Überlappung zwischen den Knotentypen: eine Firma kann als Entity, Officer, Intermediary und Other modelliert werden, u.a. mehrfach (je nach Rolle). Dies weist auf ein ernstes Modellierungsproblem hin.

Kantentypen

Anmerkung

Die Beschreibung der Kantentypen ist auf der Webseite unvollständig und noch verwirrender. Versucht man, aus den Daten, die verwendeten Kantentypen zu extrahieren, finden wir eine deutliche Diskrepanz zu dem auf der Webseite gezeigten Graph„schema“.

mehr dazu in der Übung

Schema first vs Schema later

Schema First

Erst wird ein (starkes/rigides) Schema festgelegt. **Erst danach** werden Daten geladen und in dieses Schema gezwungen.

Beispiel: relationales Datenbanksystem

vs

Schema later

Es wird kein oder nur ein schwaches Schema festgelegt. Zu jedem Zeitpunkt können Daten in das System geladen werden, sogar **bevor** ein möglicherweise schwaches Schema festgelegt wurde.

Beispiel: Neo4j, „Data Lake“, Key/Document-Stores (z.B. MongoDB)

Schema first vs Schema later

	Schema first	**Schema later**
Vorteile	Anfragen lassen sich leichter formulieren	Einfügen neuer Daten sehr einfach
Nachteile	Aufwand für Modellierung und Erstellen des Schemas, u.a. problematisch bei Erweiterungen des Modells; Einfügen neuer Daten aufwändig (Schema erzwingen)	Anfragen lassen sich schwieriger formulieren: u.U. müssen die ganzen Einzelfälle separat aufgesammelt werden mit der Anfrage

Neo4j Sandboxes, https://neo4j.com/sandbox-v2/

neo4j PRODUCTS SOLUTIONS PARTNERS CUSTOMERS LEARN DEVELOPERS Search

Launch a New Sandbox

Each sandbox includes data, interactive guides with example queries, and sample code.

Bloom Visual Discovery

Neo4j Bloom is a graph exploration application for visually interacting with graph data.

Launch Sandbox

Recommendations

Generate personalized real-time recommendations using a dataset of movie reviews.

Launch Sandbox

Crime Investigation

Explore connections in crime data using the POLE - Person, Object, Location, Event - model in a public dataset from Manchester, U.K.

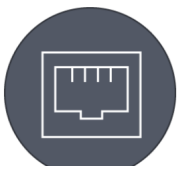
Launch Sandbox

Women's World Cup 2019

Explore the data behind the Women's World Cup with our World Cup Graph.

Launch Sandbox

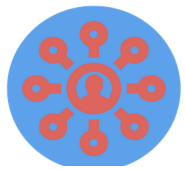
Network and IT Management



Dependency and root cause analysis + more for network and IT management

Launch Sandbox

Russian Twitter Trolls



Explore data released by NBC News from their investigation into Russian Twitter Trolls around the 2016 US election.

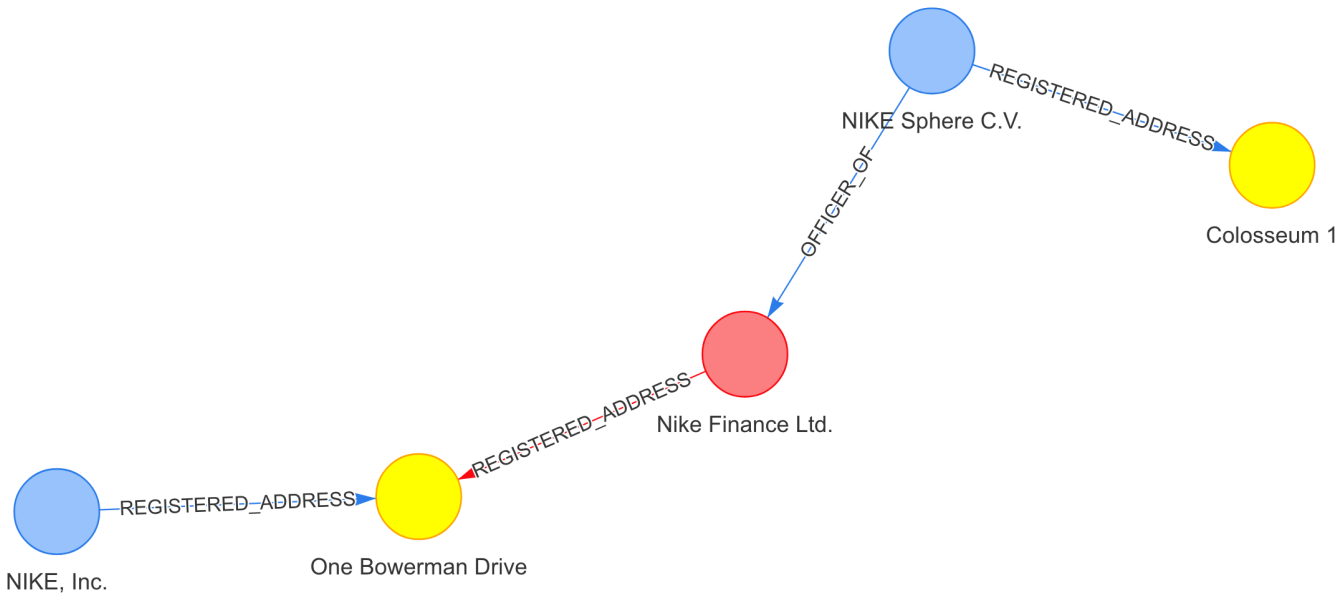
Launch Sandbox

Paradise Papers in Jupyter (Paradise Papers.ipynb)

In [7]:
```
query = """
MATCH p = SHORTESTPATH((a:Address) - [*] - (o:Officer))
WHERE a.address = 'Colosseum 1'
    AND o.name = 'NIKE, Inc.'
RETURN p
"""

data = graph.run(query).to_subgraph()
drawSubgraph(data, options, height="400", filename="nike_shortest_path_colosseum1_nike.html", physics=physics)
```

Out[7]:



Paradise Papers.ipynb

Datenjournalismus

2. Was sind die Datenmanagement und -analyseprobleme dahinter?

3. Grundlagen, um diese Probleme lösen zu können
   (a) Folien
   (b) Jupyter/Python/SQL Hands-on

jetzt:

Frage 3

Wie interagieren Nutzerinterface und DBMS?

danach:

Frage 4

Wie visualisieren wir die Daten?

Die wichtigsten Sicherheitsprobleme im Umgang mit Datenbanksystemen

1. SQL Injection
2. Passwörter falsch abgelegt
3. Defaultpasswörter nicht geändert
4. fehlende Zugriffskontrolle (access control lists, ACL)
5. Systeme nicht genügend isoliert (z.b. durch Virtualisierung oder physisch getrennte Systeme)
6. Offene Ports

Achtung

Für alle folgenden Beispiele und Techniken gilt: **Don't try this at home!**

Insbesondere diese Techniken an fremden Systemen auszuprobieren (selbst nur zum Lesen), kann bereits strafbar sein.

SQL Injection

Idee

Wenn eine Applikation einen String abfragt/generiert und diesen Wert dann in einen SQL-String einbaut, verändere den String so, dass in die SQL-Anfrage nicht nur dieser Parameter eingesetzt wird, sondern die SQL-Anfrage prinzipiell geändert wird.

```
statement = """
SELECT * FROM users
WHERE name = '" + userName + "';
"""
```

mit dem folgenden Wert von userName:

```
userName = "' OR '1'='1"
```

wird das folgende Statement erzeugt:

```
statement = """
SELECT * FROM users
WHERE name = '' OR '1'='1';
"""
```

D.h. jetzt werden alle Tupel ausgegeben und nicht nur ein bestimmter Nutzer!

Beachten Sie das kreative Nutzen von '!

SQL Injection mit Verändern der Datenbank

```
statement = """
SELECT * FROM users
WHERE name = '" + userName + "';
"""
```

mit dem folgenden Wert von userName:

```
userName = "bla'; DROP TABLE users;
SELECT * FROM foo WHERE '1' = '1"
```

wird das folgende Statement erzeugt:

```
statement = """
SELECT * FROM users
WHERE name = 'bla'; DROP TABLE users;
SELECT * FROM foo WHERE '1' = '1';
"""
```

D.h. wir löschen die gesamte Tabelle users!

Ablegen von Passwörtern

Achtung

Passwörter sollten grundsätzlich nicht im Klartext abgelegt werden. Es sollte nur ein Hash-Wert $h(pw)$ des Passworts pw gespeichert werden und dieser Hashwert bei Bedarf mit der Nutzereingabe verglichen werden.

Problem 1: Ist ein Paar $h(pw_i), pw_i$ bekannt, kennen wir jedes Passwort pw_j für das gilt $h(pw_i) = h(pw_j)$ (unter der Annahme, dass dies keine Hash-Kollision ist).

Problem 2: Ist die Domäne der Passwörter 'klein', können wir Passwörter in einer Brute-Force-Attacke aufzählen.

Beispiel: Passwort der Länge 8 Zeichen, Klein- und Großbuchstaben, Ziffern, 10 Sonderzeichen ergibt $72^8 = 7,2 \cdot 10^{14}$ mögliche Passwörter. Das ist zwar ein relativ großer Suchraum. Mit einer Wörterbuchattacke kann man darin aber mit Glück und Aufwand schwache Passwörter finden.

Salt

Salt

Für jeden Nutzer i wird einmalig eine zufällige Zahl S_i (der Salt) gewürfelt. Dieser Salt wird konkateniert mit dem Klartext-Passwort des Nutzers und das Ergebnis gehasht:

$$h(pw_i \oplus S_i)$$

Hier bezeichnet $\oplus$ den Konkatenierungsoperator. In der Datenbank wird für jeden Nutzer i das Tupel $(S_i, h(pw_i \oplus S_i))$ abgelegt.

Effekt 1: wenn zwei Nutzer dasselbe Passwort haben, sind die Hashwerte der Passwörter mit hoher Wahrscheinlichkeit unterschiedlich.

Effekt 2: eine Brute-Force-Attacke wird deutlich schwerer.

Problem 3: Was, wenn die Salts bekannt werden?

Dann wird eine Brute-Force-Attacke wieder möglich!

Pepper

Pepper

Es wird für die Anwendung **eine** zufällige Zahl P (der Pepper) gewürfelt. Dieser Pepper wird **nicht** in der Datenbank abgelegt sondern an einem „sicheren Ort“. Für jeden Nutzer i wird der Pepper konkateniert mit seinem Klartext-Passwort und Salt:

$$h(pw_i \oplus S_i \oplus P)$$

In der Datenbank wird für jeden Nutzer i nur das Tupel $(S_i, h(pw_i \oplus S_i \oplus P))$ abgelegt, P aber nicht!

Effekt 3: eine Brute-Force-Attacke wird deutlich schwerer, solange der „sichere Ort“ nicht auch kompromittiert wurde.

SQL Injection in Jupyter (SQL Injection.ipynb)

In the end, we print the `evil_string` that should be provided as username on login.

In [17]:
```
# Choose attacker's username and password
evil_username = 'student'
evil_password = 'evil_pwd'

# Compute attacker's pwdhash and salt
evil_salt = '0'*64
evil_pwdhash = binascii.hexlify(scrypt.hash(evil_password, evil_salt)).decode()

# Build sql injection string
evil_string = f"\'; "\
              f"INSERT INTO users(username, pwdhash, salt) VALUES"\
              f"(\'{evil_username}\', \'{evil_pwdhash}\', \'{evil_salt}\');"\
              f" --"

print(f"Insert this as username on login:\n{evil_string}")
```

```
Insert this as username on login:
'; INSERT INTO users(username, pwdhash, salt) VALUES('student2', 'ce65df4add0866ebef7969a
3522734dad70730244e78d0c01b4d5018b6de417d42c8a9b1e3fdf994f5e983fd7fd35bd3c6fa4b771e51fe0b
697023b8a6a93047', '0000000000000000000000000000000000000000000000000000000000000000'); --
```

SQL Injection.ipynb

Empfehlungen, um SQL Injection zu verhindern

1. Prepared Statements nutzen (direkt in SQL oder indirekt durch die DB-Bibliothek)
2. Eingabeparameter überprüfen (sanitizen): in der Applikation und/oder in der Datenbank
3. Zugriffsrechte (ACLs) einschränken wo möglich
4. nur den Teil der Datenbank zugreifbar machen, der für die Anwendung notwendig ist, entweder:
   - 4.1 die notwendigen Teile der Datenbank durch Sichten wrappen und freigeben
   - 4.2 noch besser: eine extra physische Kopie der Datenbank benutzen, die nur die notwendigen Daten enthält (schwierig bei Update-Szenarios)
5. Datenbank auf separater VM/Docker Image installieren
6. (automatisch) Testen, Testen, Testen, ...

Weitere Empfehlungen, um Sicherheitsprobleme zu vermeiden

1. falls möglich Snapshot der benötigten Datenbank direkt an den Client ausliefern und in der Webseite einbetten, statt den Client mit einer externen Datenbank kommunizieren zu lassen (gute Anwendung hierfür: Europawahlergebnisse der Zeit).
2. Die Datenbankverbindung niemals für den Client sichtbar machen, sondern Server-seitig bereitstellen (mittels PHP oder AJAX), Achtung: das alleine verhindert natürlich kein SQL-Injection!
3. alle Anfragen an die Datenbank tracken und (automatisch) in Echtzeit auf Auffälligkeiten überprüfen, im Zweifel eine Anfrage nicht ausführen
4. alle Passwörter aller Default-Accounts setzen
5. Ports überprüfen: müssen die offen sein, wenn ja für was eigentlich?

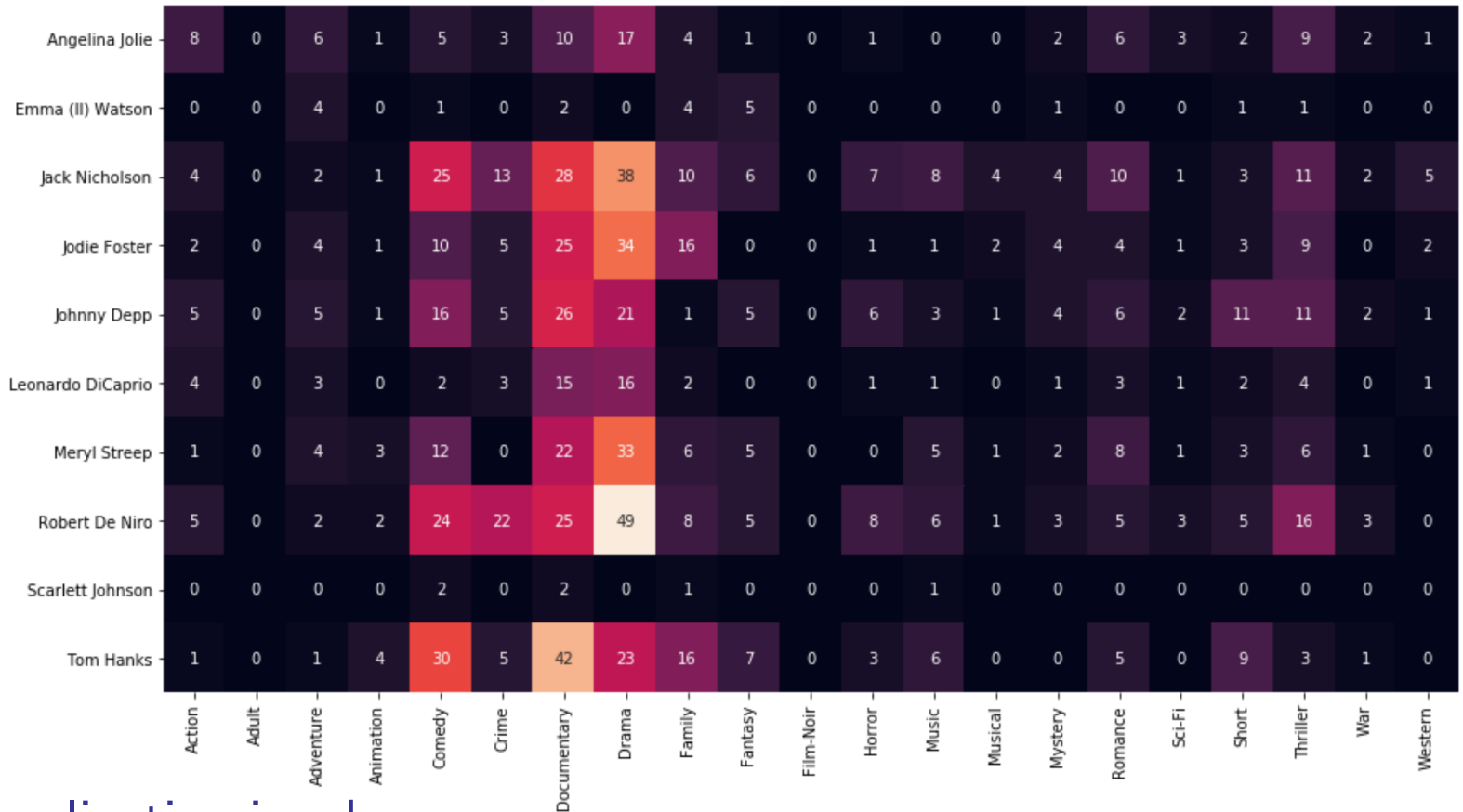
   Und hatten wir schon?
6. (automatisch) Testen, Testen, Testen, ...

Datenvisualisierung in Jupyter (Data Visualization.ipynb)

```
# Plot heatmap
sns.heatmap(pivottable, ax=ax[0], annot=True,
            xticklabels=genres, yticklabels=actors, cbar=False)#, cmap="YlGnBu", cbar=False)

# Plot heatmap with other color scheme
sns.heatmap(pivottable, ax=ax[1], annot=True,
            xticklabels=genres, yticklabels=actors, cbar=False, cmap="YlGnBu")
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x124b5bd68>



Data Visualization.ipynb