L19: Graph-Datenbanken & Semantic Web

Session 19 – Lecture

Dauer: 90 Minuten

Lernziele: LZ 1 (Paradigmen & Einsatzszenarien verstehen), LZ 5 (Stärken/Schwächen bewerten)

Block: 4 – Theorie, Optimierung & Polyglot

Willkommen zur neunzehnten Session! Sie haben bereits Property Graphs mit Cypher kennengelernt. Heute tauchen wir in eine andere Graph-Welt ein: das Semantic Web mit RDF und SPARQL. Klingt akademisch? Ist es – aber auch extrem praktisch!

Stellen Sie sich vor: Die gesamte Wikipedia als strukturierte Datenbank, die Sie mit SQL-ähnlichen Queries abfragen können. Das ist DBpedia – und genau damit arbeiten wir heute!

Motivation: Das Semantic Web

Heute tauchen wir in die Welt der Graph-Datenbanken ein – aber nicht mit einem proprietären System, sondern mit dem offenen Standard des Semantic Web: RDF und SPARQL!

Was ist RDF?

RDF (Resource Description Framework):

```
<http://example.org/alice> a <http://example.org/Person> .
<http://example.org/alice> <http://example.org/name> "Alice" .
<http://example.org/alice> <http://example.org/age> 30 .
<http://example.org/alice> <http://example.org/knows> <http://example.org/b
```

RDF ist ein W3C-Standard für strukturierte, verlinkte Daten. Es basiert auf Triples (Subject-Predicate-Object) und ist die Grundlage des Semantic Web!

Die Vision: Linked Open Data

Warum ist RDF wichtig? Weil es das Web maschinenlesbar macht!

Das Problem heute:

```
Wikipedia    →  Nur für Menschen lesbar (HTML)
Google       →  Eigenes Schema (Knowledge Graph)
Wikidata     →  Eigenes Format
```

Die Vision: Semantic Web

```
Alle Daten     →  RDF (standardisiert!)
```

```
Alle Daten     →   RDF (standardisiert!)
Alle Queries   →   SPARQL (standardisiert!)
Alle Links     →   URIs (eindeutig!)
```

Stellen Sie sich vor: Jede Website stellt ihre Daten als RDF bereit. Ihr SPARQL-Query kann Wikipedia, Wikidata, DBpedia und Ihre eigene Datenbank gleichzeitig abfragen. Das ist Linked Open Data!

Reale Use Cases:

- **DBpedia** – Wikipedia als Knowledge Graph (15 Millionen Entities!)
- **Wikidata** – Kollaborative Wissensdatenbank (100 Millionen Items!)
- **Schema.org** – Strukturierte Daten für Suchmaschinen
- **Bio2RDF** – Biomedizinische Datenbanken vernetzt
- **GeoNames** – Geografische Daten weltweit

Teil 1: RDF-Grundlagen

RDF basiert auf einem einfachen Konzept: Alles ist ein Triple! Subject – Predicate – Object. Das war's.

Triples: Subject – Predicate – Object

Ein Triple ist wie eine Mini-Aussage in der Form „A tut B“ oder „A ist B“.

Beispiel: The Beatles

```
Subject                  Predicate        Object
_________________________  _______________  _____________________

:TheBeatles              rdf:type         :Band
:TheBeatles              :foundedIn       1960
:TheBeatles              :genre           :Rock
:TheBeatles              :hasMember       :JohnLennon
:JohnLennon              rdf:type         :Person
:JohnLennon              :name            "John Lennon"
```

Sehen Sie das Pattern? Jedes Triple beschreibt eine Beziehung. Zusammen bilden sie einen Graph!

RDF = Flexible Tabelle:

Subject	Predicate	Object
:TheBeatles	rdf:type	:Band
:TheBeatles	:foundedIn	1960
:TheBeatles	:genre	:Rock
:TheBeatles	:hasMember	:JohnLennon
:JohnLennon	rdf:type	:Person
:JohnLennon	:name	„John Lennon“

RDF ist eigentlich nur eine große Tabelle mit drei Spalten. Aber durch die Verknüpfungen entsteht ein Graph!

URIs: Globale Identifikatoren

Der Schlüssel zum Semantic Web: Statt `id=42` verwenden wir URIs – eindeutig im gesamten Web!

Beispiel:

```
Schlecht:  id=12345, name="London"
Gut:       <http://dbpedia.org/resource/London>
```

Warum URIs?

- **Eindeutig:** London (UK) vs. London (Ontario) – verschiedene URIs
- **Verlinkbar:** URIs können auf andere Datensätze zeigen
- **Dereferencable:** URI aufrufen → Daten erhalten!

```
<http://dbpedia.org/resource/London>
  a dbo:City ;
  dbo:country <http://dbpedia.org/resource/United_Kingdom> ;
  dbo:population 8982000 ;
  rdfs:label "London"@en ;
  owl:sameAs <http://www.wikidata.org/entity/Q84> .
```

Die letzte Zeile ist Gold wert: `owl:sameAs` verlinkt DBpedia-London mit Wikidata-London. So entsteht das Linked Data Web!

Turtle-Syntax: RDF für Menschen

Turtle ist eine lesbare Syntax für RDF. Viel besser als XML!

Volle URI-Form (unleserlich!):

```
<http://example.org/TheBeatles> <http://www.w3.org/1999/02/22-rdf-syntax-ns
  <http://example.org/Band> .
<http://example.org/TheBeatles> <http://example.org/foundedIn> 1960 .
```

Mit Prefixes (lesbar!):

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:TheBeatles rdf:type ex:Band .
ex:TheBeatles ex:foundedIn 1960 .
```

Noch kompakter (Shortcuts!):

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:TheBeatles
  a ex:Band ;            # 'a' = rdf:type
  ex:foundedIn 1960 ;
  ex:genre ex:Rock ;
  ex:hasMember ex:JohnLennon , ex:PaulMcCartney , ex:GeorgeHarrison .
```

Das Semikolon bedeutet „gleiches Subject, neues Predicate“. Das Komma bedeutet „gleiches Subject + Predicate, neues Object“. Elegant!

Interaktiv: Euer erstes RDF

Jetzt seid ihr dran! Hier ist ein kleines Musik-Knowledge-Graph. Schaut euch die Turtle-Syntax an – gleich fragen wir es mit SPARQL ab!

Turtle: Musik-Bands

```
@prefix : <http://example.org/music#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:TheBeatles
  a :Band ;
  :name "The Beatles" ;
  :foundedIn 1960 ;
  :genre :Rock ;
  :hasMember :JohnLennon , :PaulMcCartney , :GeorgeHarrison , :RingoS
  :album :SgtPeppers , :AbbeyRoad .

:PinkFloyd
  a :Band ;
  :name "Pink Floyd" ;
  :foundedIn 1965 ;
  :genre :ProgressiveRock ;
  :hasMember :DavidGilmour , :RogerWaters , :NickMason , :RichardWrig
  :album :DarkSideOfTheMoon , :TheWall .

:Radiohead
  a :Band ;
  :name "Radiohead" ;
  :foundedIn 1985 ;
  :genre :AlternativeRock ;
  :hasMember :ThomYorke , :JonnyGreenwood ;
  :album :OKComputer .

:JohnLennon
  a :Person ;
  :name "John Lennon" ;
  :birthYear 1940 .

:PaulMcCartney
  a :Person ;
  :name "Paul McCartney" ;
  :birthYear 1942 .

:GeorgeHarrison
  a :Person ;
  :name "George Harrison" ;
  :birthYear 1943 .

:RingoStarr
  a :Person ;
  :name "Ringo Starr" ;
  :birthYear 1940 .

```

```
:ThomYorke
  a :Person ;
  :name "Thom Yorke" ;
  :birthYear 1968 .

:DavidGilmour
  a :Person ;
  :name "David Gilmour" ;
  :birthYear 1946 .

:SgtPeppers
  a :Album ;
  :title "Sgt. Pepper's Lonely Hearts Club Band" ;
  :releaseYear 1967 .

:AbbeyRoad
  a :Album ;
  :title "Abbey Road" ;
  :releaseYear 1969 .

:DarkSideOfTheMoon
  a :Album ;
  :title "The Dark Side of the Moon" ;
  :releaseYear 1973 .

:OKComputer
  a :Album ;
  :title "OK Computer" ;
  :releaseYear 1997 .
```

SPARQL: Alle Bands

```
PREFIX : <http://example.org/music#>

SELECT ?band ?name ?year
WHERE {
  ?band a :Band .
  ?band :name ?name .
  ?band :foundedIn ?year .
}
ORDER BY ?year
```

i	band	year	name
0	http://example.org/music#TheBeatles	"1960"^^http://www.w3.org/2001/XMLSchema#integer	"The Beatles"
1	http://example.org/music#PinkFloyd	"1965"^^http://www.w3.org/2001/XMLSchema#integer	"Pink Floyd"
2	http://example.org/music#Radiohead	"1985"^^http://www.w3.org/2001/XMLSchema#integer	"Radiohead"

Perfekt! Ihr seht das Pattern? Das RDF definiert die Daten (Triples), SPARQL fragt sie ab (wie SQL SELECT). Gleich lernt ihr die SPARQL-Syntax!

RDF versteckt in HTML: Microdata & Schema.org

Das Geniale am Semantic Web: RDF kann direkt in normalem HTML eingebettet werden! So werden Websites maschinenlesbar.

Beispiel: Person als HTML ohne Semantik

```
<div>
  <h1>John Lennon</h1>
  <p>Geboren: 9. Oktober 1940</p>
  <p>Band: The Beatles</p>
  <p>Beruf: Musiker, Songwriter</p>
</div>
```

Für Menschen lesbar, für Maschinen nutzlos! Suchmaschinen wissen nicht, dass „John Lennon“ eine Person ist.

Jetzt mit Microdata & Schema.org:

```
<div itemscope itemtype="https://schema.org/Person">
  <h1 itemprop="name">John Lennon</h1>
  <p>Geboren: <time itemprop="birthDate" datetime="1940-10-09">9. Oktober 1
    </time></p>
  <p>Band:
    <span itemscope itemtype="https://schema.org/MusicGroup">
      <span itemprop="name">The Beatles</span>
    </span>
  </p>
  <p>Beruf:
    <span itemprop="jobTitle">Musiker</span>,
    <span itemprop="jobTitle">Songwriter</span>
```

```
  </p>
</div>
```

Jetzt versteht Google, dass dies eine Person ist! Die `itemscope` und `itemprop`-Attribute fügen semantische Bedeutung hinzu.

Als RDF gedacht (unsichtbar für Nutzer):

```
<http://example.org/john-lennon> a schema:Person ;
  schema:name "John Lennon" ;
  schema:birthDate "1940-10-09"^^xsd:date ;
  schema:memberOf [
    a schema:MusicGroup ;
    schema:name "The Beatles"
  ] ;
  schema:jobTitle "Musiker" , "Songwriter" .
```

Das ist das Semantic Web in Aktion! HTML für Menschen, Microdata für Maschinen – beide in einem Dokument!

Noch ein Beispiel: Produkt mit Bewertung

```
<div itemscope itemtype="https://schema.org/Product">
  <h2 itemprop="name">Wireless Headphones XM4</h2>
  <img itemprop="image" src="headphones.jpg" alt="Product Image" />

  <div itemprop="offers" itemscope itemtype="https://schema.org/Offer">
    <span itemprop="price" content="299.99">299,99 €</span>
    <meta itemprop="priceCurrency" content="EUR" />
    <link itemprop="availability" href="https://schema.org/InStock" />
  </div>

  <div itemprop="aggregateRating" itemscope itemtype="https://schema.org
    /AggregateRating">
    Bewertung:
    <span itemprop="ratingValue">4.5</span> von
    <span itemprop="bestRating">5</span> Sternen
    (<span itemprop="ratingCount">127</span> Bewertungen)
  </div>
</div>
```

Solche Annotationen nutzt Google für Rich Snippets in Suchergebnissen – Sterne, Preise, Verfügbarkeit direkt sichtbar!

Warum ist das wichtig?

- **SEO:** Bessere Platzierung in Suchmaschinen
- **Rich Snippets:** Attraktivere Suchergebnisse
- **Knowledge Graph:** Google kann Fakten extrahieren
- **Linked Data:** Websites werden vernetzbar

Jede moderne Website sollte Microdata nutzen. Es ist unsichtbar für Nutzer, aber macht das Web maschinenlesbar!

Teil 2: SPARQL – Die Query Language

SPARQL ist für RDF, was SQL für relationale Datenbanken ist. Aber mit Graph-Semantik!

SELECT: Grundstruktur

Die Basis-Syntax ähnelt SQL – aber statt Tabellen matchen wir Triple-Patterns!

SQL-Reminder:

```
SELECT name, price
FROM products
WHERE category = 'Electronics';
```

SPARQL-Äquivalent:

```
SELECT ?name ?price
WHERE {
  ?product :name ?name .
  ?product :price ?price .
  ?product :category :Electronics .
}
```

Verstehen Sie den Unterschied? In SQL beschreiben Sie Tabellen und Spalten. In SPARQL beschreiben Sie Triple-Patterns: „Finde alles, wo Subject-Predicate-Object passt!“

Pattern Matching: Das Herzstück

SPARQL funktioniert durch Pattern Matching. Variables (`?variable`) werden an passende Werte gebunden.

Beispiel: Alle Mitglieder einer Band

Daten (RDF)

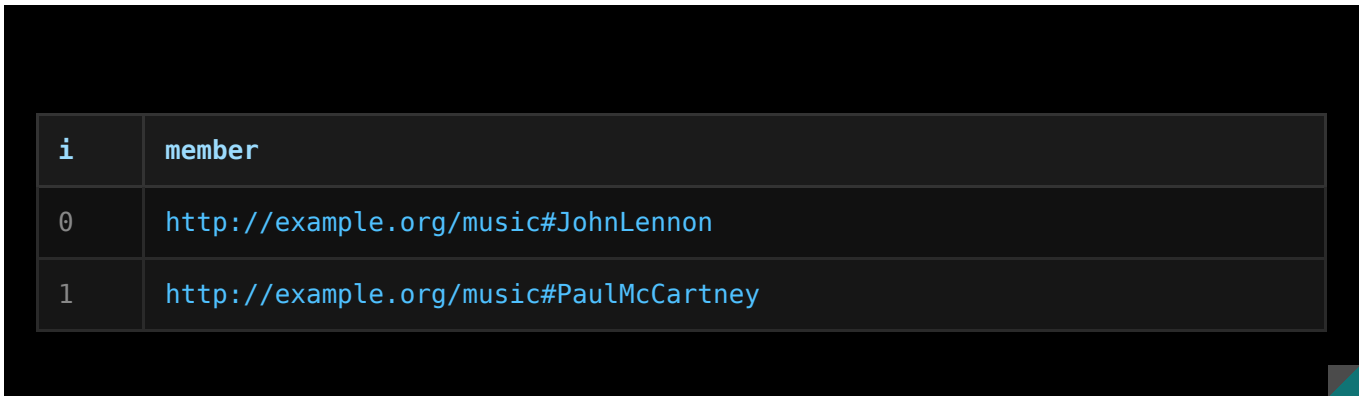
```
@prefix : <http://example.org/music#> .

:TheBeatles :hasMember :JohnLennon .
:TheBeatles :hasMember :PaulMcCartney .
:PinkFloyd :hasMember :DavidGilmour .
```

Query (SPARQL)

```
PREFIX : <http://example.org/music#>

SELECT ?member
WHERE {
  :TheBeatles :hasMember ?member .
}
```



i	member
0	http://example.org/music#JohnLennon
1	http://example.org/music#PaulMcCartney

Das Pattern `:TheBeatles :hasMember ?member` matcht alle Triples mit diesem Subject und Predicate. Die Object-Werte werden an `?member` gebunden!

Variables & Joins

Mehrere Patterns in einer Query? Das ist ein JOIN – nur implizit durch gemeinsame Variables!

Beispiel: Bands mit ihren Mitgliedern UND deren Geburtsjahr

Daten

```
@prefix : <http://example.org/music#> .

:TheBeatles a :Band ;
  :name "The Beatles" ;
  :hasMember :JohnLennon , :PaulMcCartney .

:JohnLennon
  :name "John Lennon" ;
  :birthYear 1940 .

:PaulMcCartney
  :name "Paul McCartney" ;
  :birthYear 1942 .
```

Query: Band-Mitglieder mit Geburtsjahr

```
PREFIX : <http://example.org/music#>

SELECT ?bandName ?memberName ?birthYear
WHERE {
  ?band a :Band .
  ?band :name ?bandName .
  ?band :hasMember ?member .
  ?member :name ?memberName .
  ?member :birthYear ?birthYear .
  FILTER(?bandName = "The Beatles")
}
ORDER BY ?birthYear
```

i	bandName	birthYear	memberName
0	"The Beatles"	"1940"^^http://www.w3.org/2001/XMLSchema#integer	"John Lennon"
1	"The Beatles"	"1942"^^http://www.w3.org/2001/XMLSchema#integer	"Paul McCartney"

Sehen Sie die Magie? `?member` ist die gemeinsame Variable – sie verbindet beide Pattern-Gruppen. Das ist ein impliziter JOIN!

SPARQL vs. SQL

Vergleichstabelle:

Feature	SQL	SPARQL
Datenmodell	Tabellen (relational)	Triples (Graph)
SELECT	`SELECT col1, col2`	`SELECT ?var1 ?var2`
FROM	`FROM table`	Implizit (Pattern matching)
WHERE	`WHERE col = value`	`WHERE { pattern }`
JOIN	Explizit (`INNER JOIN`)	Implizit (shared variables)
FILTER	`WHERE col > 100`	`FILTER(?var > 100)`
ORDER BY	`ORDER BY col`	`ORDER BY ?var`
LIMIT	`LIMIT 10`	`LIMIT 10`
DISTINCT	`SELECT DISTINCT`	`SELECT DISTINCT`
Aggregation	`COUNT()`, `SUM()`, `AVG()`	`COUNT()`, `SUM()`, `AVG()`
GROUP BY	`GROUP BY col`	`GROUP BY ?var`

SPARQL ist SQL sehr ähnlich – aber statt Tabellen-Joins nutzen Sie Triple-Pattern-Matching. Das macht es flexibler für Graph-Daten!

FILTER: Bedingungen wie SQL WHERE

Mit FILTER schränken Sie Ergebnisse ein – wie SQL WHERE nach dem JOIN!

Daten: Bands mit Gründungsjahr

```
@prefix : <http://example.org/music#> .

:TheBeatles a :Band ; :name "The Beatles" ; :foundedIn 1960 .
:PinkFloyd a :Band ; :name "Pink Floyd" ; :foundedIn 1965 .
:LedZeppelin a :Band ; :name "Led Zeppelin" ; :foundedIn 1968 .
:Radiohead a :Band ; :name "Radiohead" ; :foundedIn 1985 .
:Nirvana a :Band ; :name "Nirvana" ; :foundedIn 1987 .
```

Query: Bands vor 1970

```
PREFIX : <http://example.org/music#>

SELECT ?name ?year
WHERE {
  ?band a :Band .
  ?band :name ?name .
  ?band :foundedIn ?year .
  FILTER(?year < 1970)
}
ORDER BY ?year
```

i	name	year
0	"The Beatles"	"1960"^^http://www.w3.org/2001/XMLSchema#integer
1	"Pink Floyd"	"1965"^^http://www.w3.org/2001/XMLSchema#integer
2	"Led Zeppelin"	"1968"^^http://www.w3.org/2001/XMLSchema#integer

FILTER funktioniert wie SQL WHERE – aber nach dem Pattern Matching! Sie können `<`, `>`, `=`, `!=`, `&&`, `||` und Funktionen wie `REGEX()` nutzen.

OPTIONAL: Wie SQL LEFT JOIN

Manchmal fehlen Daten. OPTIONAL macht ein Pattern optional – wie LEFT JOIN!

Daten: Bands (manche ohne Genre)

```
@prefix : <http://example.org/music#> .

:TheBeatles a :Band ; :name "The Beatles" ; :genre :Rock .
:PinkFloyd a :Band ; :name "Pink Floyd" ; :genre :ProgressiveRock .
:Radiohead a :Band ; :name "Radiohead" .
```

Query: Alle Bands, Genre optional

```
PREFIX : <http://example.org/music#>

SELECT ?name ?genre
WHERE {
  ?band a :Band .
  ?band :name ?name .
  OPTIONAL { ?band :genre ?genre . }
}
```

i	genre	name
0	http://example.org/music#Rock	"The Beatles"
1	http://example.org/music#ProgressiveRock	"Pink Floyd"
2		"Radiohead"

Ohne OPTIONAL würde Radiohead nicht im Ergebnis erscheinen (weil kein Genre vorhanden). Mit OPTIONAL wird `?genre` einfach leer gelassen!

Aggregation & GROUP BY

Wie in SQL können Sie aggregieren und gruppieren!

Daten: Bands mit mehreren Alben

```
@prefix : <http://example.org/music#> .

:TheBeatles a :Band ; :name "The Beatles" ; :album :A1 , :A2 , :A3 , :
:PinkFloyd a :Band ; :name "Pink Floyd" ; :album :A5 , :A6 .
:Radiohead a :Band ; :name "Radiohead" ; :album :A7 , :A8 , :A9 .
```

Query: Anzahl Alben pro Band

```
PREFIX : <http://example.org/music#>

SELECT ?name (COUNT(?album) AS ?albumCount)
WHERE {
  ?band a :Band .
  ?band :name ?name .
  ?band :album ?album .
}
GROUP BY ?name
ORDER BY DESC(?albumCount)
```

i	name	albumCount
0	"The Beatles"	"4"^^http://www.w3.org/2001/XMLSchema#integer
1	"Radiohead"	"3"^^http://www.w3.org/2001/XMLSchema#integer
2	"Pink Floyd"	"2"^^http://www.w3.org/2001/XMLSchema#integer

Wie in SQL: GROUP BY gruppiert nach Variable, COUNT/SUM/AVG/MIN/MAX aggregieren. Die AS-Syntax ist identisch!

Teil 3: DBpedia Showcase

Jetzt wird es real! DBpedia ist Wikipedia als RDF – 15 Millionen Entities, 13 Milliarden Triples. Alles abfragbar mit SPARQL!

Showcase 1: Bands mit nur einem Mitglied

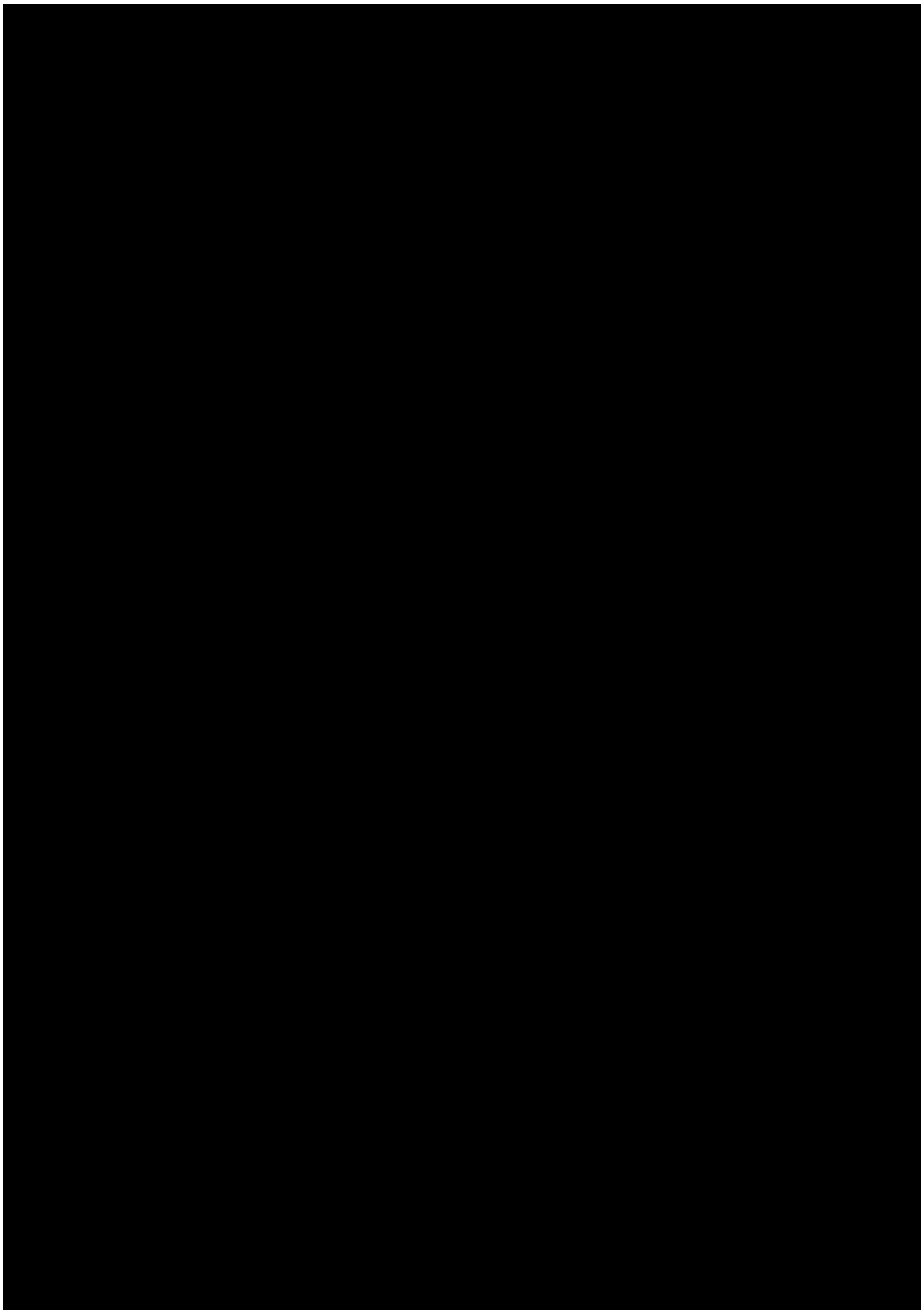
Euer Beispiel! Welche Bands haben nur ein einziges Mitglied? Das ist ungewöhnlich – schauen wir nach!

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?band (SAMPLE(?member) AS ?theMember)
WHERE {
  ?band a dbo:Band .
  ?band dbo:bandMember ?member .
}
GROUP BY ?band
```

```
12  HAVING (COUNT(?member) = 1)
13  LIMIT 20
```



i	band	theMember
0	http://dbpedia.org/resource/Poor_Moon_(band)	http://dbpedia.org/resource/Christian_Wargo
1	http://dbpedia.org/resource/JDS_(duo)	http://dbpedia.org/resource/Darren_Pearce
2	http://dbpedia.org/resource/Ascendant_Vierge	http://dbpedia.org/resource/Paul_Seul
3	http://dbpedia.org/resource/Colorblind_(band)	http://dbpedia.org/resource/Colorblind_(post-hardcore_band)
4	http://dbpedia.org/resource/Pete_Roth_Trio	http://dbpedia.org/resource/Bill_Bruford
5	http://dbpedia.org/resource/Puddle_of_Mudd__Puddle_of_Mudd__1	http://dbpedia.org/resource/Wes_Scantlin
6	http://dbpedia.org/resource/Blaze_Bayley	http://dbpedia.org/resource/List_of_Blaze_Bayley_band_members
7	http://dbpedia.org/resource/East_of_Eden_(Japanese_band)	http://dbpedia.org/resource/Ayasa_(violinist)
8	http://dbpedia.org/resource/Sicksense	http://dbpedia.org/resource/Vicky_Psarakis
9	http://dbpedia.org/resource/SKAI_(band)	http://dbpedia.org/resource/Oleh_Sobchuk
10	http://dbpedia.org/resource/Itkron_Pungkiatrussamee	http://dbpedia.org/resource/Taitosmith
11	http://dbpedia.org/resource/Casting_Crowns__Casting_Crowns__1	http://dbpedia.org/resource/Mark_Hall_(musician)
12	http://dbpedia.org/resource/Kalush_(rap_group)____1	http://dbpedia.org/resource/Oleh_Psiuk
13	http://dbpedia.org/resource/Because_of_You,_I_Shine	http://dbpedia.org/resource/Pongpol_Panyamit
14	http://dbpedia.org/resource/Venera_(band)	http://dbpedia.org/resource/James_Shaffer
15	http://dbpedia.org/resource/Warkings	http://dbpedia.org/resource/Georg_Neuhauser

16	http://dbpedia.org/resource/Video_(band)	http://dbpedia.org/resource/Wojciech_Łuszczykiewicz
17	http://dbpedia.org/resource/Grand_Ole_Party__Grand_Ole_Party__1	http://dbpedia.org/resource/Kristin_Kontrol
18	http://dbpedia.org/resource/Toke_D_Keda	http://dbpedia.org/resource/Eddie_Blazquez
19	http://dbpedia.org/resource/The_Enemy_(English_rock_band)__The_Enemy__1	http://dbpedia.org/resource/Tom_Clarke_(musician)

Interessant! Viele Solo-Projekte nutzen den Begriff „Band“. Die Query gruppiert nach Band, zählt Mitglieder und filtert mit HAVING – wie in SQL!

Showcase 2: Filme mit mehreren Sprachen

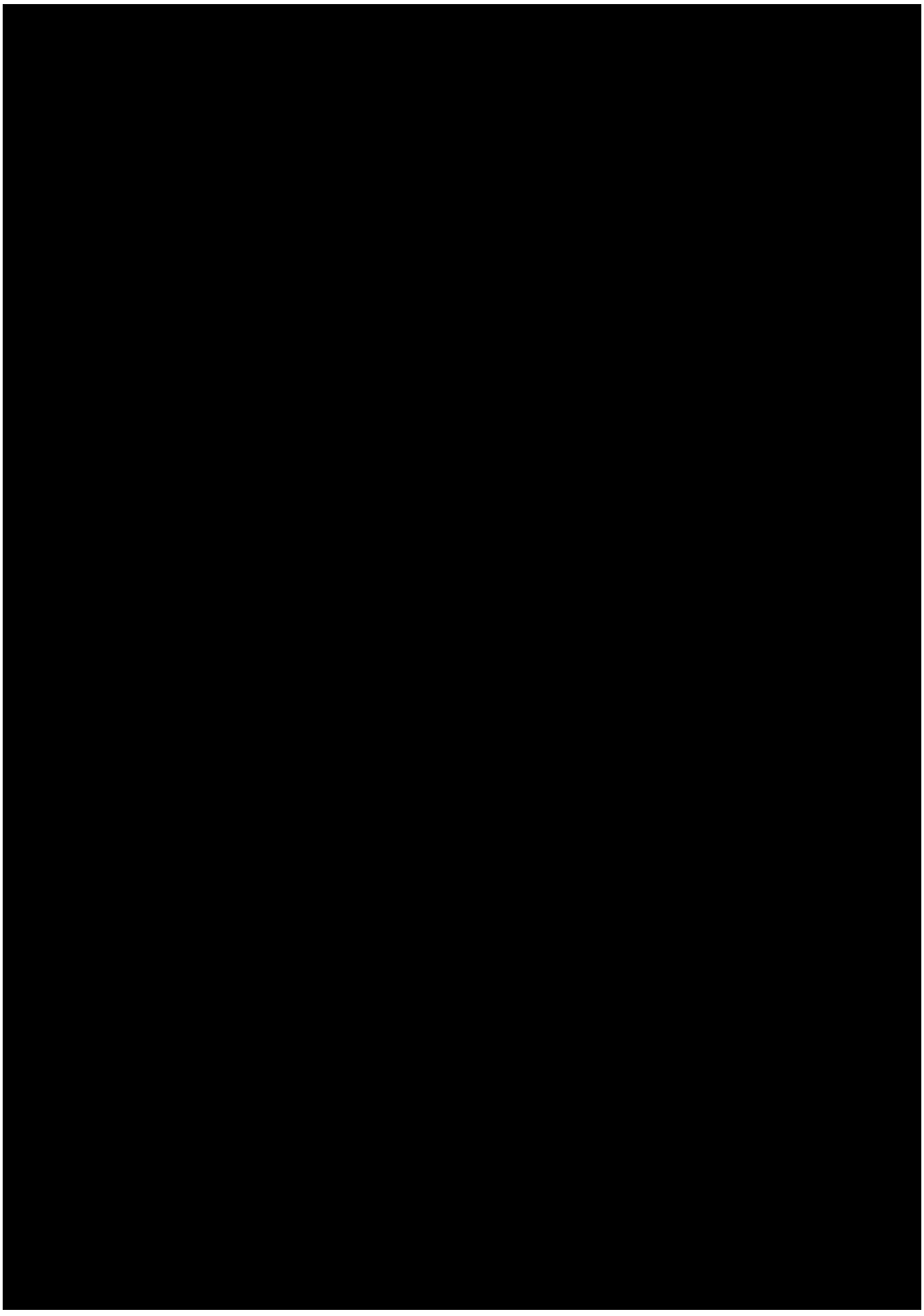
Welche Filme wurden in mehreren Sprachen veröffentlicht? Schauen wir uns die mehrsprachigsten Filme an!

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?filmName (COUNT(DISTINCT ?language) AS ?languageCount)
WHERE {
  ?film a dbo:Film .
  ?film rdfs:label ?filmName .
  ?film dbo:language ?language .

  FILTER(LANG(?filmName) = "en")
}
GROUP BY ?filmName
HAVING (COUNT(DISTINCT ?language) > 2)
ORDER BY DESC(?languageCount)
LIMIT 20
```



i	filmName	languageCount
0	"Elsewhere (2001 film)"@en	"11"^^http://www.w3.org/2001/XMLSchema#integer
1	"Viva la Muerte (film)"@en	"9"^^http://www.w3.org/2001/XMLSchema#integer
2	"How I Unleashed World War I I"@en	"8"^^http://www.w3.org/2001/XMLSchema#integer
3	"These Girls Are Missing"@en	"7"^^http://www.w3.org/2001/XMLSchema#integer
4	"The Third Half"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
5	"Kakabakaba Ka Ba?"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
6	"1770: Ek Sangram"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
7	"Ola Bola"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
8	"Code Unknown"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
9	"Suicide Killers"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
10	"Shinjuku Incident"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
11	"The Field Guide to Evil"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
12	"In the Room (film)"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
13	"The Last Communist"@en	"6"^^http://www.w3.org/2001/XMLSchema#integer
14	"Sign Gene"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer
15	"L: Change the World"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer

16	"The Gladiators (film)"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer
17	"Chasing the Dragon (film)"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer
18	"X500 (film)"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer
19	"The Ghosts Must Be Crazy"@en	"5"^^http://www.w3.org/2001/XMLSchema#integer

Interessant! Die Query zeigt Filme mit den meisten Sprachen. `COUNT(DISTINCT ...)` zählt eindeutige Werte, `HAVING` filtert nach der Aggregation – genau wie in SQL!

Showcase 3: Größte Städte in Deutschland

Welche deutschen Städte haben mehr als 500.000 Einwohner?

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?cityName ?population
WHERE {
  ?city a dbo:City .
  ?city dbo:country dbr:Germany .
  ?city rdfs:label ?cityName .
  ?city dbo:populationTotal ?population .

  FILTER(LANG(?cityName) = "de")
  FILTER(?population > 500000)
}
ORDER BY DESC(?population)
LIMIT 20
```

i	cityName	population
0	"Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
1	"Ost-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
2	"West-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
3	"Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
4	"Ost-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
5	"West-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
6	"Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
7	"Ost-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger
8	"West-Berlin"@de	"3596999"^^http://www.w3.org/2001/XMLSchema#nonNegativeInteger

Das ist die Power von DBpedia! Gleiche Query-Logik wie SQL – aber auf einem riesigen Knowledge Graph mit strukturierten Wikipedia-Daten!

Showcase 4: Fußballspieler nach Land

Wie viele Fußballspieler aus verschiedenen Ländern kennt DBpedia?

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?countryName (COUNT(?player) AS ?playerCount)
WHERE {
  ?player a dbo:SoccerPlayer .
  ?player dbo:birthPlace ?birthPlace .
```

```
  ?birthPlace dbo:country ?country .
  ?country rdfs:label ?countryName .

  FILTER(LANG(?countryName) = "en")
}
GROUP BY ?countryName
ORDER BY DESC(?playerCount)
LIMIT 15
```

i	countryName	playerCount
0	"Brazil"@en	"23253"^^http://www.w3.org/2001/XMLSchema#integer
1	"France"@en	"16964"^^http://www.w3.org/2001/XMLSchema#integer
2	"Argentina"@en	"13090"^^http://www.w3.org/2001/XMLSchema#integer
3	"Greece"@en	"4672"^^http://www.w3.org/2001/XMLSchema#integer
4	"China"@en	"4325"^^http://www.w3.org/2001/XMLSchema#integer
5	"Colombia"@en	"4093"^^http://www.w3.org/2001/XMLSchema#integer
6	"Chile"@en	"3912"^^http://www.w3.org/2001/XMLSchema#integer
7	"Croatia"@en	"3896"^^http://www.w3.org/2001/XMLSchema#integer
8	"Hungary"@en	"3694"^^http://www.w3.org/2001/XMLSchema#integer
9	"Indonesia"@en	"3571"^^http://www.w3.org/2001/XMLSchema#integer
10	"Bulgaria"@en	"3374"^^http://www.w3.org/2001/XMLSchema#integer
11	"Czech Republic"@en	"3236"^^http://www.w3.org/2001/XMLSchema#integer
12	"Iran"@en	"3156"^^http://www.w3.org/2001/XMLSchema#integer
13	"Australia"@en	"3141"^^http://www.w3.org/2001/XMLSchema#integer
14	"Belarus"@en	"2983"^^http://www.w3.org/2001/XMLSchema#integer

Aggregation über mehrere Joins! Das Pattern matcht Spieler → Geburtsort → Land, dann gruppiert und zählt. Wie SQL – nur mit Graph-Traversal!

Showcase 5: Filme mit Budgets

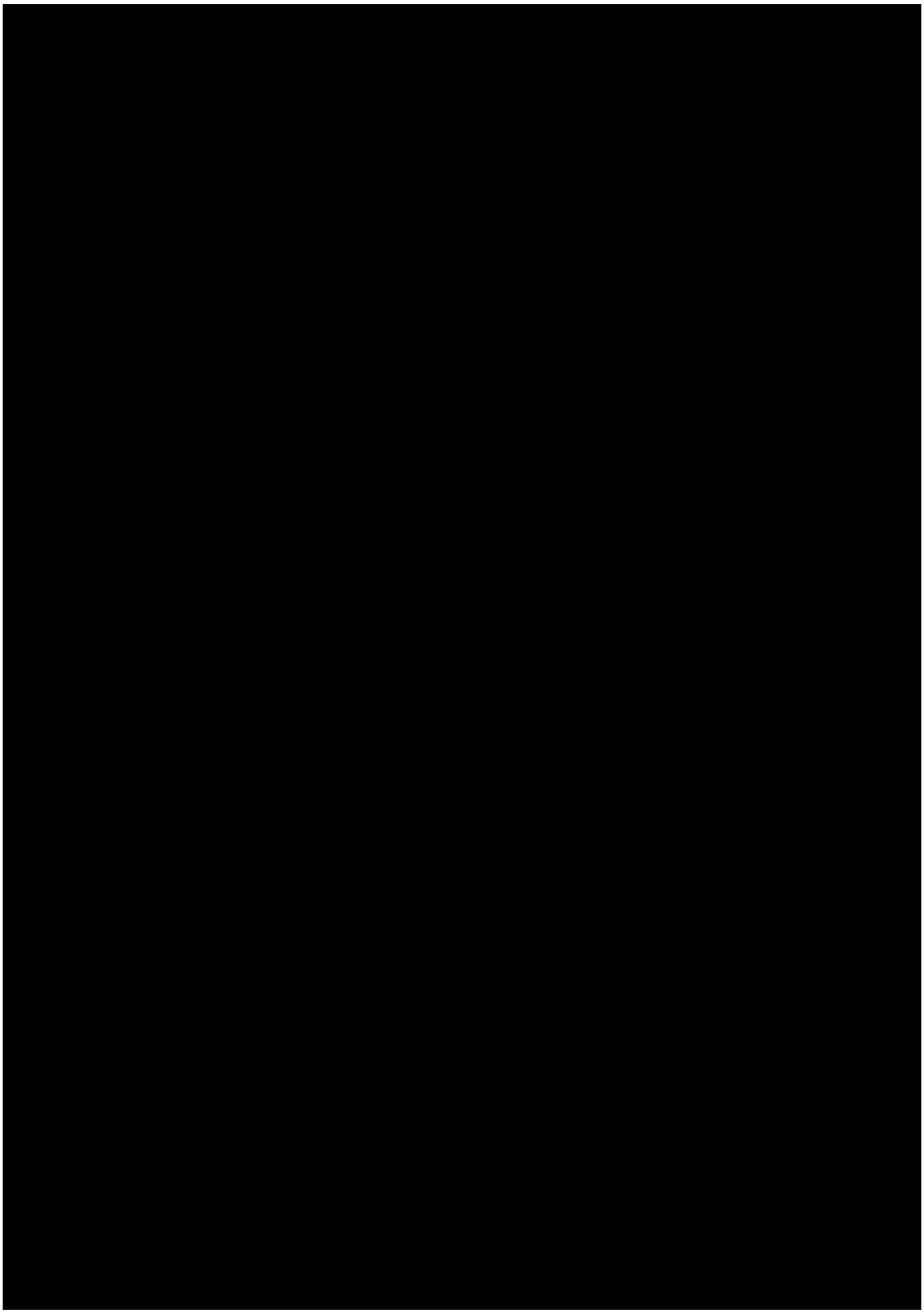
Welche Filme hatten die höchsten Budgets? Und wie viel haben sie eingespielt?

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?filmName ?budget ?gross
WHERE {
  ?film a dbo:Film .
  ?film rdfs:label ?filmName .
  ?film dbo:budget ?budget .
  ?film dbo:gross ?gross .

  FILTER(LANG(?filmName) = "en")
  FILTER(?budget > 200000000)
}
ORDER BY DESC(?budget)
LIMIT 20
```



i	filmName	budget	gross
0	"Badsha – The Don"@en	"6.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"6.4"^^http://dbpedia.org/datatype/bangladeshiTaka
1	"Badsha – The Don"@en	"6.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"6.4"^^http://dbpedia.org/datatype/bangladeshiTaka
2	"Badsha – The Don"@en	"6.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"6.4"^^http://dbpedia.org/datatype/bangladeshiTaka
3	"Boss 2: Back to Rule"@en	"5.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"10.5"^^http://dbpedia.org/datatype/indianRupee
4	"Boss 2: Back to Rule"@en	"5.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"10.5"^^http://dbpedia.org/datatype/indianRupee
5	"Boss 2: Back to Rule"@en	"5.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"10.5"^^http://dbpedia.org/datatype/indianRupee
6	"Jongli"@en	"2.05"^^http://dbpedia.org/datatype/bangladeshiTaka	"10.22"^^http://dbpedia.org/datatype/bangladeshiTaka
7	"Jongli"@en	"2.05"^^http://dbpedia.org/datatype/bangladeshiTaka	"10.22"^^http://dbpedia.org/datatype/bangladeshiTaka
8	"Utshob"@en	"2.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"11.0"^^http://dbpedia.org/datatype/bangladeshiTaka
9	"Utshob"@en	"2.0"^^http://dbpedia.org/datatype/bangladeshiTaka	"11.0"^^http://dbpedia.org/datatype/bangladeshiTaka
10	"Foxtrot Six"@en	"7.0E10"^^http://dbpedia.org/datatype/indonesianRupiah	"2.06E10"^^http://dbpedia.org/datatype/indonesianRupiah
11	"Foxtrot Six"@en	"7.0E10"^^http://dbpedia.org/datatype/indonesianRupiah	"2.06E10"^^http://dbpedia.org/datatype/indonesianRupiah
12	"Foxtrot Six"@en	"7.0E10"^^http://dbpedia.org/datatype/indonesianRupiah	"2.06E10"^^http://dbpedia.org/datatype/indonesianRupiah
13	"Darah Garuda"@en	"3.0E10"^^http://dbpedia.org/datatype/indonesianRupiah	"6.11139E9"^^http://dbpedia.org/datatype/indonesianRupiah
14	"Taxidermia"@en	"5.0E8"^^http://dbpedia.org/datatype/hungarianForint	"11408.0"^^http://dbpedia.org/datatype/usDollar

15	"Taxidermia"@en	"5.0E8"^^http://dbpedia.org/datatype/hungarianForint	"11408.0"^^http://dbpedia.org/datatype/usDollar
16	"Taxidermia"@en	"5.0E8"^^http://dbpedia.org/datatype/hungarianForint	"11408.0"^^http://dbpedia.org/datatype/usDollar
17	"Elk*rtuk"@en	"1.143E9"^^http://dbpedia.org/datatype/hungarianForint	"202300.0"^^http://dbpedia.org/datatype/usDollar
18	"Elk*rtuk"@en	"1.143E9"^^http://dbpedia.org/datatype/hungarianForint	"202300.0"^^http://dbpedia.org/datatype/usDollar
19	"Elk*rtuk"@en	"1.143E9"^^http://dbpedia.org/datatype/hungarianForint	"202300.0"^^http://dbpedia.org/datatype/usDollar

Optional könnten Sie hier auch `OPTIONAL { ?film dbo:gross ?gross }` nutzen, falls manche Filme kein Einspielergebnis haben. Dann würden sie trotzdem erscheinen!

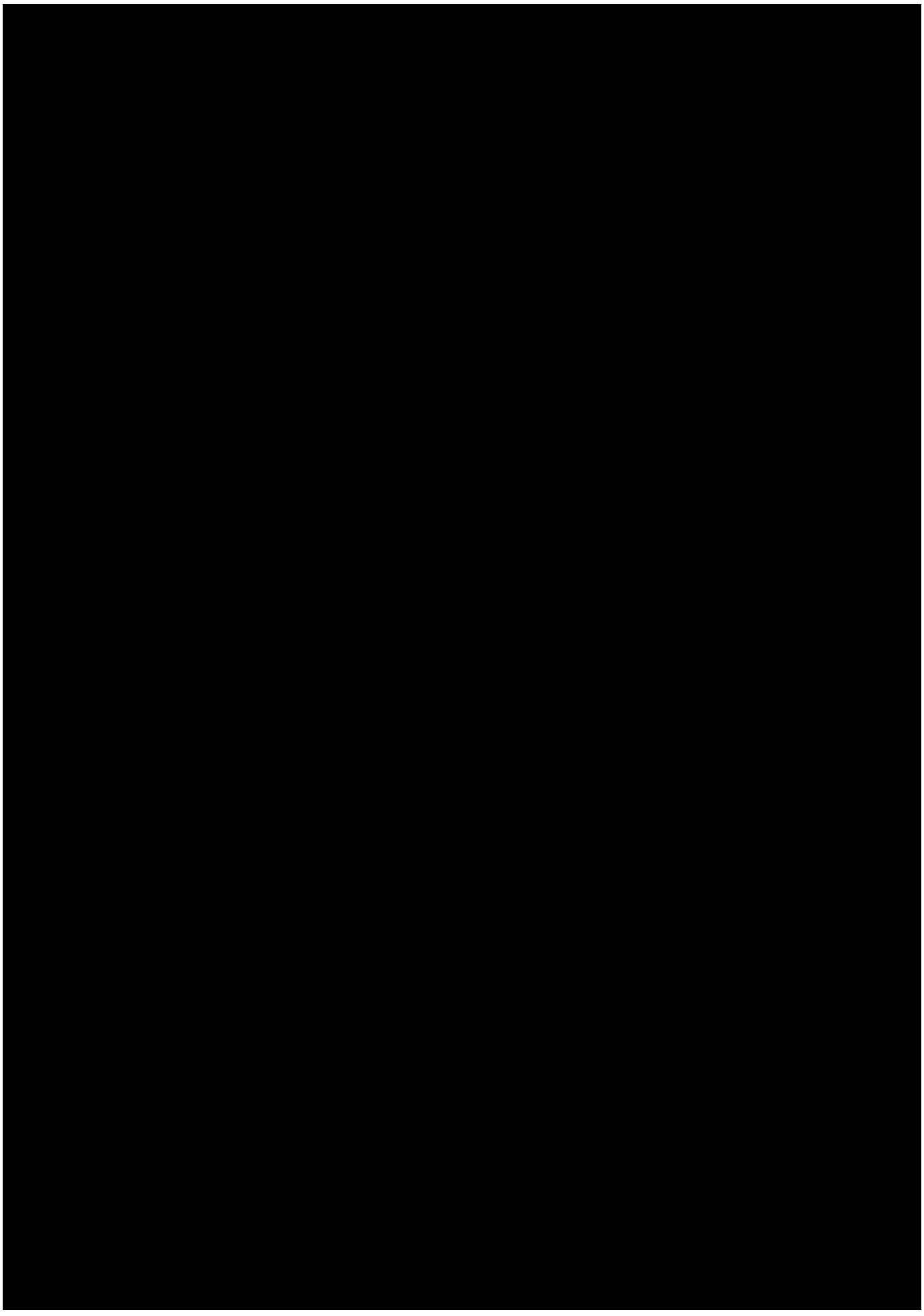
Showcase 6: Property Paths – Alle Alphabet-Unternehmen

Property Paths sind ein SPARQL-Feature ohne SQL-Äquivalent! Sie traversieren Graphen in beliebiger Tiefe. Finden wir alle Unternehmen, die zu Alphabet gehören:

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT DISTINCT ?sub ?p
FROM <http://dbpedia.org>
WHERE {
  ?sub ?p <http://dbpedia.org/resource/Alphabet_Inc.> .
  FILTER(?p IN (dbo:owner, dbp:owner, dbp:parent))
}
LIMIT 50
```



i	sub	p
0	http://dbpedia.org/resource/Vevo	http://dbpedia.org/property/owner
1	http://dbpedia.org/resource/Oyster_(company)	http://dbpedia.org/property/owner
2	http://dbpedia.org/resource/Dropcam	http://dbpedia.org/property/owner
3	http://dbpedia.org/resource/Liftware	http://dbpedia.org/property/owner
4	http://dbpedia.org/resource/YouTube	http://dbpedia.org/property/owner
5	http://dbpedia.org/resource/Flutter_(American_company)	http://dbpedia.org/property/owner
6	http://dbpedia.org/resource/Meka_Robotics	http://dbpedia.org/property/owner
7	http://dbpedia.org/resource/Google_DeepMind	http://dbpedia.org/property/owner
8	http://dbpedia.org/resource/Google_Nest	http://dbpedia.org/property/owner
9	http://dbpedia.org/resource/Songza	http://dbpedia.org/property/owner
10	http://dbpedia.org/resource/YouTube_Kids	http://dbpedia.org/property/owner
11	http://dbpedia.org/resource/Wavii	http://dbpedia.org/property/owner
12	http://dbpedia.org/resource/BufferBox	http://dbpedia.org/property/owner
13	http://dbpedia.org/resource/YouTube_TV	http://dbpedia.org/property/owner
14	http://dbpedia.org/resource/Vevo	http://dbpedia.org/ontology/owner
15	http://dbpedia.org/resource/Oyster_(company)	http://dbpedia.org/ontology/owner

16	http://dbpedia.org/resource/Dropcam	http://dbpedia.org/ontology/owner
17	http://dbpedia.org/resource/Liftware	http://dbpedia.org/ontology/owner
18	http://dbpedia.org/resource/YouTube	http://dbpedia.org/ontology/owner
19	http://dbpedia.org/resource/Google_Images__Google_Images__1	http://dbpedia.org/ontology/owner
20	http://dbpedia.org/resource/Meka_Robotics	http://dbpedia.org/ontology/owner
21	http://dbpedia.org/resource/Google_DeepMind	http://dbpedia.org/ontology/owner
22	http://dbpedia.org/resource/Flutter_(American_company)__Flutter_Ltd.__1	http://dbpedia.org/ontology/owner
23	http://dbpedia.org/resource/Google_Fiber__GFiber__1	http://dbpedia.org/ontology/owner
24	http://dbpedia.org/resource/Verily	http://dbpedia.org/ontology/owner
25	http://dbpedia.org/resource/Google_Nest	http://dbpedia.org/ontology/owner
26	http://dbpedia.org/resource/Songza	http://dbpedia.org/ontology/owner
27	http://dbpedia.org/resource/YouTube_Kids	http://dbpedia.org/ontology/owner
28	http://dbpedia.org/resource/Wavii	http://dbpedia.org/ontology/owner
29	http://dbpedia.org/resource/BufferBox	http://dbpedia.org/ontology/owner
30	http://dbpedia.org/resource/YouTube_TV__YouTube_TV__1	http://dbpedia.org/ontology/owner
31	http://dbpedia.org/resource/Makani_(company)	http://dbpedia.org/property/parent
32	http://dbpedia.org/resource/Google	http://dbpedia.org/property/parent

```
33 http://dbpedia.org/resource/Isomorphic_Labs http://dbpedia.org/property/parent
34 http://dbpedia.org/resource/Anvato http://dbpedia.org/property/parent
35 http://dbpedia.org/resource/Verily http://dbpedia.org/property/parent
36 http://dbpedia.org/resource/Wing_(company) http://dbpedia.org/property/parent
37 http://dbpedia.org/resource/Google_Energy http://dbpedia.org/property/parent
38 http://dbpedia.org/resource/Calico_(company) http://dbpedia.org/property/parent
39 http://dbpedia.org/resource/Loon_LLC http://dbpedia.org/property/parent
```

Das `+` ist der Property Path Operator! `dbo:parentCompany+` bedeutet „folge parentCompany ein oder mehrere Male“: Google → Alphabet, YouTube → Google → Alphabet, und so weiter. Rekursives Graph-Traversal – in SQL bräuchten Sie `WITH RECURSIVE` CTEs!

Property Path Operatoren:

Operator	Bedeutung	Beispiel
`/`	Pfad folgen (genau 1×)	`?person dbo:spouse/dbo:birthPlace ?city`
`+`	Ein oder mehrere Male	`?company dbo:parentCompany+ ?topCompany`
`*`	Null oder mehrere Male	`?company dbo:parentCompany* ?anyCompany`
`?`	Optional (0 oder 1×)	`?person dbo:spouse? ?partner`
`|`	Alternative Pfade	`?person dbo:birthPlace|dbo:hometown ?place`

Das ist die Power von Graph-Datenbanken: Beliebig tiefe Traversierungen ohne komplexe Queries!

Showcase 7: Federated Queries

Das Killer-Feature! Eine Query über mehrere SPARQL-Endpoints hinweg – DBpedia + Wikidata gleichzeitig!

```
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?cityName ?dbpediaPop ?wikidataPop
WHERE {
  ?city a dbo:City .
  ?city dbo:country <http://dbpedia.org/resource/Germany> .
  ?city rdfs:label ?cityName .
  ?city dbo:populationTotal ?dbpediaPop .

  # Federated Query zu Wikidata!
  SERVICE <https://query.wikidata.org/sparql> {
    ?wikidataCity wdt:P31 wd:Q515 .
    ?wikidataCity rdfs:label ?cityName .
    ?wikidataCity wdt:P1082 ?wikidataPop .
  }

  FILTER(LANG(?cityName) = "en")
  FILTER(?dbpediaPop > 500000)
}
LIMIT 10
```

```
Invalid SPARQL endpoint response from https://dbpedia.org/sparql (HTTP
status 500):
Virtuoso 42000 Error SQ070:SECURITY: Must have SELECT privileges on
view DB.DBA.SPARQL_SINV_2 for group ID 110 (SPARQL), user ID 110
(SPARQL)

SPARQL query:
# format: table
# source: https://dbpedia.org/sparql

PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT ?cityName ?dbpediaPop ?wikidataPop
WHERE {
  ?city a dbo:City .
  ?city dbo:country <http://dbpedia.org/resource/Germany> .
  ?city rdfs:label ?cityName .
  ?city dbo:populationTotal ?dbpediaPop .

  # Federated Query zu Wikidata!
  SERVICE <https://query.wikidata.org/sparql> {
    ?wikidataCity wdt:P31 wd:Q515 .
    ?wikidataCity rdfs:label ?cityName .
    ?wikidataCity wdt:P1082 ?wikidataPop .
  }

  FILTER(LANG(?cityName) = "en")
  FILTER(?dbpediaPop > 500000)
}
LIMIT 10
```

SERVICE ist das Federated-Query-Feature! Es fragt einen anderen SPARQL-Endpoint ab. So können Sie Daten aus verschiedenen Quellen kombinieren – das Linked Data Prinzip in Aktion!

Showcase 8: Komplexe Analyse – Alben pro Dekade

Zum Abschluss eine komplexere Analyse: Wie viele Alben wurden pro Dekade veröffentlicht?

```
1  # format: table
2  # source: https://dbpedia.org/sparql
3
```

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?decade (COUNT(?album) AS ?albumCount)
WHERE {
  ?album a dbo:Album .
  ?album dbo:releaseDate ?releaseDate .

  FILTER(?releaseDate >= "1950-01-01"^^xsd:date)
  FILTER(?releaseDate < "2030-01-01"^^xsd:date)

  BIND(FLOOR(YEAR(?releaseDate) / 10) * 10 AS ?decade)
}
GROUP BY ?decade
ORDER BY ?decade
```

i	decade	albumCount
0	"1950"^^http://www.w3.org/2001/XMLSchema#integer	"1"^^http://www.w3.org/2001/XMLSchema#integer
1	"1960"^^http://www.w3.org/2001/XMLSchema#integer	"2"^^http://www.w3.org/2001/XMLSchema#integer
2	"1970"^^http://www.w3.org/2001/XMLSchema#integer	"1"^^http://www.w3.org/2001/XMLSchema#integer
3	"1980"^^http://www.w3.org/2001/XMLSchema#integer	"13"^^http://www.w3.org/2001/XMLSchema#integer
4	"1990"^^http://www.w3.org/2001/XMLSchema#integer	"22"^^http://www.w3.org/2001/XMLSchema#integer
5	"2000"^^http://www.w3.org/2001/XMLSchema#integer	"63"^^http://www.w3.org/2001/XMLSchema#integer
6	"2010"^^http://www.w3.org/2001/XMLSchema#integer	"83"^^http://www.w3.org/2001/XMLSchema#integer
7	"2020"^^http://www.w3.org/2001/XMLSchema#integer	"36"^^http://www.w3.org/2001/XMLSchema#integer

Mathematik in SPARQL! `FLOOR(YEAR() / 10) * 10` rundet auf Dekaden. Das Ergebnis zeigt Trends in der Musikindustrie!

Teil 4: Semantic Web & Linked Data

Sie haben SPARQL in Aktion gesehen. Jetzt verstehen Sie, warum das Semantic Web wichtig ist!

Die vier Linked Data Prinzipien

Tim Berners-Lee's Regeln für Linked Data:

1. **Nutze URIs** für Dinge (nicht IDs!)
2. **Nutze HTTP URIs** (dereferencable – auflösbar!)
3. **Biete nützliche Informationen** wenn jemand die URI aufruft (RDF!)
4. **Verlinke zu anderen Dingen** (externe URIs)

Stellen Sie sich vor: Jede Website folgt diesen Regeln. Dann könnten Sie Daten aus beliebigen Quellen verknüpfen und abfragen – ohne APIs, ohne Integration!

Real-World Use Cases

Wo wird RDF/SPARQL produktiv eingesetzt?

- **Google Knowledge Graph** – Schema.org Markup auf Websites
- **BBC** – Nachrichtenartikel als RDF verknüpft
- **Pharma & Life Sciences** – Bio2RDF verbindet Forschungsdatenbanken
- **Regierungen** – Open Government Data (UK, US)
- **Libraries** – BIBFRAME für Bibliothekskataloge
- **E-Commerce** – Schema.org für Produktdaten (SEO!)

RDF ist keine akademische Spielerei – es steckt überall, wo strukturierte Daten im Web verknüpft werden!

SPARQL Endpoints: Überall abfragbar

Öffentliche SPARQL-Endpoints (probiert sie aus!):

- **DBpedia:** https://dbpedia.org/sparql
- **Wikidata:** https://query.wikidata.org/
- **Bio2RDF:** http://bio2rdf.org/sparql
- **GeoNames:** http://factforge.net/sparql
- **OpenStreetMap:** https://sophox.org/sparql

Jeder Endpoint ist wie eine öffentliche SQL-Datenbank – nur standardisiert mit SPARQL!

RDF-Technologien im Überblick

Das RDF-Ökosystem:

Technologie	Zweck	Beispiel
RDF	Datenmodell (Triples)	Turtle, RDF/XML, JSON-LD
RDFS	Schema-Definition (Klassen, Properties)	`rdfs:subClassOf`, `rdfs:domain`
OWL	Ontologien mit Logik	`owl:sameAs`, `owl:equivalentClass`
SPARQL	Query Language	`SELECT`, `CONSTRUCT`, `ASK`
Schema.org	Vokabular für Websites	`Person`, `Product`, `Event`
Microdata	HTML-Einbettung	`itemscope`, `itemprop`
JSON-LD	RDF als JSON	`@context`, `@type`, `@id`
Triple Stores	Datenbanken für RDF	Virtuoso, GraphDB, Jena

RDF ist nicht eine Technologie, sondern ein ganzes Ökosystem für das Semantic Web!

Wann RDF nutzen?

- ✅ **Knowledge Graphs** – vernetzte Wissensbasen
- ✅ **Linked Open Data** – öffentliche, verlinkte Daten
- ✅ **SEO & Microdata** – strukturierte Daten für Suchmaschinen
- ✅ **Integration** – verschiedene Datenquellen verknüpfen
- ✅ **Ontologien** – komplexe Domänenmodelle mit Reasoning

Wann NICHT RDF?

- ❌ **Hohe Performance** – relationale DBs sind schneller
- ❌ **einfache Anwendungen** – JSON reicht oft
- ❌ **proprietäre Systeme** – wenn Linked Data nicht wichtig ist
- ❌ **komplexe Aggregationen** – SQL ist mächtiger

Wrap-up & Quiz

Sie haben heute eine komplett neue Welt kennengelernt: das Semantic Web mit RDF und SPARQL!

Was Sie gelernt haben

✅ **RDF-Konzept:** Triples, URIs, Turtle-Syntax

✅ **SPARQL-Syntax:** SELECT, WHERE, FILTER, OPTIONAL, GROUP BY

✅ **Pattern Matching:** Implizite JOINs durch gemeinsame Variablen

✅ **DBpedia:** Real-world Knowledge Graph mit 15M Entities

✅ **Linked Data:** Vision des maschinenlesbaren Webs

✅ **Property Paths:** Graph-Traversal ohne SQL-Rekursion

✅ **Federated Queries:** Multiple Endpoints in einer Query

Quiz: RDF & SPARQL

1) Was ist ein RDF Triple?

- ◯ Eine dreifache Beziehung zwischen Tabellen
- ◯ Subject – Predicate – Object
- ◯ Ein dreifach normalisiertes Schema
- ◯ Drei verknüpfte Graphenknoten

2) Wie funktionieren JOINs in SPARQL?

- ◯ Mit `INNER JOIN` wie in SQL
- ◯ Implizit durch gemeinsame Variablen
- ◯ Mit `MATCH` wie in Cypher
- ◯ Gar nicht – SPARQL hat keine JOINs

3) Was macht `OPTIONAL { }` in SPARQL?

- ◯ Markiert optionale Felder im Schema
- ◯ Funktioniert wie LEFT JOIN (Pattern kann fehlen)
- ◯ Optimiert die Query-Performance
- ◯ Definiert Fallback-Werte

4) Was ist DBpedia?

- ◯ Eine neue SQL-Datenbank von Wikipedia
- ◯ Wikipedia-Inhalte als RDF Knowledge Graph
- ◯ Ein Python-Library für Datenbankzugriff
- ◯ Ein Property-Graph-Datenbank-System

5) Was bedeutet Linked Data?

- ◯ Datenbanken mit Foreign Keys
- ◯ Mehrere Nodes mit Relationships
- ◯ Web-Ressourcen verknüpft mit URIs und RDF
- ◯ GraphQL-APIs mit Nested Queries

Resources

Zum Weiterlesen:

- **W3C SPARQL Spec:** https://www.w3.org/TR/sparql11-query/
- **DBpedia SPARQL Endpoint:** https://dbpedia.org/sparql
- **Wikidata Query Service:** https://query.wikidata.org/
- **„Learning SPARQL“** (Bob DuCharme, O'Reilly)
- **LOD Cloud Diagram:** https://lod-cloud.net/

Das war's für heute! Ihr habt das Semantic Web kennengelernt – eine Vision eines maschinenlesbaren Webs mit strukturierten, verlinkten Daten. In der nächsten Vorlesung schauen wir uns an, wie man verschiedene Datenbank-Paradigmen in einer Architektur kombiniert: Polyglot Persistence!

Bis zur nächsten Session! 🎓