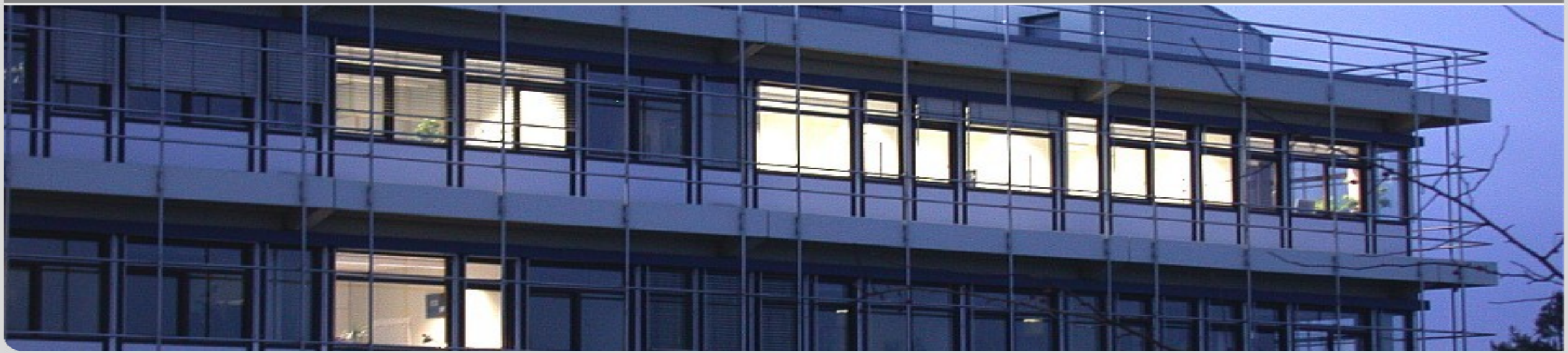
Karlsruher Institut für Technologie

# GraphDB Praktikum WS 22/23

## Arbeiten mit Servern | Einführung zu Cypher | 1. Gruppenaufgabe

**Daniel Betsche (daniel.betsche@kit.edu)**

IPD – Institute for Program Structures and Data Organization



# Arbeiten mit Servern

- Datenbanken sollten immer vor unberechtigten Zugriffen geschützt werden, dazu gehört neben Passwörtern und Nutzerrollen auch das Einschränken der Zugriffsmöglichkeiten.
- Bei uns geschieht dies durch Firewalls, welche nur Verbindungen aus dem LAN oder über SSH auf Port 22 aus dem Internet zulassen.
- Um auf die Datenbank zugreifen zu können müsst Ihr erst eine SSH Verbindung herstellen und könnt anschließend die Webseite aufrufen oder die Python API nutzen

# Grundlagen zu SSH

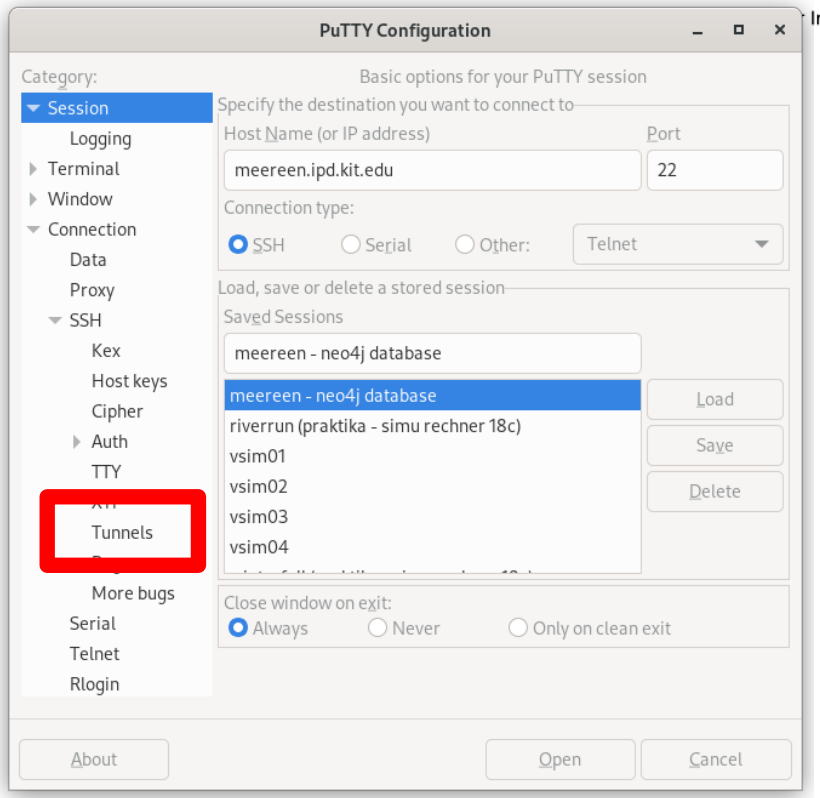
- Verbindung herstellen
  - `ssh <nutzer>@<servername>`
  - `Bsp.: ssh dbprakXX@meereen.ipd.kit.edu`
- Portweiterleitung **(wichtig!)**
  - `ssh -L <lokaler_port>:<remote_ziel>:<remote_port> <nutzer>@<servername>`
  - `Bsp.: ssh -L 7474:localhost:7474 dbprakXX@meereen.ipd.kit.edu`
  - Anschließend im Browser `localhost:<lokaler_port>` aufrufen
- SSH Key kopieren für Passwortloses verbinden (auf eurem Rechner ausführen, Linux/Mac only, auf Windows manuell kopieren mit scp)
  - `ssh-copy-id dbprakXX@meereen.ipd.kit.edu`

# Verbinden per SSH

- **SSH** Client Auswählen
  - Windows: MobaXTerm oder PuTTY
    https://mobaxterm.mobatek.net/.
  - Mac und Linux: eingebautes Terminal oder PuTTY
    https://www.putty.org/
- Eine Verbindung herstellen
  - Im Terminal:

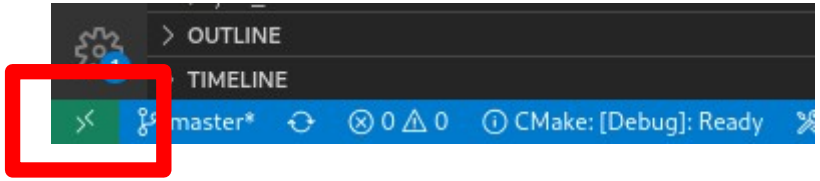
```
ssh -L 7474:localhost:7474 \
    -L 7689:localhost:7689 \
    dbprakXX@meereen.ipd.kit.edu
```

  - In PuTTY:
    Unter Connection > SSH > Tunnels zwei neue forwarded Ports einfügen.
    Z.B.: Source Port = `7474`, Destination = `localhost:7474`
  - In MobaXTerm: Analog zu Putty unter Tunneling > New SSH Tunnel

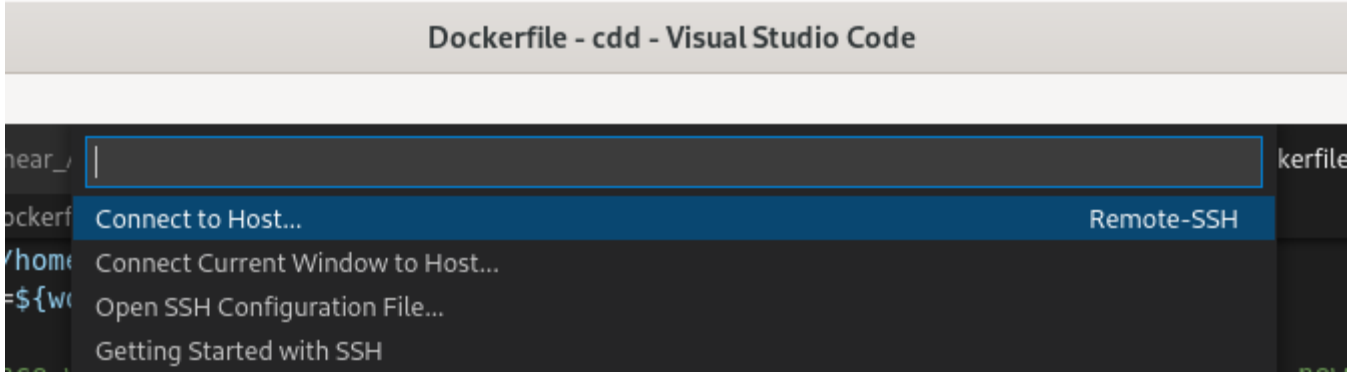


# Entwickeln per SSH

- Mit **VS Code** und der Erweiterung **Remote SSH**
  - Auf das Feld Links unten klicken




  - Hostnamen eingeben und Verbinden
    - Z.B.: dbkursXX@meereen.ipd.kit.edu



# Prozesse auf Servern ausführen

- Bei normaler Nutzung muss die (SSH-)Verbindung zu einem Server aufrecht gehalten werden während der Ausführung eines Programms.
  - Unerwartete Unterbrechungen (z.B. durch loses LAN Kabel, WLAN Ausfall) brechen auch Programm-Ausführung ab
  - Lang laufende Prozesse erfordern auch, das der Client PC (Laptop) aktiv bleibt
- Die Lösung dafür sind sogennante „terminal multiplexer“
  - Programme die mehrere Terminals unabhängig voneinader erzeugen können und es erlauben mehrere Programme parallel auszuführen
  - Können auch ohne aktive Verbindung zu einem Server weiterlaufen
  - Am bekanntesten sind Screen und Tmux

# Prozesse auf Servern ausführen – mit Screen

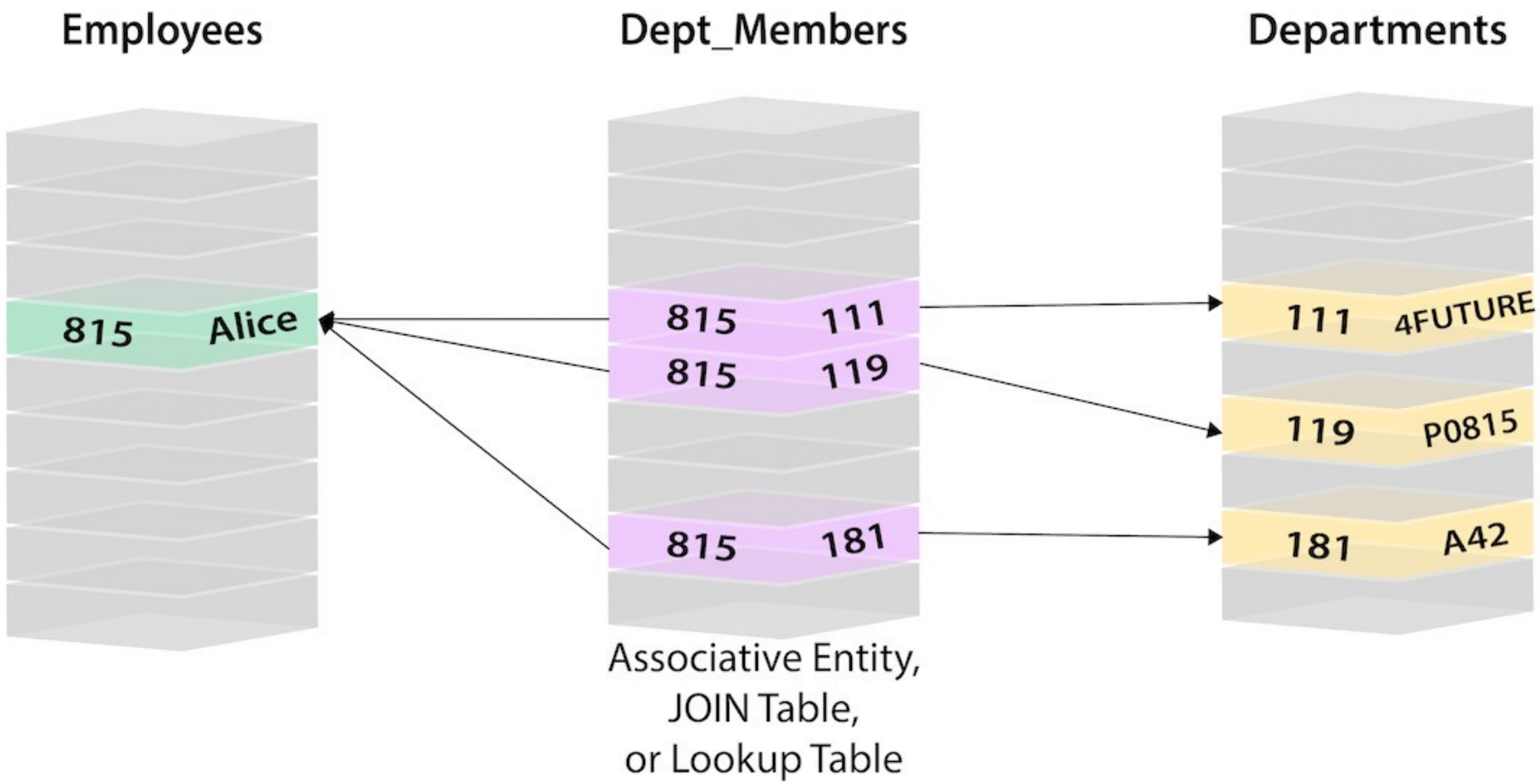
- Screen ist bei (fast) allen Linux Distributionen per default installiert und dadurch die einfachste Wahl
- Die wichtigsten Befehle lauten:
  - Neues Terminal erzeugen: `screen`
  - Von aktivem Terminal detachen: Tastenkombination Strg + a, d
  - Aktive Terminals auflisten: `screen -ls`
  - Zu aktivem Terminal erneut attachen: `screen -r [session]`
    - Hier kann die Autovervollständigung mit Tab genutzt werden
  - Aktives Terminal beenden (muss attached sein) Tastenkombination Strg + d
  - Hilfe: `screen -h`

# Prozesse auf Servern ausführen – mit Screen

- Typischer Arbeitsablauf
  - screen
  - Eigenes Programm ausführen, z.B. `python myprogram.py`
  - Von aktivem Terminal detachen → Tastenkombination Strg + a, d
  - < Zeit verstreicht >
  - Zu aktivem Terminal erneut attachen: `screen -r [session]`
  - Ergebnisse anschauen und speichern oder neues Programm starten
  - Wenn fertig: Aktives Terminal beenden mit Tastenkombination Strg + d

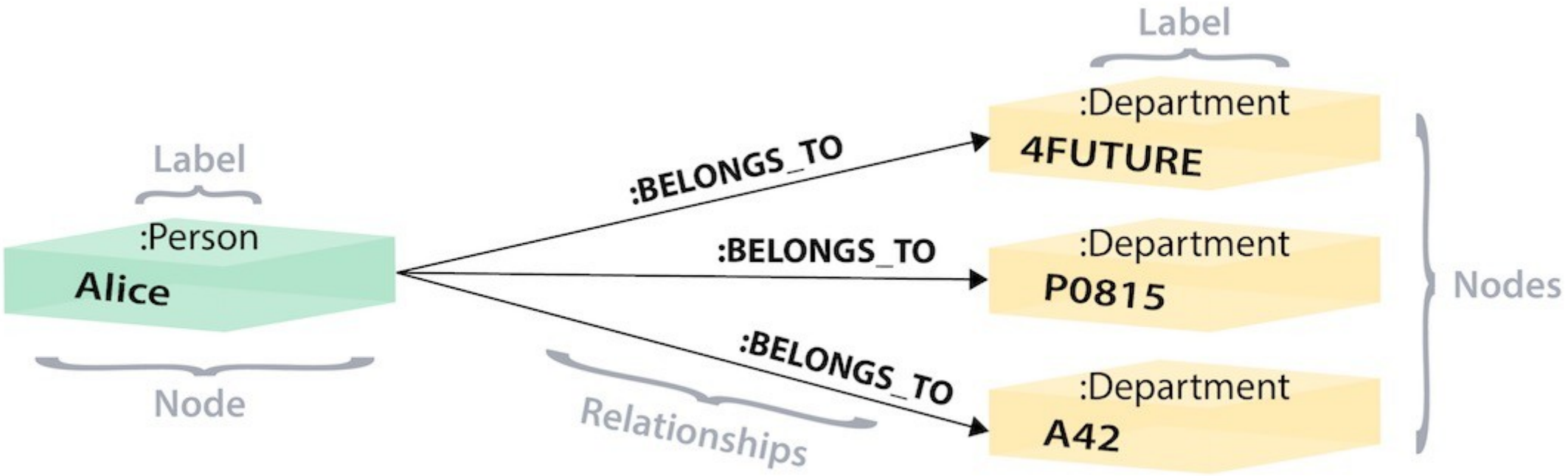
# Cypher Einführung

# Von relationalem Modell zum Graphmodell (1/3)




Quelle: https://neo4j.com/developer/relational-to-graph-modeling/

# Von relationalem Modell zum Graphmodell (2/3)




Quelle: https://neo4j.com/developer/relational-to-graph-modeling/

# Von relationalem Modell zum Graphmodell (3/3)
## Vergleich der Komponenten

- Relational:
  - Tabelle
  - Reihe
  - Spalte
  - Primärschlüssel
  - Constraints/Indexes
  - Fremdschlüssel
  - Standardwerte
  - Join Tables
- Graph:
  - Knoten-Label
  - Knoten
  - Knoten-Attribut
  - Als Attribut wenn natürlich
  - Constraints/Indexes
  - Kanten
  - Keine Standardwerte
  - Kanten
- Weitere Informationen unter https://neo4j.com/developer/relational-to-graph-modeling/

# Begrifflichkeiten Graphdatenbank

- **Neo4j** ist die Datenbank selbst. Es gibt unterschiedliche Versionen sowohl für Arbeitsplätze als auch Serverinstanzen. Wir arbeiten mit der Community-Server Variante.
- **Cypher** ist die Querysprache von Neo4j, das Äquivalent von SQL bei relationalen Datenbanken
  - Dokumentation: https://neo4j.com/docs/cypher-manual/current/
- **Patterns** sind Ausdrücke um (Teil-)Graphen darzustellen. Sie sind Kern von Graph-queries mit Cypher.

# Cypher – Patterns

- Patterns dienen zur Beschreibung von (Teil-)Graphen und sind essentieller Bestandteil jeder Anfrage
- Die ganze Übersicht ist hier zu finden: https://neo4j.com/docs/cypher-manual/current/syntax/patterns/
- Die wichtigsten Elemente
  - Knoten: ()
  - Kanten: --, -->, <--, -[*n]-, -[*2]-, -[*3..5]-, -[*]-
  - Labels:  (:User), [:ARBEITET_IN]
  - Attribute:  ({id:5, name:‘Bob’, sport:‘Skiing’}); [:{active:true, distance:50}]
  - Variablen:  (a)--(b), -[r]-
  - Verknüpfte Patterns mit Komma Operator: (a)--(b)--(c), (b)--(d)

# Cypher – Die wichtigsten Statements

- Die ganze Übersicht: https://neo4j.com/docs/cypher-manual/current/
- EXPLAIN/PROFILE
- MATCH (a)--(b)
- WHERE a.id = 5
- WITH b, collect(a.id) as ids
- MATCH (c)-->(d), (b)--(c)
- RETURN ids, b, c

# Cypher – Livebeispiel

# Python API

- Neben mehreren Alternativen gibt es einen offiziellen Python Driver mit welchem wir arbeiten werden.
  - https://neo4j.com/developer/python/
- Die Nutzung des Neo4j Treibers erfolgt Analog zu bereits kennengelernten SQL Treibern.
  - Durch den Aufruf von session.run() wird die Anfrage an die Datenbank geschickt und dort ausgeführt.
  - Erst durch den Aufruf von result.data() wird das Ergebnis von der DB angefordert.
  - ACHTUNG: Ergebnis kann nur innerhalb einer aktiven session angefordert werden und ist außerhalb leer. Daher an dieser Stelle in anderer Variable speichern. Pandas Dataframes bieten sich dazu an.

```python
import pandas as pd
from neo4j import GraphDatabase

uri, user, password = "bolt://localhost:7689", "neo4j", "password"
driver = GraphDatabase.driver(uri, auth=(user, password))


def run_query(query, params):
    with driver.session() as session:
        result = session.run(query, params)
        df = pd.DataFrame(result.data())
    return df


query_count_loops_in_timeslice = """
    MATCH (n:Loop {time:$time})
    RETURN count(n)
    """

params = {"time": 50}
result = run_query(query_count_loops_in_timeslice, params)
driver.close()
```