
# Teil XII

# NoSQL

# NoSQL


1. Motivation für NoSQL
2. Datenmodelle für NoSQL
3. KV-Stores und Wide Column
4. Document Stores
5. Graph Stores


# Motivation für NoSQL

# Motivation für NoSQL

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

NoSQL = Not only SQL

- im Umfeld vieler aktueller Buzzwords
  - NoSQL
  - Big Data
  - BASE
  - ....
- oft einfach als Etikett einer Neuentwicklung eines DBMS pauschal vergeben

# Was ist NoSQL?

- **SQL - No!**
  - SQL-Datenbanken sind zu komplex, nicht skalierbar, ...
  - man braucht was einfacheres!
- **Not only SQL**
  - SQL-Datenbanken haben zu wenig (oder die falsche) Funktionalität
  - Operationen auf Graphen, Data Mining Operatoren, ...
- **New SQL**
  - SQL-Datenbanken sind (software-technisch) in die Jahre gekommen
  - eine neue Generation von DBMS muss her (ohne die etablierten Vorteile von SQL zu ignorieren)

# Kritik an RDBMS / SQL

- nicht skalierbar
  - Normalisierung von Relationen, viele Integritätsbedingungen zu prüfen
  - **kann man in RDBMS auch vermeiden!**
- starre Tabellen nicht flexibel genug
  - schwach typisierte Tabellen (Tupel weichen in den tatsächlich genutzten Attributen ab)
    - viele Nullwerte wenn alle potentiellen Attribute definiert
    - alternativ Aufspaltung auf viele Tabellen
    - Schema-Evolution mit **alter table** skaliert bei Big Data nicht
  - **tatsächlich in vielen Anwendungen ein Problem**
- Integration von spezifischen Operationen (Graphtraversierung, Data-Mining-Primitive) mit Stored Procedures zwar möglich führt aber oft zu schwer interpretierbarem Code


# Datenmodelle für NoSQL

# Datenmodelle für NoSQL


- KV-Stores
- Wide Column Stores
- Dokumenten-orientierte Datenhaltung
- Graph-Speicher
- ....

# Anfragesprachen für NoSQL

- unterschiedliche Ansätze:
  - einfache funktionale API
  - Programmiermodell für parallele Funktionen
  - angelehnt an SQL-Syntax
  - ....


# KV-Stores und Wide Column

# Datenmodell: Key-Value-Stores

- *Key-Value-Store*: binäre Relationen, bestehend aus
  - einem Zugriffsschlüssel (dem Key) und
  - den Nutzdaten (dem Value)
- Nutzdaten
  - binäre Daten ohne Einschränkung,
  - Dateien oder Dokumente,
    $\rightarrow$ *Document Databases*
  - oder schwachstrukturierte Tupel
    $\rightarrow$ *Wide Column Store*

# Anfragen an KV-Stores

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

- einfache API

```
store.put(key, value)
value = store.get(key)
store.delete(key)
```

- aufgesetzte höherer Sprache angelehnt an SQL
- Map-Reduce
  - Framework zur Programmierung paralleler Datenaggregation auf KV-Stores

# Beispielsysteme für KV-Stores


- Amazon DynamoDB
- Riak

# Datenmodell: Wide Column


- Basisidee: KV-Store mit schwachstrukturiertem Tupel als Value
- Value = Liste von Attributname-Attributwert-Paaren
  - schwache Typisierung für Attributwerte (auch Wiederholgruppen)
- nicht alle Einträge haben die selben Attributnamen
  - offene Tupel
  - Hinzufügen eines neuen Attributs unproblematisch
  - Nullwerte aus SQL ersetzt durch fehlende Einträge
- Beispiel in DynamoDB

# Datenmodell: Wide Column /2

| Key | Value (Attributliste) | | |
|---|---|---|---|
| WeinID = 1 | Name = Zinfandel | Farbe = Rot | Jahrgang = 2004 |
| WeinID = 2 | Name = Pinot Noir | Weingut = {Creek, Helena} | |
| WeinID = 3 | Name = Chardonnay | Jahrgang = 2002 | Weingut = Bighorn |

# Anfragen bei Wide Column

- *CRUD*: Create, Read, Update und Delete
- in DynamoDB
  - `PutItem` fügt einen neuen Datensatz mit der gegebenen Attribut-Wert-Liste ein bzw. ersetzt einen existierenden Datensatz mit gleichem Schlüssel.
  - `GetItem`-Operation liest alle Felder eines über einen Primärschlüssel identifizierten Datensatzes.
  - `Scan` erlaubt einen Lauf über alle Datensätze mit Angabe von Filterkriterien.
- Aufruf über HTTP oder aus Programmiersprachen heraus

# Beispielanfrage in DynamoDB

```
POST / HTTP/1.1
x-amz-target: DynamoDB_20111205.GetItem
content-type: application/x-amz-json-1.0

{ "TableName": "Weine",
   " Key ":
      { "HashKeyElement": {"N": "1" }
        "RangeKeyElement": {"S": "Zinfandel" }
   },
   "AttributesToGet": ["Farbe", "Jahrgang"],
   "ConsistentRead": false
}
```

- Primärschlüssel (`HashKeyElement`) ist numerisch (`N`)
- Feld `Name` ist Bereichsschlüssel vom Typ String (`S`)

# Beispielanfrage in DynamoDB: Ergebnis

```
HTTP/1.1 200
x-amzn-RequestId: ...
content-type: application/x-amz-json-1.0
content-length: ...

{"Item":
   {"Farbe": {"S": "Rot" },
   "Jahrgang": {"N": "2004" }
   },
"ConsumedCapacityUnits": 0.5
}
```


# Document Stores

# Datenmodell: dokumentenorientierte Speicherung


- Basisidee: KV-Store mit (hierarchisch) strukturiertem Dokument als Value
- strukturiertes Dokument:
  - JSON-Format
    - geschachtelte Wide Column-Daten
  - XML (eher unüblich auf KV-Stores)

# Beispiel für Dokument in JSON

```
{
   "id" : "kritiker08154711",
   "Name" : "Bond",
   "Vorname" : "Jamie",
   "Alter" : 42,
   "Adresse" :
   {
      "Strasse" : "Breiter Weg 1",
      "PLZ" : 39007,
      "Stadt" : "Machdeburch"
   },
   "Telefon" : [7007, 110]
}
```

# Anfragen bei dokumentenorientierter Speicherung


- CRUD erweitert um dokumentspezifische Suche
- Beispiele (MongoDB mit BSON statt JSON)

```
db.kritiker.find({Name: "Bond"})
db.kritiker.find({Alter: 40})
db.kritiker.find({Alter{$lt: 50}})
db.kritiker.find({Name: "Bond", Alter: 42})
db.kritiker.find($or[{Name: "Bond"},
   { Alter: 42}}])
```

# Beispielsysteme für dokumentenorientierte Speicherung


- MongoDB
- CouchDB

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

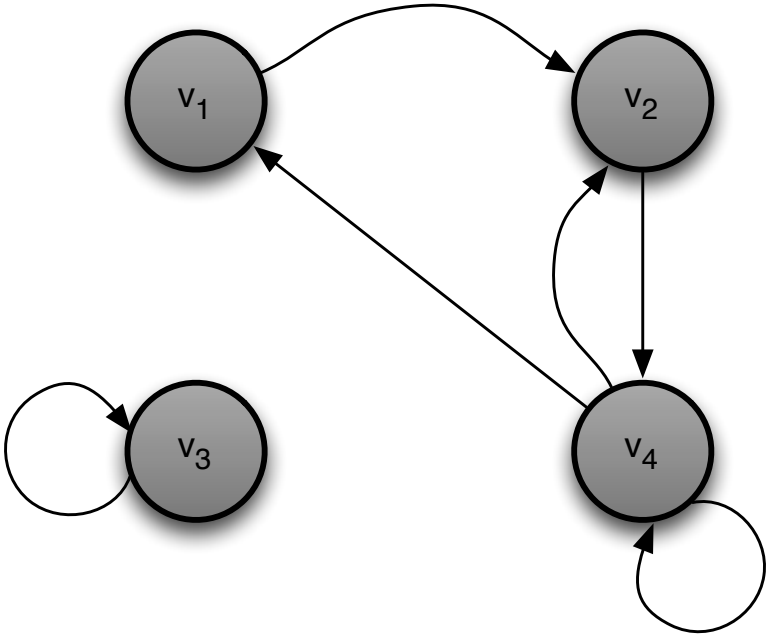
# Graph Stores

# Graph-Datenmodelle: Grundlagen

- spezielle Form der Datenrepräsentation = Graphen, insb. Beziehungen zwischen Objekten
- Anwendungsgebiete:
  - Transportnetze
  - Networking: Email-Verkehr, Mobilfunk-Nutzer
  - Soziale Netzwerke: Eigenschaften, Communities
  - Web: Verlinkte Dokumente
  - Chemie: Struktur chemischer Komponenten
  - Bioinformatik: Proteinstrukturen, metabolische Pathways, Genexpressionen

# Grundbegriffe

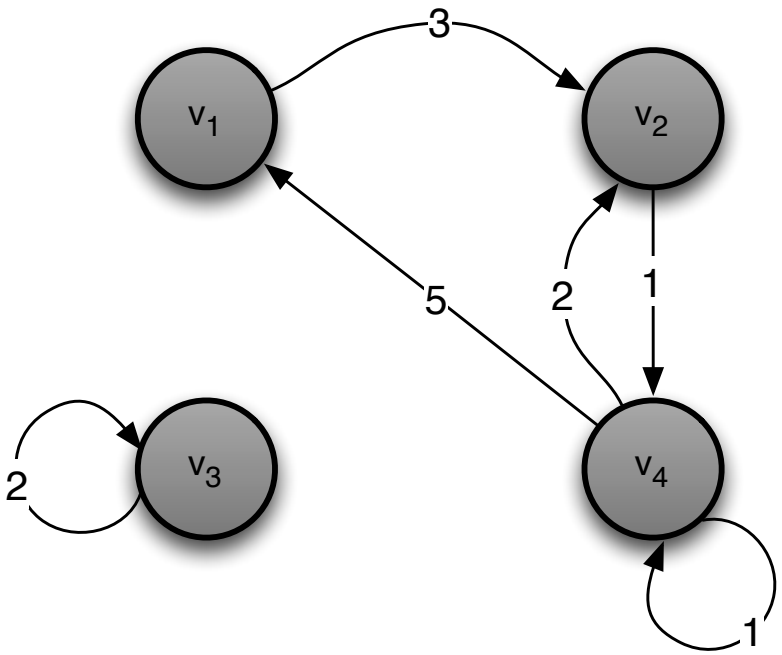
- Graph $G = (V, E)$
  - $V$: Menge der Knoten (vertices)
  - $E \subseteq V \times V$: Menge der Kanten (edges)




  - Kanten können mit Gewicht versehen werden

# Grundbegriffe: Adjazenzmatrix

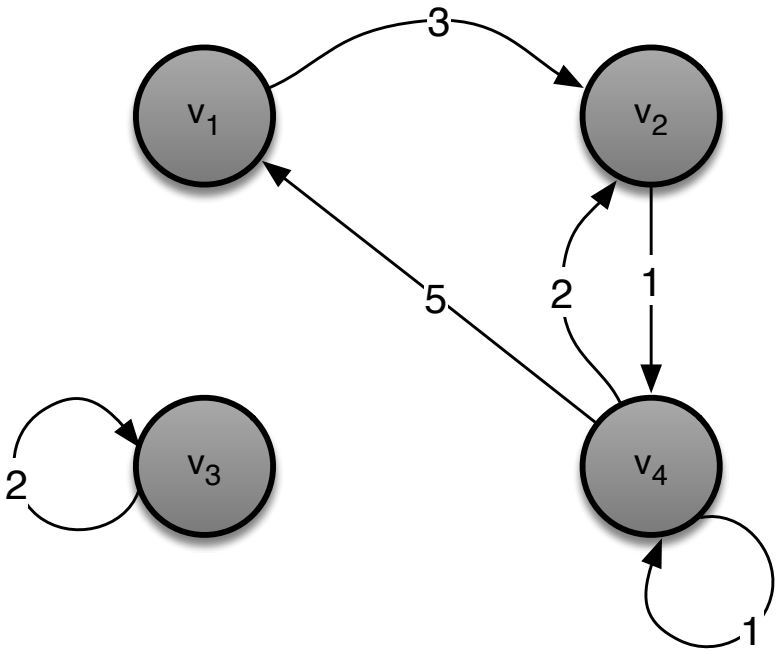
- Repräsentation von Graphen durch Matrix (Knoten als Zeilen und Spalten)
- ungerichteter Graph: symmetrische Matrix
- ungewichteter Graph: Zellen nur 0 oder 1




| von \ nach | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | 3 | 0 | 0 |
| $v_2$ | 0 | 0 | 0 | 1 |
| $v_3$ | 0 | 0 | 2 | 0 |
| $v_4$ | 5 | 2 | 0 | 1 |

# Grundbegriffe: Knotengrad

- Eigenschaft eines Knotens: Anzahl der verbundenen Knoten
- bei gerichteren Graphen: Unterscheidung in Eingangs- und Ausgangsgrad




*nach*

| *von* | $v_1$ | $v_2$ | $v_3$ | $v_4$ | *Ausgangsgrad* |
|---|---|---|---|---|---|
| $v_1$ | 0 | 1 | 0 | 0 | 1 |
| $v_2$ | 0 | 0 | 0 | 1 | 1 |
| $v_3$ | 0 | 0 | 1 | 0 | 1 |
| $v_4$ | 1 | 1 | 0 | 1 | 3 |
| *Eingangsgrad* | 1 | 2 | 1 | 2 | |

# Grundbegriffe: Traversierung

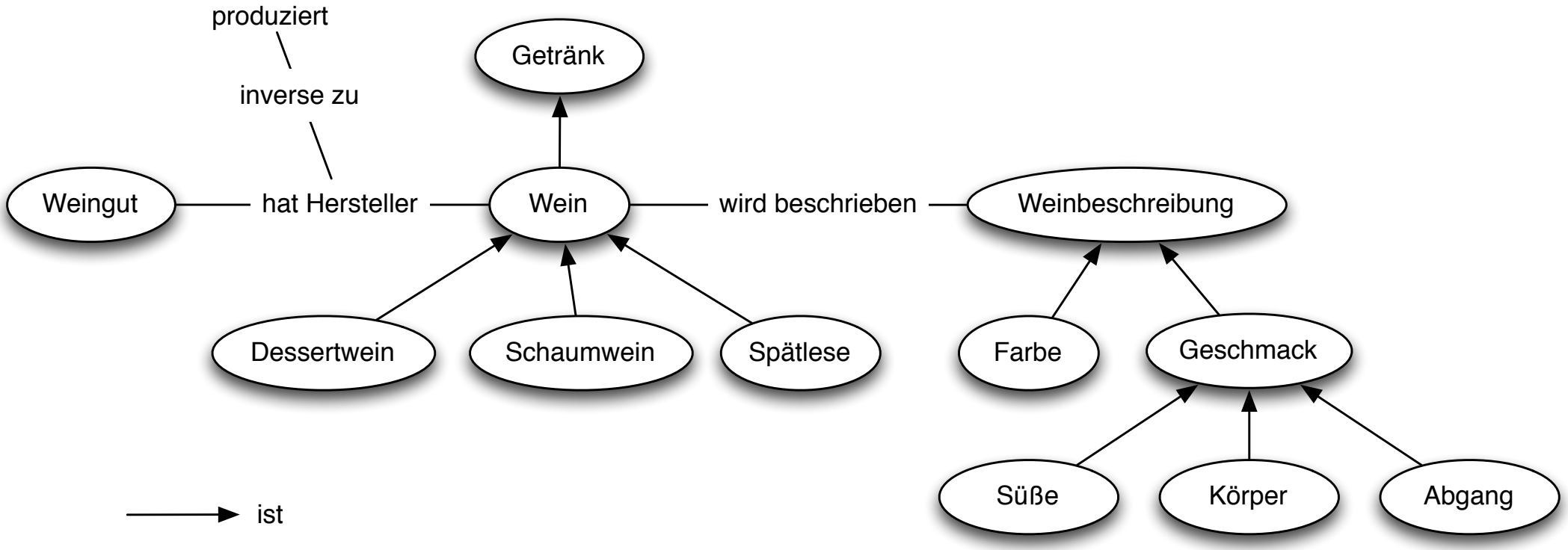
- Tiefensuche (DFS): zunächst rekursiv alle Kindknoten besuchen bevor alle Geschwisterknoten besucht werden
  - Bestimmung der Zusammenhangskomponente
  - Wegsuche um Labyrinth
- Breitensuche (BFS): zunächst alle Geschwisterknoten besuchen bevor die Kindknoten besucht werden
  - Bestimmung des kürzesten Weges

# Subjekt-Prädikat-Objekt-Modell: RDF

- Sprache zur Repräsentation von Informationen über (Web)-Ressourcen
- Ziel: automatisierte Verarbeitung
- zentraler Bestandteil von Semantic Web, Linked (Open) Data
- Repräsentation von Daten, aber auch Wissensrepräsentation (z.B. Ontologie)

# Ontologien

- Ontologie = formale Spezifikation einer Konzeptualisierung, d.h. einer Repräsentation von Begriffen (Konzepten) und deren Beziehungen
- Anwendung: Annotation von Daten, semantische Suche



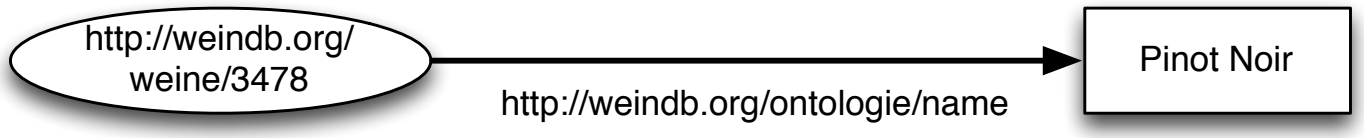
# RDF: Graphen & Tripel

- Graph = Menge von Tripeln, die Aussagen über Web-Ressourcen repräsentieren
- Identifikation der Web-Ressourcen über Uniform Resource Identifier (URI)
- Tripel:

```
subjekt prädikat objekt .
```

- Beispiel

```
<http://weindb.org/weine/2171> \
   <http://weindb.org/ontologie/name> "Pinot Noir".
```



# RDF: Graphen & Tripel

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

- **Subjekt**: URI-Referenz, d.h. Ressource, auf die sich die Aussage bezieht
- **Prädikat**: Eigenschaft, ebenfalls in Form einer URI-Referenz
- **Objekt**: Wert der Eigenschaft als Literal (Konstante) oder URI-Referenz

# RDF: Abkürzende Notation

- abkürzende Notation für Namensräume über Präfixe:

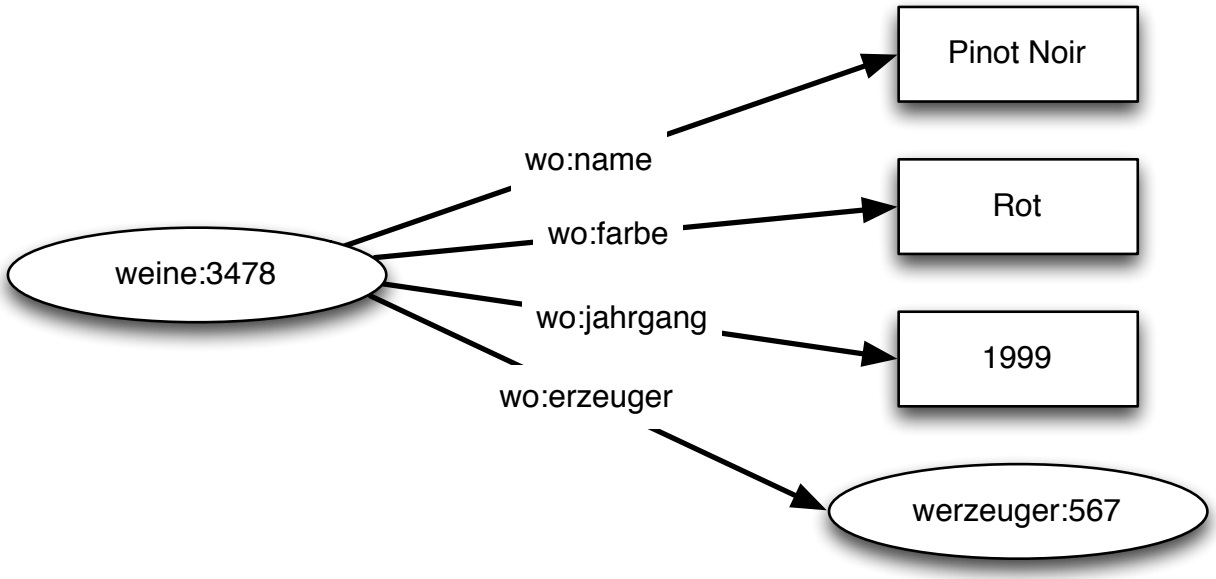
```
prefix wo: <http://weindb.org/ontologie/>
prefix weine: <http://weindb.org/weine/>

weine:2171 wo:name "Pinot Noir".
```

# RDF: Komplexe Graphen

- mehrere Aussagen zum gleichen Subjekt
- Objekte nicht nur Literale sondern selbst Objekte (URI)

```
weine:2171 wo:name "Pinot Noir".
weine:2171 wo:farbe "Rot".
weine:2171 wo:jahrgang "1999".
weine:2171 wo:erzeuger werzeuger:567 .
```



# RDF: Repräsentation, Schema und Vokabulare


- Repräsentation von RDF-Daten: N-Tripel (siehe oben), RDF/XML
- RDF Schema:
  - objektorientierte Spezifikationssprache
  - erweitert RDF um Typsystem: Definition von Klassen und Klassenhierarchien mit Eigenschaften, Ressourcen als Instanzen von Klassen
  - RDF Schema ist selbst RDF-Spezifikation

# RDF: Repräsentation, Schema und Vokabulare


- Beispiel RDF Schema

```
Wein rdf:type rdfs:Class .
Schaumwein rdf:type rdfs:Class .
Schaumwein rdfs:subClassOf Wein .
Name rdf:type rdf:Property .
Jahrgang rdf:type rdf:Property .
Jahrgang rdfs:domain Wein .
Jahrgang rdfs:range xsd:integer .
```

- für komplexere Ontologien: OWL (Web Ontology Language)

# RDF: Repräsentation, Schema und Vokabulare


- Vokabular: vordefinierte Klassen und Eigenschaften
  - Bsp: Dublin Core (Metadaten für Dokumente), FOAF (Soziale Netze), ...
  - wichtig z.B. für Linked Open Data

# SPARQL als RDF-Anfragesprache

- SPARQL Protocol And RDF Query Language: Anfragesprache für RDF
- W3C-Recommendation
- unterschiedliche Implementierungen möglich:
  - Aufsatz für SQL-Backends (z.B. DB2, Oracle)
  - Triple Stores (RDF-Datenbank)
  - SPARQL-Endpoints
- syntaktisch an SQL angelehnt, aber Unterstützung für Graph-Anfragen

# SPARQL-Elemente

- Grundelemente: `select-where`-Block und Tripelmuster

```
?wein wo:name ?name .
```

- Auswertung: finden aller Belegungen (Bindung) für Variable (`?name`) bei Übereinstimmung mit nicht-variablen Teilen

```
<http://weindb.org/weine/2171> wo:name "Pinot Noir".
<http://weindb.org/weine/2168> wo:name "Creek Shiraz".
<http://weindb.org/weine/2169> wo:name "Chardonnay".
```

# SPARQL: Basic Graph Pattern

- Graphmuster (BGP = Basic Graph Pattern): Kombination von Tripelmustern über gemeinsame Variablen

```
?wein wo:name ?name .
?wein wo:farbe ?farbe .
?wein wo:erzeuger ?erzeuger .
?erzeuger wo:weingut ?ename .
```

- Einsatz in SPARQL-Anfragen im `where`-Teil

```
select ?wein ?name ?farbe ?ename
where { ?wein wo:name ?name .
   ?wein wo:farbe ?farbe .
   ?wein wo:erzeuger ?erzeuger .
   ?erzeuger wo:weingut ?ename . }
```

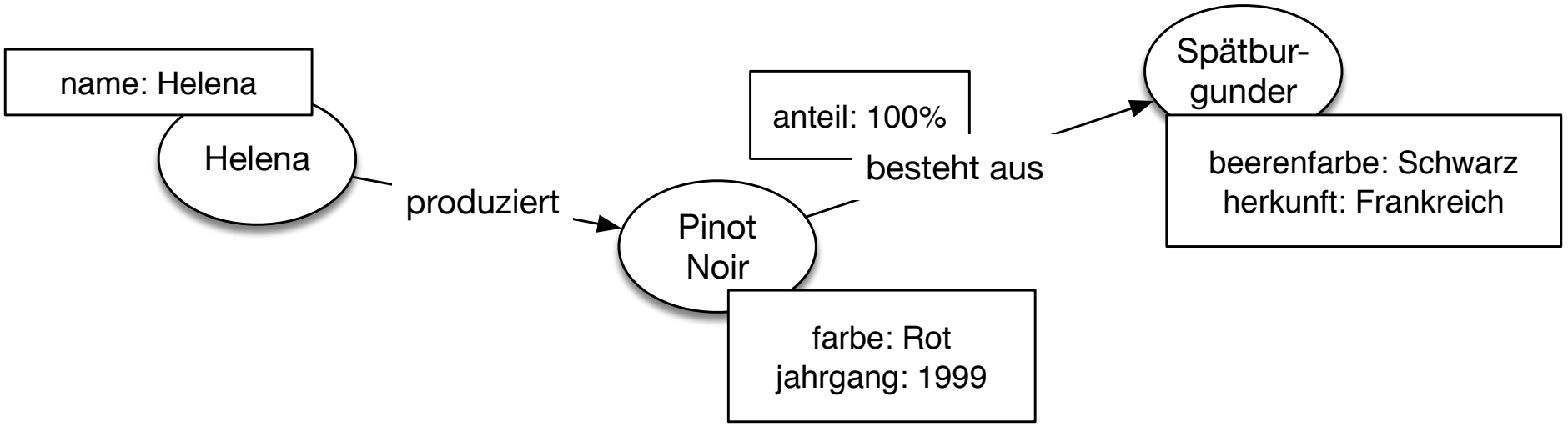
# SPARQL: Weitere Elemente

- `filter`: Filterbedingungen für Bindungen
- `optional`: optionale Muster – erfordern nicht zwingend ein Matching

```
prefix wo: <http://weindb.org/ontologie/>
select ?name
where { ?wein wo:name ?name . }
   optional { ?wein wo:jahrgang ?jahrgang } .
   filter ( bound(?jahrgang) && ?jahrgang < 2010 )
```

# Property-Graph-Modell

- Knoten und (gerichtete) Kanten mit Eigenschaften (Properties)
- nicht streng typisiert, d.h. Eigenschaften als Name-Wert-Paare
- Unterstützung in diversen Graph-Datenbanksystemen: neo4j, Microsoft Azure Cosmos DB, OrientDB, Amazon Neptune, ...

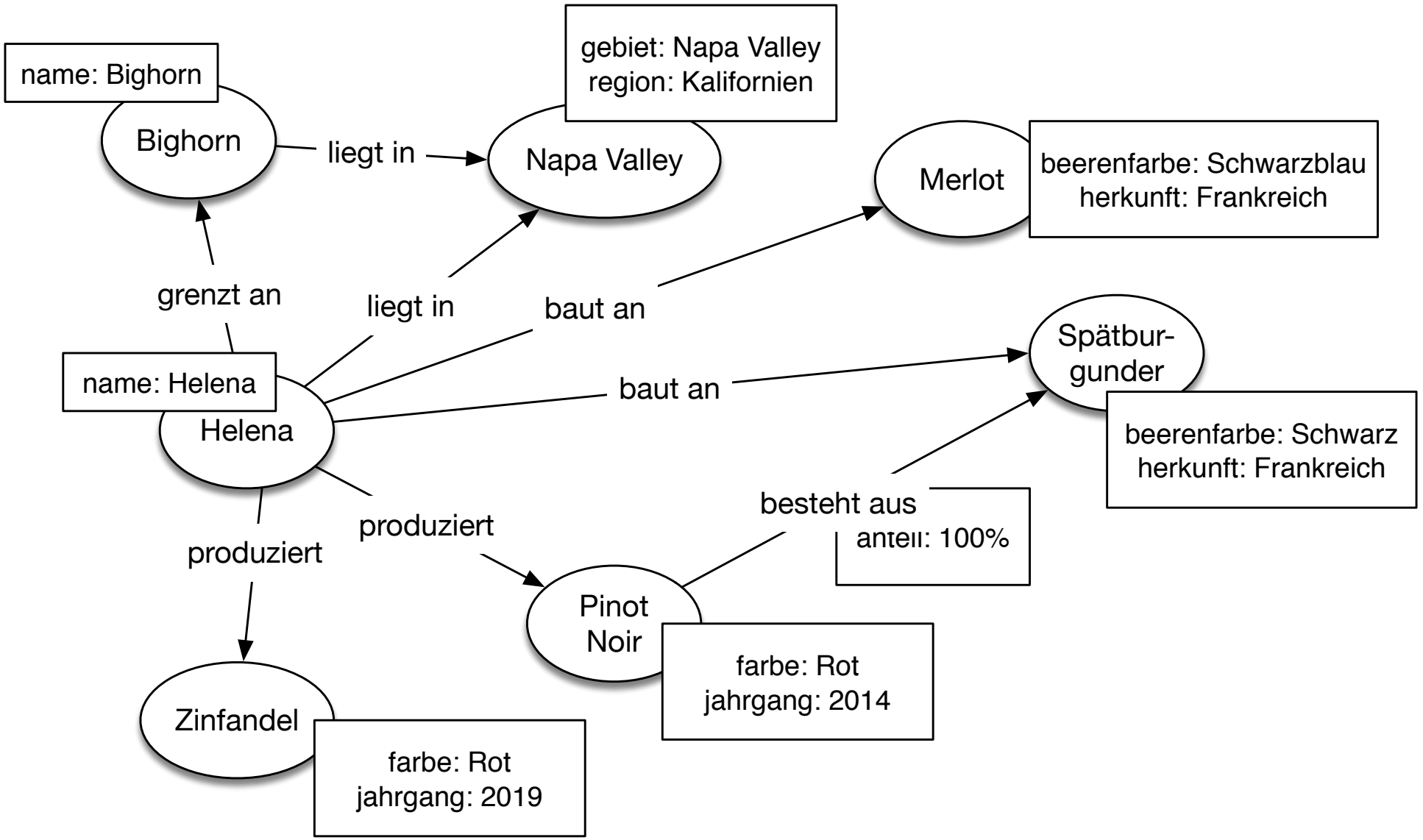


# Property-Graph-Modell in Neo4j

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

- Elemente: Nodes, Relationships, Properties, Labels
- Properties = Key-Value-Paare: Key (=String), Value (=Java-Datentypen + Felder)
- Nodes mit Labels ($\approx$ Klassenname)
- Relationships: sind gerichtet, mit Namen und ggf. Properties

# Property-Graph-Modell: Beispiel



# Anfragen auf Graphen

- keine Standardsprache
- aber wiederkehrende Grundelemente
  - Graph Matching: Knoten, Kanten, Pfade (siehe BGP in SPARQL)
  - Filter für Knoten- und Kanteneigenschaften
  - Konstruktion neuer Graphen
- hier: Cypher (neo4j)

# Anfragen in Cypher

- Basis: Muster der Form „Knoten → Kante → Knoten . . . “

```
(von)-[:relationship]->(nach)
```

- Beschränkung über Label und Properties

```
(e:ERZEUGER)-[:LIEGT_IN]->(a:ANBAUGEBIET {
      gebiet: 'Napa Valley' } )
```

# Cypher: Klauseln

- `match`: Beispielmuster für Matching
- `return`: Festlegung der Rückgabedaten (Projektion)
- `where`: Filterbedingung für „gematchte“ Daten
- `create`: Erzeugen von Knoten oder Beziehungen
- `set`: Ändern von Property-Werten
- `...`

# Cypher: Beispiele

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

- Anlegen von Daten

```
create
   (napavalley:ANBAUGEBIET {
      gebiet: 'Napa Valley', region: 'Kalifornien' }),
   (helena:ERZEUGER { name: 'Helena' }),
   ...
   (helena)-[:LIEGT_IN]->(napavalley),
   ...
```

# Cypher: Beispiele

- Alle Weingüter aus dem Napa Valley

```
match (e:ERZEUGER)-[:LIEGT_IN]->(a:ANBAUGEBIET {
      gebiet: 'Napa Valley' })
return e
```

- Alle Weingüter, die die Merlot-Traube anbauen

```
match (r:REBE { name: 'Merlot' })<-[:BAUT_AN]-(w) \
   -[:LIEGT_IN]->(g)
return g
```

# Cypher: Beispiele /2

- Alle Weingüter, die Weine mit eiem Spätburgunder-Anteil von mehr als 50% produzieren sowie die Anzahl dieser Weine pro Weingut

```
match (e:ERZEUGER)-[:PRODUZIERT]->(w:WEIN)-
   [b:BESTEHT_AUS]->(r:REBE { name: 'Spätburgunder' })
where b.anteil > 50
return e.name, count(w.name)
```

- Alle Weingüter, direkt an das Weingut Helena grenzen oder an ein Weingut, das direkt an Helena grenzt

```
match (e1:ERZEUGER { name: 'Helena' })-[:GRENZT_AN*..2] \
   -(e2:ERZEUGER)
return e2
```

# Cypher: Beispiele /3

- alle Knoten des Typs `WEINE`

```
match (w)
where w:WEINE
return w
```

- Knotengrade pro Knoten im Graph

```
match (n)-[r]-()
return n, count(r)
```

# Zusammenfassung

- NoSQL als Oberbegriff für diverse Datenbanktechniken
- große Bandbreite: von einfachen KV-Stores bis zu Graphdatenbanken
- höhere Skalierbarkeit / Performance gegenüber SQL-DBMS meist durch Einschränkungen erkauft
  - Abschwächung von ACID-Eigenschaften
  - begrenzte Anfragefunktionalität
  - Nicht-Standard bzw. proprietäre Schnittstellen

# Weiterführende Literatur

- Lena Wiese: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. De Gruyter / Oldenburg, 2015
- Ian Robinson, Jim Webber, Emil Eifrem: Graph Databases. O'Reilly, 2015