ulm university universität ulm

Semantic Web Grundlagen

SPARQL Implementierungstechniken

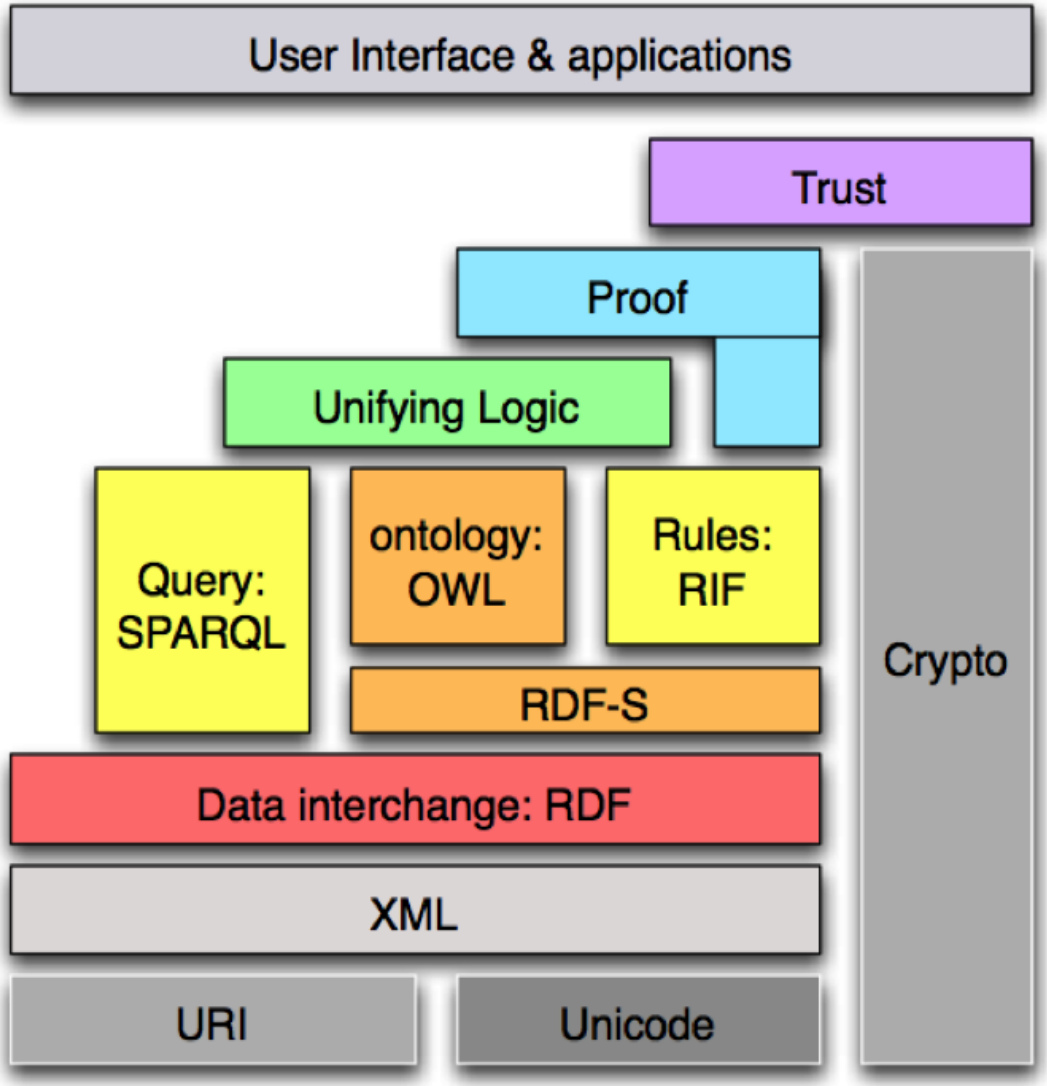
Birte Glimm
Institut für Künstliche Intelligenz | 19. Jan 2012

Organisatorisches: Inhalt

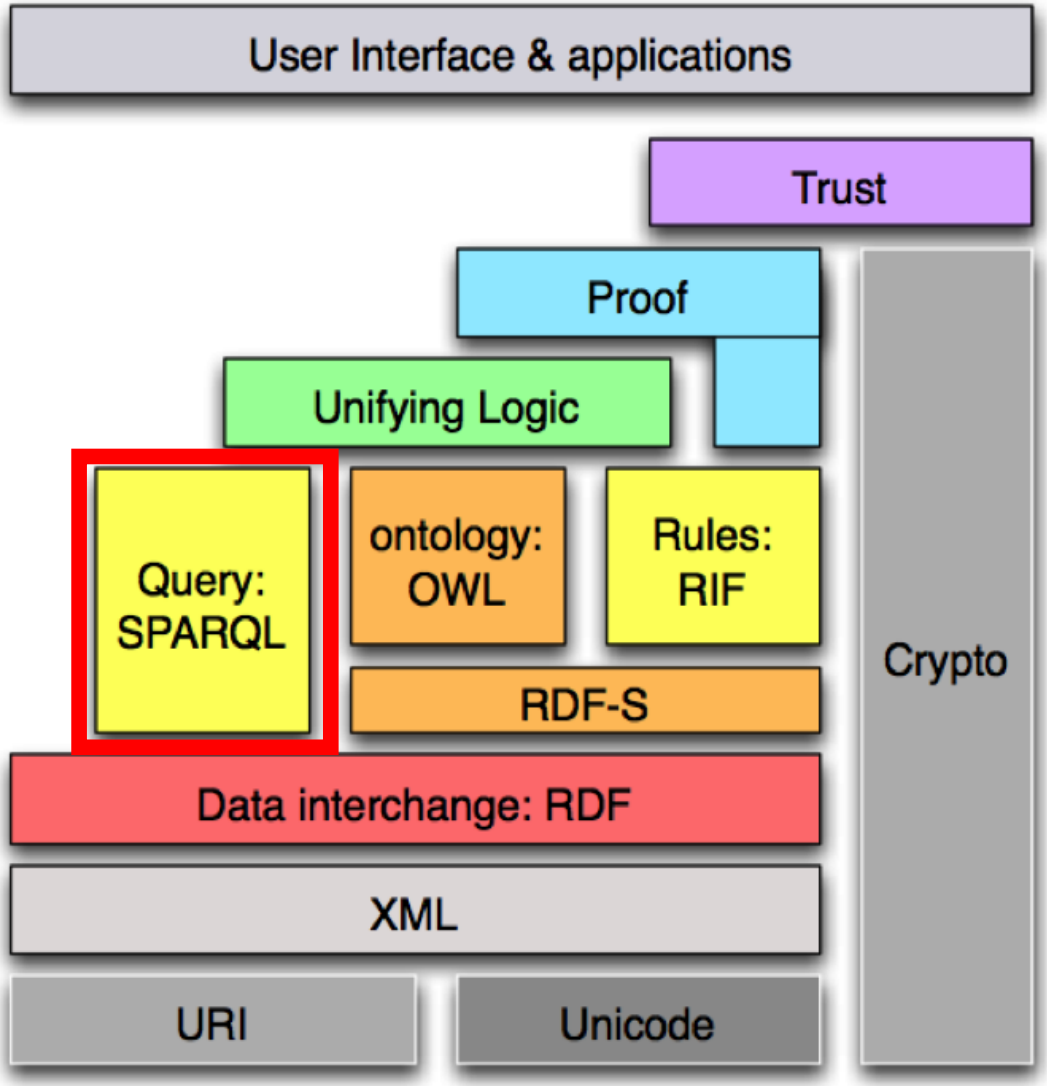
Einleitung und XML	17. Okt	Hypertableau II	12. Dez
Einführung in RDF	20. Okt	Übung 4	15. Dez
RDF Schema	24. Okt	SPARQL Syntax & Intuition	19. Dez
fällt aus	27. Okt	SPARQL Semantik	22. Dez
Logik – Grundlagen	31. Okt	SPARQL 1.1	9. Jan
Übung 1	3. Nov	Übung 5	12. Jan
Semantik von RDF(S)	7. Nov	SPARQL Entailment	16. Jan
RDF(S) & Datalog Regeln	10. Nov	SPARQL Implemetierung	19. Jan
OWL Syntax & Intuition	14. Nov	Ontology Editing	23. Jan
Übung 2	17. Nov	Übung 6	26. Jan
OWL & BLs	21. Nov	Ontology Engineering	30. Jan
OWL 2	24. Nov	Linked Data	2. Feb
Tableau	28. Nov	SemWeb Anwendungen	6. Feb
Übung 3	1. Dez	Übung 7	9. Feb
Blocking & Unravelling	5. Dez	Wiederholung	13. Feb
Hypertableau	8. Dez	Übung 8	16. Feb

Abfragen und RIF wurde gestrichen

Die Abfragesprache SPARQL



Die Abfragesprache SPARQL



Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

Einführung und Motivation

- Die Menge an Daten im Semantic Web wächst kontinuierlich
  - Mehr als 25 Milliarden RDF Triple im Web verfügbar (Linked Data)
  - DBPedia: 3,4 Millionen Entitäten, 1 Milliarde Triple
- Effiziente Techniken zur Auswertung von Anfragen essentiell
- Wie können wir den Bgp$(\cdot)$ Operator effizient auswerten (ohne Entailment)?
- In welcher Reihenfolge sollten die anderen Operatoren ausgewertet werden?

Mehr und Mehr Tripel

- Billion Triple Challenge 2008: 1 Milliarde Tripel
- Billion Triple Challenge 2010: 3 Milliarden Tripel
- Billion Triple Challenge 2011: 2 Milliarden Tripel
- http://www.w3.org/wiki/LargeTripleStores:
  - BigOWLIM: 12 Milliarden Tripel (Juni 2009)
  - Garlik 4store: 15 Milliarden Tripel (Oktober 2009)
  - OpenLink Virtuoso: 15.4 Milliarden Tripel
  - AllegroGraph: 1+ Billionen Tripel

Und Komplexe Abfragemuster

```
SELECT DISTINCT ?a ?b ?lat ?long WHERE
{ ?a dbpedia:spouse ?b.
  ?a dbpedia:wikilink dbpediares:actor.
  ?b dbpedia:wikilink dbpediares:actor.
  ?a dbpedia:placeOfBirth ?c.
  ?b dbpedia:placeOfBirth ?c.
  ?c owl:sameAs ?c2.
  ?c2 pos:lat ?lat.
  ?c2 pos:long ?long.
}
```

Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

RDF in Row Stores

Row Store:

- Relationale Datenbank, die Tripel in Reihen einer Relation speichert
- MySQL, PostgreSQL, Oracle, DB2, SQLServer, ...

Zugrunde liegendes Prinzip:

- Tripel werden in einer (sehr großen) Tabelle mit drei Attributen gespeichert (s, p, o)
- SPARQL Abfragen werden in SQL Abfragen umgeschrieben
- Die Datenbank übernimmt die Auswertung
  - Strings werden oft auf eindeutige Integer IDs abgebildet
  - Verwendet von vielen Tripel Stores z.B. 3Store, Jena, HexaStore, RDF-3X, ...

RDF in Row Stores

Row Store:

- Relationale Datenbank, die Tripel in Reihen einer Relation speichert
- MySQL, PostgreSQL, Oracle, DB2, SQLServer, ...

Zugrunde liegendes Prinzip:

- Tripel werden in einer (sehr großen) Tabelle mit drei Attributen gespeichert (s, p, o)
- SPARQL Abfragen werden in SQL Abfragen umgeschrieben
- Die Datenbank übernimmt die Auswertung
  - Strings werden oft auf eindeutige Integer IDs abgebildet
  - Verwendet von vielen Tripel Stores z.B. 3Store, Jena, HexaStore, RDF-3X, ...

Einfache Erweiterung auf Quadrupel (Graph, Subjekt, Prädikat, Objekt) Hier aber Fokus auf Tripel

Beispiel Tripel Tabelle

```
ex:Sebastian  ex:lehrt          ex:EinführungKI ;
              ex:arbeitetFür    ex:KIT ;
              ex:drVon          ex:TUDresden .
ex:Birte      ex:lehrt          ex:SemWebGrundlagen ;
              ex:arbeitetFür    ex:UUlm ;
              ex:drVon          ex:UoManchester .
ex:Markus     ex:lehrt          ex:KR&R ;
              ex:drVon          ex:KIT ;
              ex:arbeitetFür    ex:UoOxford ,
                                ex:SemanticWiki .
```

Subjekt	**Prädikat**	**Objekt**
ex:Sebastian	ex:lehrt	ex:EinführungKI
ex:Sebastian	ex:arbeitetFür	ex:KIT
ex:Sebastian	ex:drVon	ex:TUDresden
ex:Birte	ex:lehrt	ex:SemWebGrundlagen
ex:Birte	ex:arbeitetFür	ex:UUlm
ex:Birte	ex:drVon	ex:UoManchester
ex:Markus	ex:lehrt	ex:KR&R
ex:Markus	ex:arbeitetFür	ex:UoOxford
ex:Markus	ex:arbeitetFür	ex:SemanticWiki
ex:Markus	ex:drVon	ex:KIT

Umwandlung von SPARQL in SQL Abfragen

Prinzipieller Ansatz:

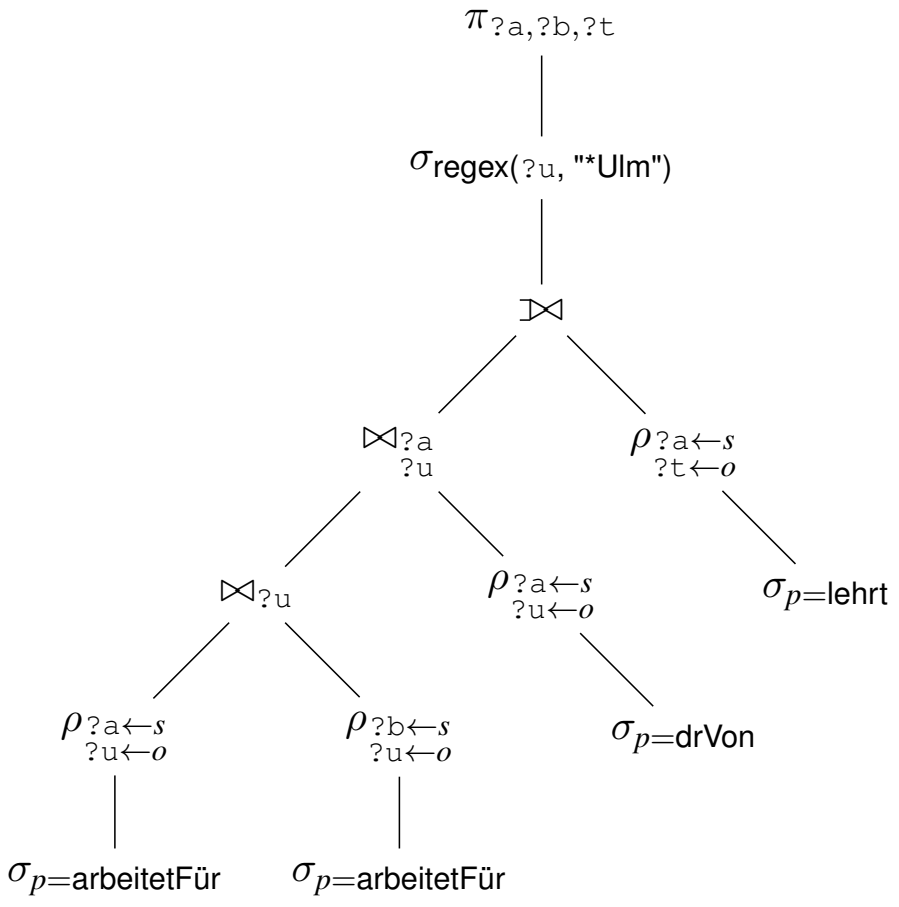
- Für jedes Tripel Muster eine Kopie der Tripel Tabelle
- Konstanten im Abfragemuster führen zu Bedingungen
- Gemeinsame Variablen in Mustern führen zu Joins
- FILTER Bedingungen erzeugen Constraints
- OPTIONAL Klauseln erzeugen Outer Joins
- UNION Klauseln erzeugen Union Ausdrücke

Beispiel: Umwandlung von SPARQL in SQL Abfragen

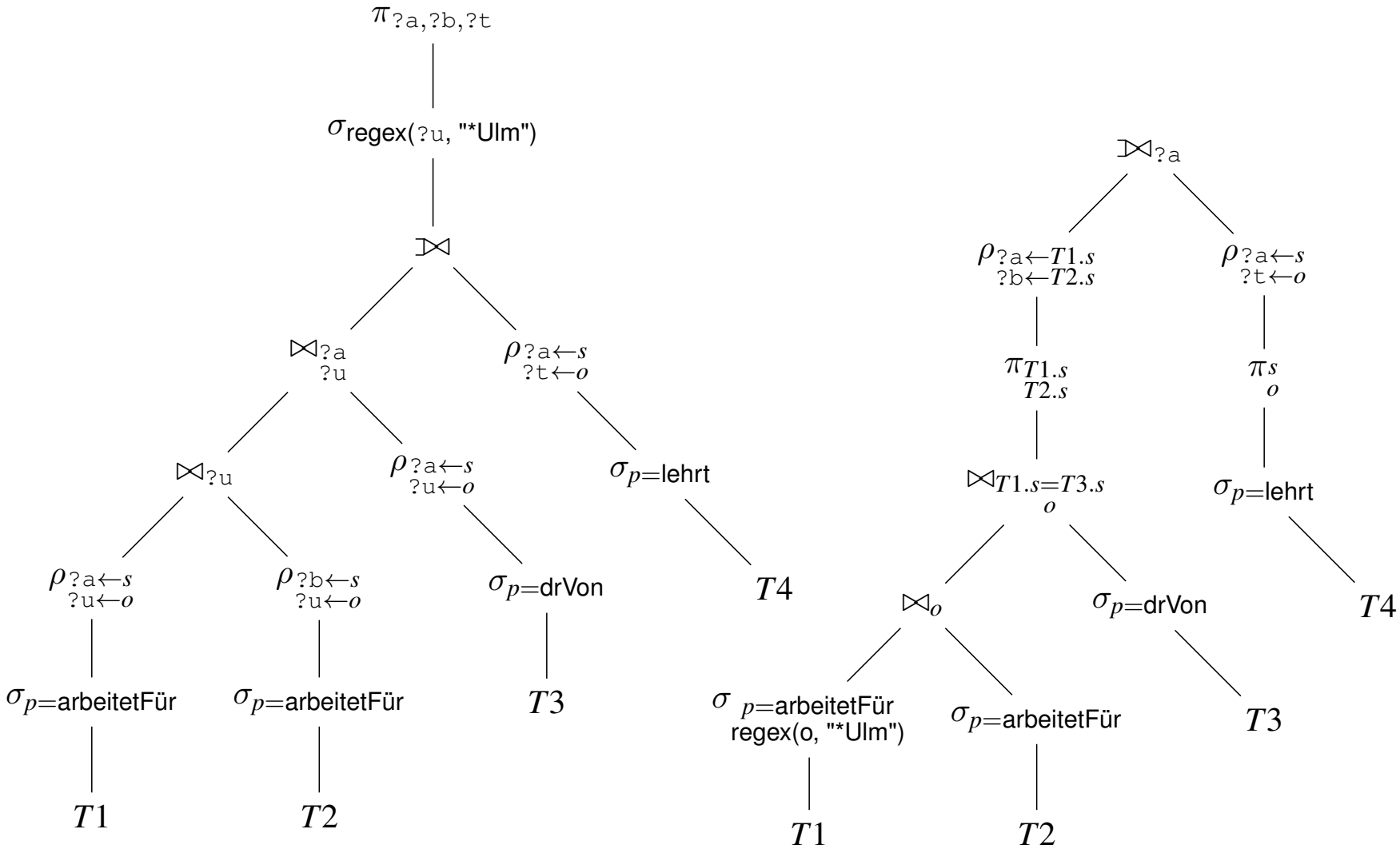
```
SELECT ?a ?b ?t WHERE {
  ?a arbeitetFür ?u . ?b arbeitetFür ?u . ?a drVon ?u .
  OPTIONAL { ?a lehrt ?t } FILTER (regex(?u, "*Ulm"))}
```

Algebra Übersetzung:

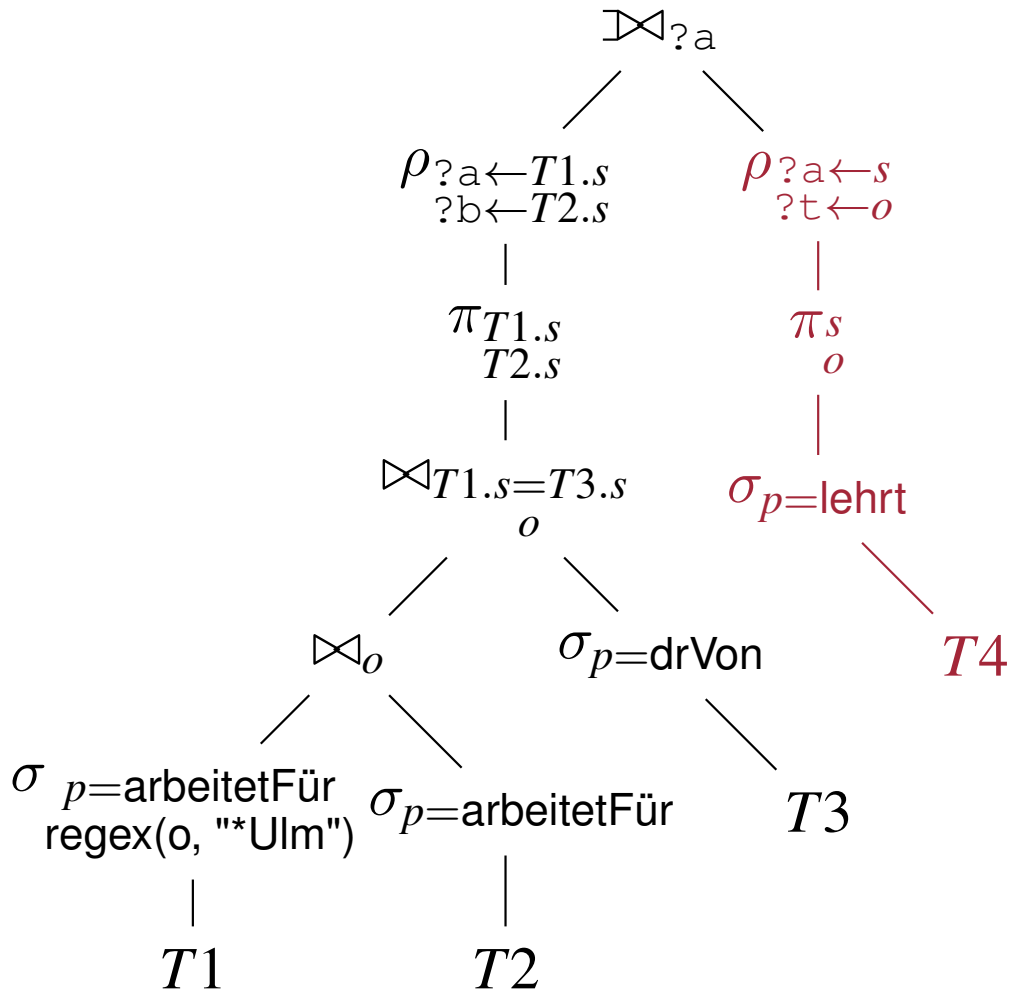
```
Project(
 Filter(
  regex(?u, "*Ulm"),
  LeftJoin(
   Bgp(...),
   Bgp(...),
   true
  )
 ),
{?a, ?b, ?t })
```



Beispiel: Algebra Optimierung

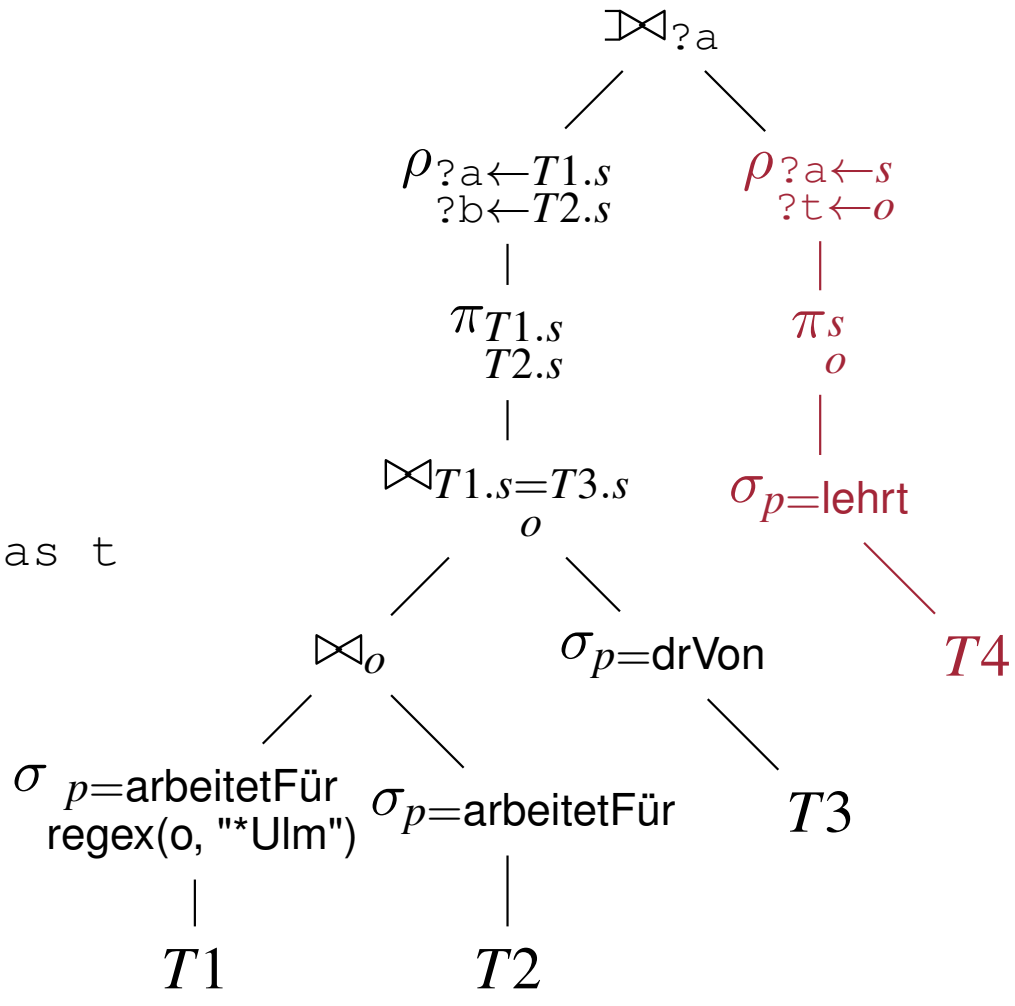


Beispiel: Umwandlung von SPARQL in SQL Abfragen



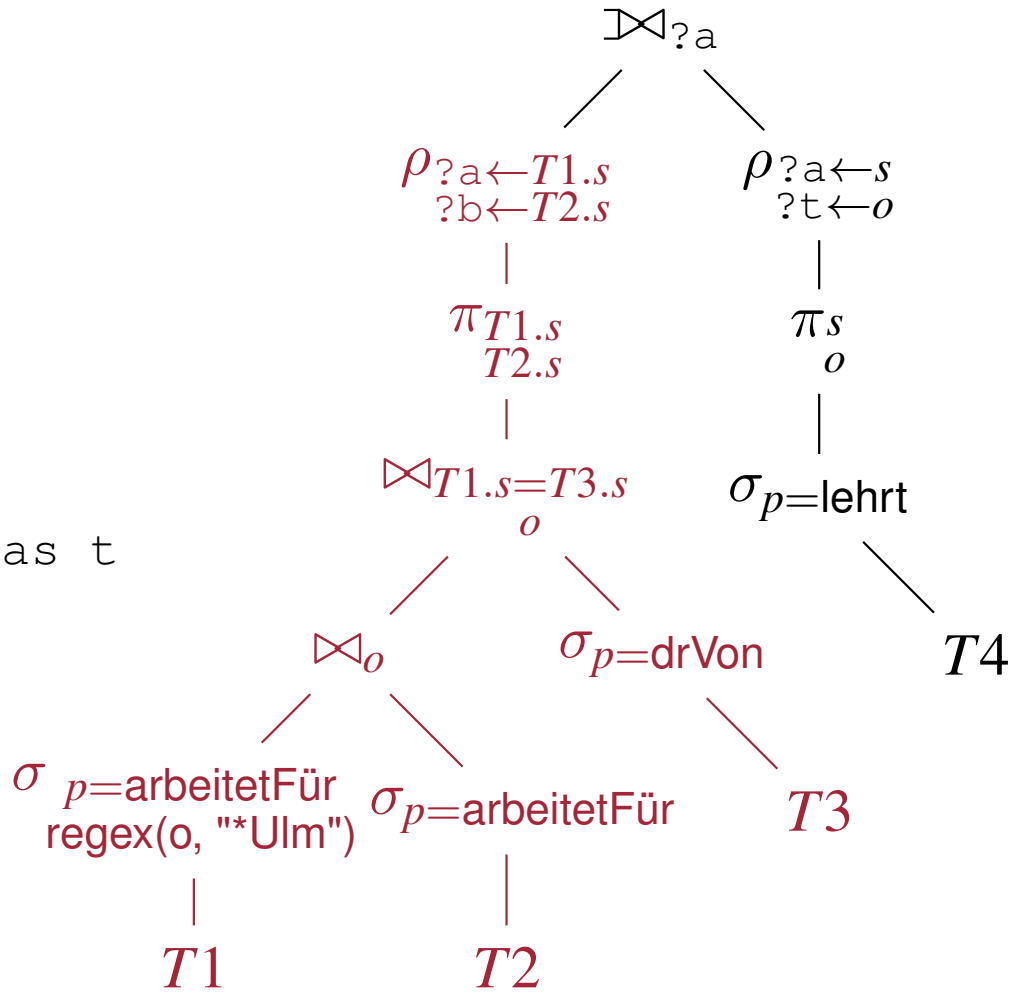
Beispiel: Umwandlung von SPARQL in SQL Abfragen

```
SELECT T4.s as a, T4.o as t
FROM Triples T4
WHERE T4.p="lehrt"
```



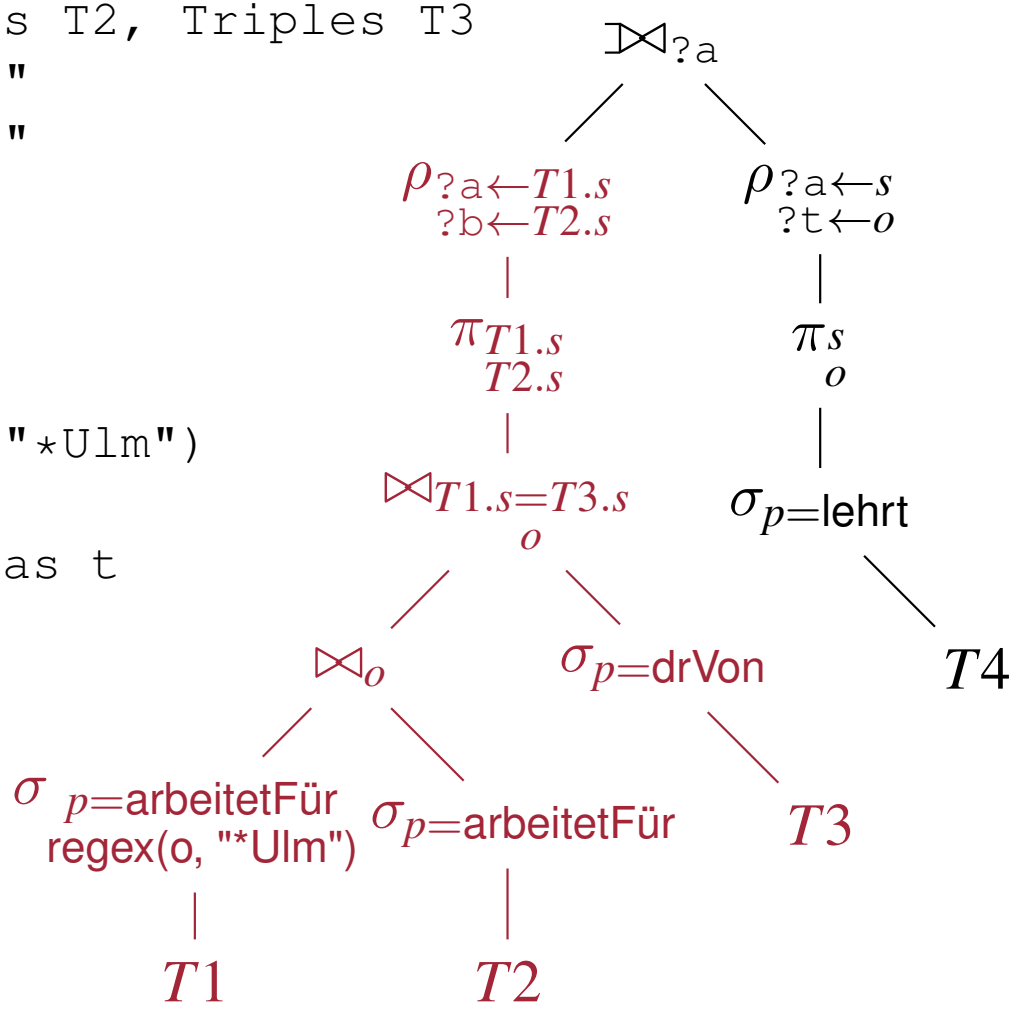
Beispiel: Umwandlung von SPARQL in SQL Abfragen

```
SELECT T4.s as a, T4.o as t
FROM Triples T4
WHERE T4.p="lehrt"
```



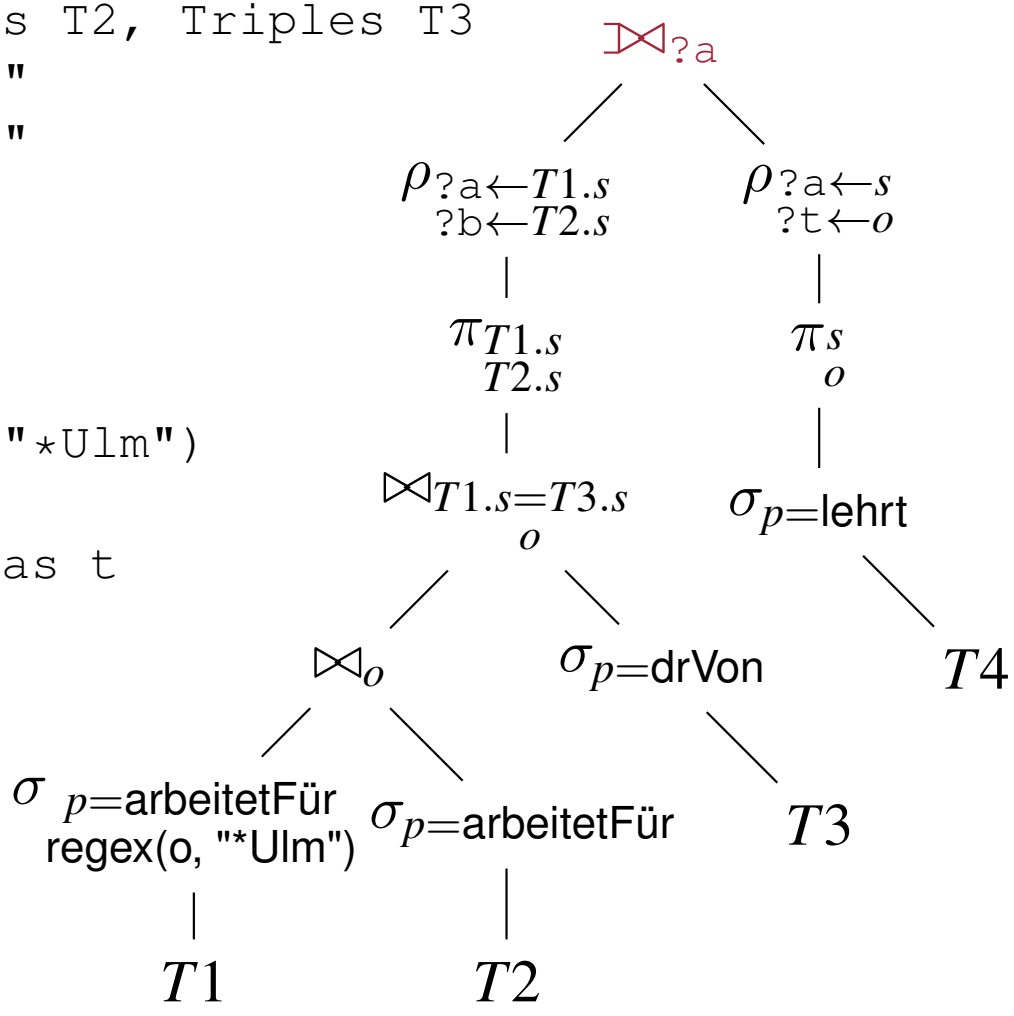
Beispiel: Umwandlung von SPARQL in SQL Abfragen

```
SELECT T1.s as a, T2.s as b
FROM Triples T1, Triples T2, Triples T3
WHERE T1.p="arbeitetFür"
  AND T2.p="arbeitetFür"
  AND T3.p="drVon"
  AND T1.o=T2.o
  AND T1.s=T3.s
  AND T1.o=T3.o
  AND REGEXP_LIKE(T1.o,"*Ulm")

SELECT T4.s as a, T4.o as t
FROM Triples T4
WHERE T4.p="lehrt"
```



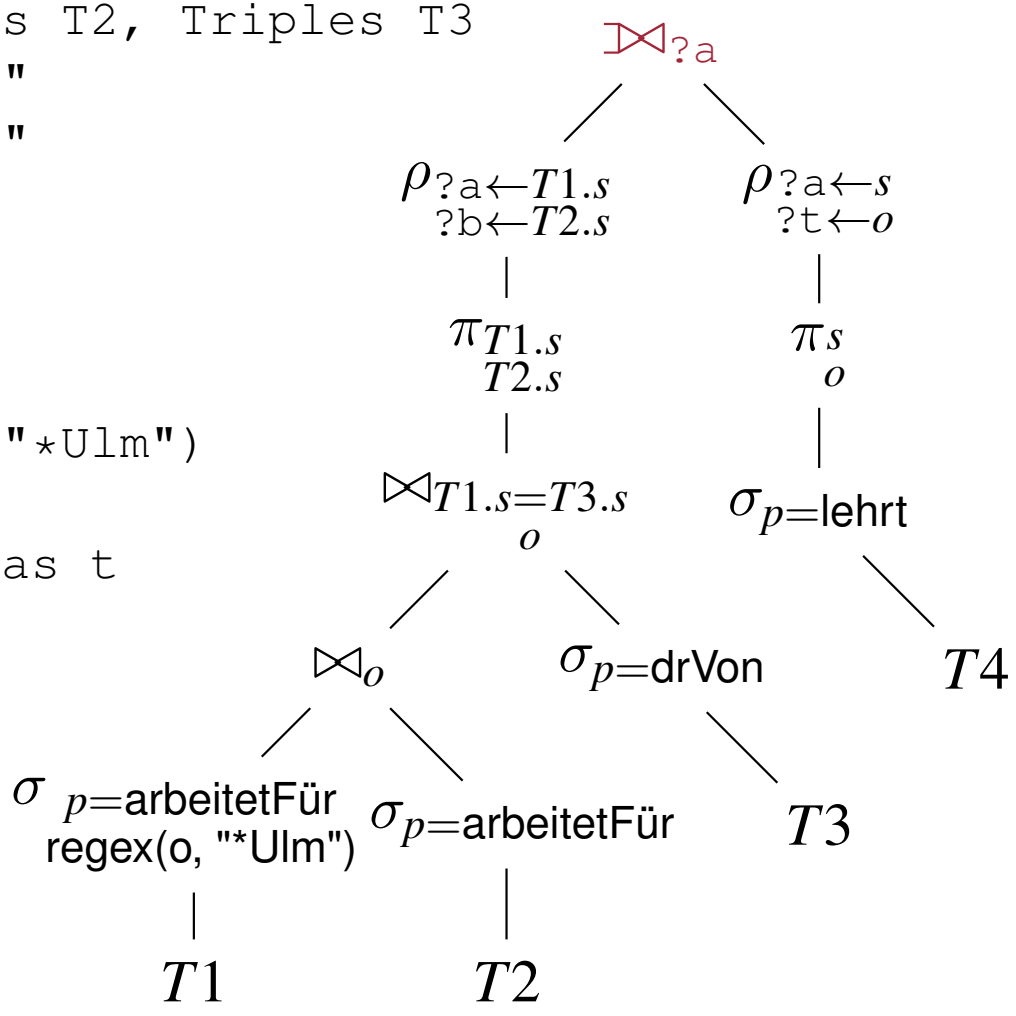
Beispiel: Umwandlung von SPARQL in SQL Abfragen

```
SELECT T1.s as a, T2.s as b
FROM Triples T1, Triples T2, Triples T3
WHERE T1.p="arbeitetFür"
  AND T2.p="arbeitetFür"
  AND T3.p="drVon"
  AND T1.o=T2.o
  AND T1.s=T3.s
  AND T1.o=T3.o
  AND REGEXP_LIKE(T1.o,"*Ulm")

SELECT T4.s as a, T4.o as t
FROM Triples T4
WHERE T4.p="lehrt"
```



Beispiel: Umwandlung von SPARQL in SQL Abfragen

```
SELECT * FROM
( SELECT T1.s as a, T2.s as b
  FROM Triples T1, Triples T2, Triples T3
  WHERE T1.p="arbeitetFür"
    AND T2.p="arbeitetFür"
    AND T3.p="drVon"
    AND T1.o=T2.o
    AND T1.s=T3.s
    AND T1.o=T3.o
    AND REGEXP_LIKE(T1.o,"*Ulm")
) AS R1 LEFT OUTER JOIN (
  SELECT T4.s as a, T4.o as t
  FROM Triples T4
  WHERE T4.p="lehrt"
) AS R2 ON (R1.a=R2.a)
```



Ist das Alles?

Ist das Alles?

Nein

Ist das Alles?

Nein

- Welche Indexe sollen wir zur Auswertung der Tripel Muster verwenden?
- Wie können wir die Speicheranforderungen reduzieren?
- Was ist der beste Ausführungsplan?

Ist das Alles?

Nein

- Welche Indexe sollen wir zur Auswertung der Tripel Muster verwenden?
- Wie können wir die Speicheranforderungen reduzieren?
- Was ist der beste Ausführungsplan?

Existierende Datenbanken nicht unbedingt geeignet:

- Zahlreiche self-joins auf einer großen Tabelle schlecht unterstützt
- Abfrageoptimierer generieren oft schlechte Pläne für eine große Tripel Tabelle
- Erweiterbares generisches Tabellenmanagement nicht nötig

Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

Dictionary für Strings

Strings werden auf eindeutige Integers abgebildet (z.B. durch Hashing)

- Gleichmäßige Größe $\rightsquigarrow$ einfacher zu handhaben
- Mapping ist meist klein $\rightsquigarrow$ kann im Hauptspeicher gehalten werden

Beispiel

`<http://example.org/Birte>`	194760
`<http://example.org/Sebastian>`	679375
`<http://example.org/Markus>`	4634

Dictionary für Strings

Strings werden auf eindeutige Integers abgebildet (z.B. durch Hashing)

- Gleichmäßige Größe $\rightsquigarrow$ einfacher zu handhaben
- Mapping ist meist klein $\rightsquigarrow$ kann im Hauptspeicher gehalten werden

Beispiel

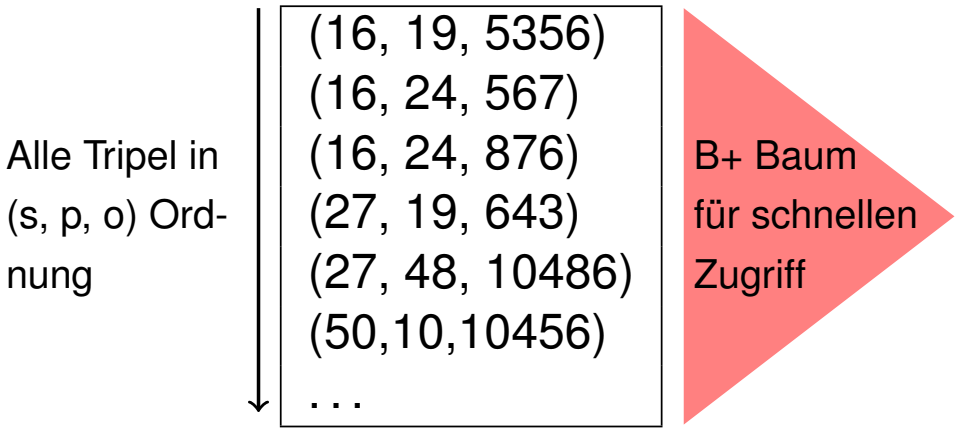
`<http://example.org/Birte>`	194760
`<http://example.org/Sebastian>`	679375
`<http://example.org/Markus>`	4634

Achtung: Natürliche Ordnung für das Sortieren geht nicht mehr $\rightsquigarrow$ Filterauswertung kann teurer werden

Indexe für häufige Tripel Muster

- Muster mit einer Variable sind häufig
- Bsp.: `ex:Einstein ex:invented ?x`

⇝ Anlegen eines clustered Index über (s, p, o)



1. ID Lookup:
   `ex:Einstein` =16,
   `ex:invented` =24
2. Lookup des Präfix im Index: (16,24,0)
3. Lese Ergebnisse solange Präfix matched:
   (16, 24, 567), (16, 24, 876)
   (sind sortiert!)

Geht auch für Muster der Form `ex:Einstein ?p ?x`

Indexe für häufige Tripel Muster

- Ähnliche Indexe für alle 6 Kombinationen: SPO, OPS, PSO, SOP, OSP, POS
- Deckt alle typischen Muster ab
- Ergibt sortierte Indexe für alle Muster mit zwei Variablen

⇝ Triple Tabelle nicht länger benötigt da alle Tripel in den Indexen

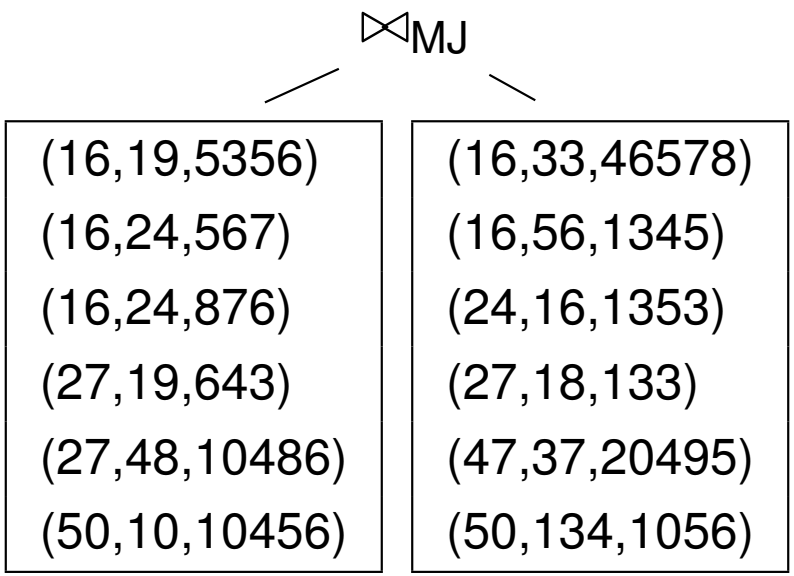
Relevanz der Sortierung für Joins

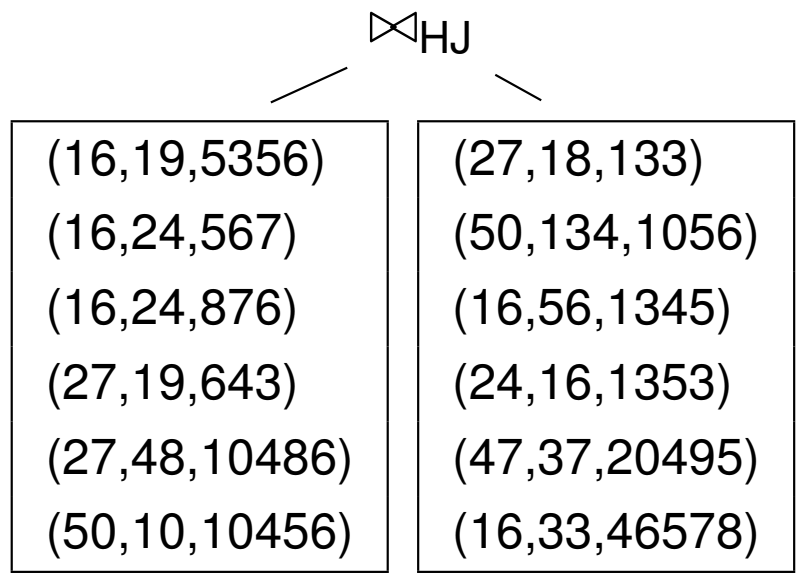
Eingaben auf dem Join Attribut sortiert $\rightsquigarrow$ Merge Join:

- Sequentieller Scan beider Tabellen
- Sofortiger Join der Tripel
- Teile ohne Matches überspringen
- Pipelining möglich

Eingaben auf dem Join Attribut unsortiert $\rightsquigarrow$ Hash Join:

- Hash Tabelle f. eine Tabelle
- Scanne die andere, nutze Hashing zum matchen
- Jeder Eingabe Tripel wird angefasst
- Kein Pipelining





Weitere Indexe

SPARQL erlaubt Duplikate (außer mit `DISTINCT`)

- Duplikate können für Anwendung wichtig sein (Zählen)

$\rightsquigarrow$ Häufig Abfragen mit vielen Duplikaten
Bsp.: `SELECT ?x WHERE {?x ?y ex:Deutschland}`
Um Elemente abzufragen, die zu Deutschland in Beziehung stehen

$\rightsquigarrow$ Viele identische Zwischenergebnisse

Lösung:

- Berechnung von aggregierten Indexen SP, SO, PO,PS, OP, OS, S, P, O
  Bsp.: SO enthält, für jedes Paar (s, o), die Anzahl der Tripel mit Subjekt s und Objekt o
- Statt identischer Bindungen speichern wir Multiplizitäten
  Bsp.: $\texttt{?x} \mapsto \texttt{ex:Einstein} : 4, \texttt{?x} \mapsto \texttt{ex:Merkel} : 10$

Kompression um Speicher zu Sparen

- Komprimiere Sequenzen von Tripeln in lexikographischer Ordnung
  (v_1, v_2, v_3); für SPO, $v_1 = S, v_2 = P, v_3 = O$
- Schritt 1: Berechne Deltas pro Attribut

(16, 19, 5356)	(16, 19, 5356)
(16, 24, 567)	(0, 5, -4798)
(16, 24, 676)	(0, 0, 109)
(27, 19, 643)	(11, -5, -34)
(27, 48, 10486)	(0, 29, 9843)
(50, 10, 10456)	(23, -38, -30)

- Schritt 2: Kodiere jedes Delta Tripel separat in 1-13 bytes

gap bit	header (7 bits)	Delta mit Wert 1 (0–4 bytes)	Delta mit Wert 2 (0–4 bytes)	Delta mit Wert 3 (0–4 bytes)

Wenn gap=1, Delta für v3 mit im Header sonst Wert 0

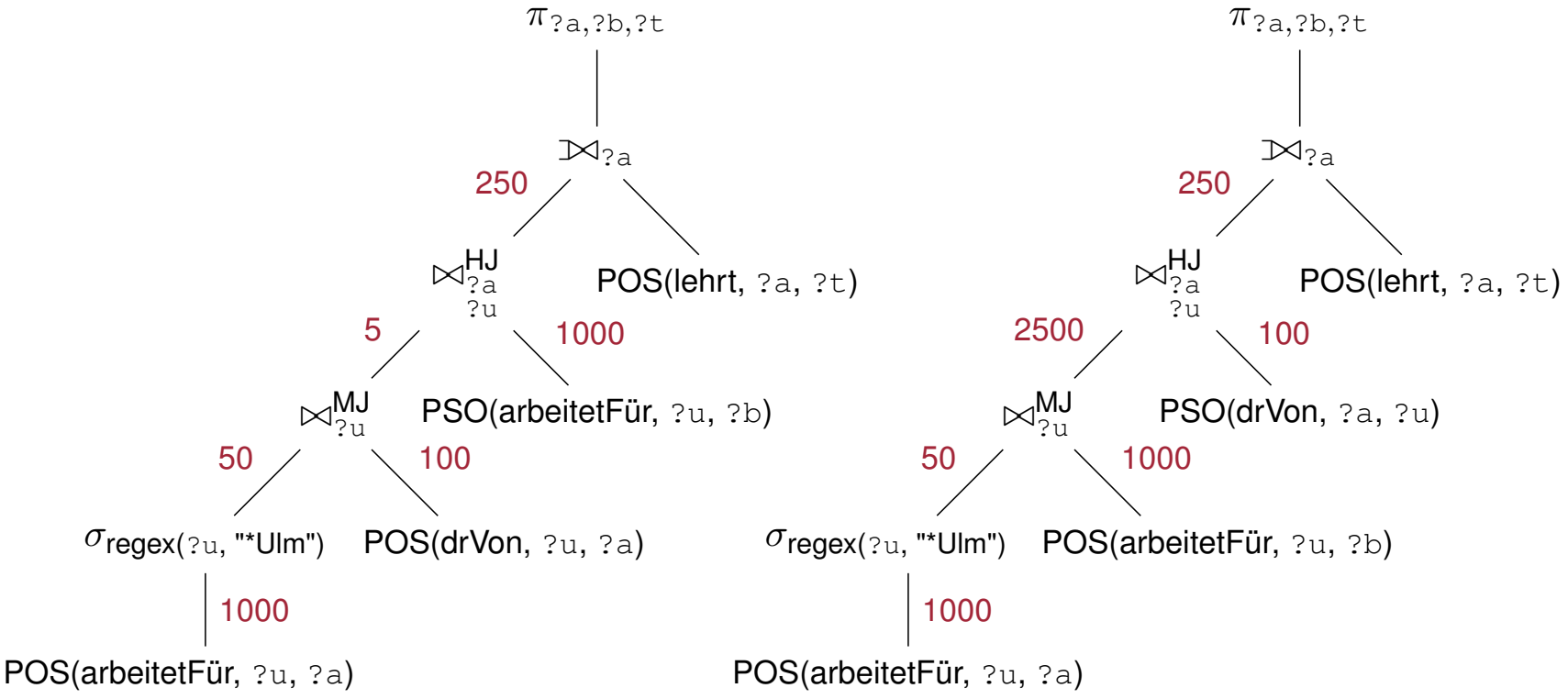
Sonst enthält der Header die Länge der Kodierungen für die drei Deltas (5*5*5=125 Kombinationen)

Effektivität und Effizienz der Kompression

- Byte-level Kodierung fast so effektiv wie bit-level Kodierungstechniken (Gamma, Delta, Golomb)
- Deutlich schneller (10×) bei der Dekodierung
- Beispiel Barton Datensatz (Neumann & Weikum 2010):
  - Rohdaten 51 Millionen Tripel, 7GB unkomprimiert
  - Alle 6 Hauptindexe:
    - 1.1 GB Größe, 3.2s Dekomprimierung mit Byte-level Kodierung
    - 1.06 GB Größe, 42.5s Dekomprimierung mit Delta Kodierung
- Zusätzliche Kompression mit LZ77 2× kompakter, aber deutlich langsamer bei der Dekomprimierung
- Kompression immer auf Seiten Level

Zurück zur Beispiel Abfrage

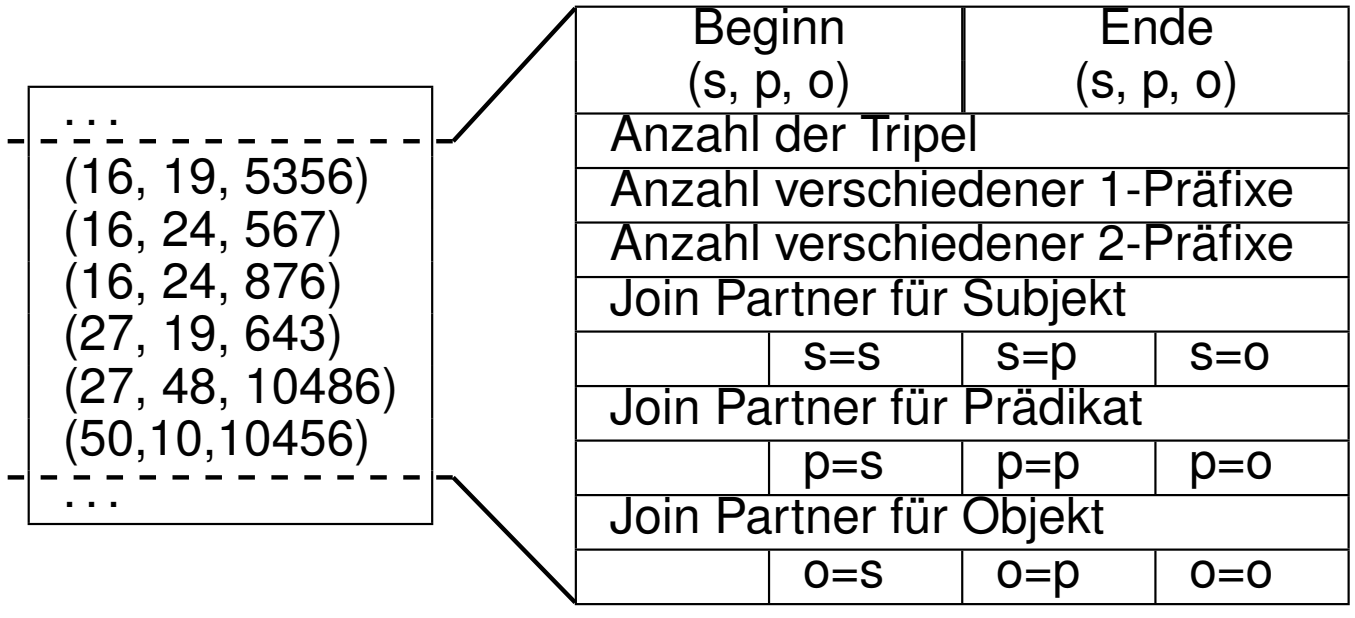
```
SELECT ?a ?b ?t WHERE {
  ?a arbeitetFür ?u . ?b arbeitetFür ?u . ?a drVon ?u .
  OPTIONAL { ?a lehrt ?t } FILTER (regex(?u, "*Ulm"))}
```



- Welcher der beiden Pläne ist besser?
- Wie viele Zwischenergebnisse?

Selektivitätsabschätzung für Abfragemuster

- Selektivitätsabschätzung sehr wichtig für gute Abfrageoptimierer (join order)
- In Standard Datenbanken:
  - Histogramm für die Attribute
  - Angenommene Unabängigkeit zwischen Attributen
  - ⇝ zu vereinfachend und ungenau
- Zusätzliche Join Statistiken für Tripel Blöcke:



Annahme: Unabhängigkeit zwischen Tripeln; zusätzlich Berechnung exakter Statistiken für häufige Pfade in den Daten

Verarbeitung von Updates

Wie können wir Änderungen an den Daten verarbeiten?
SPARQL 1.1 definiert Updates!

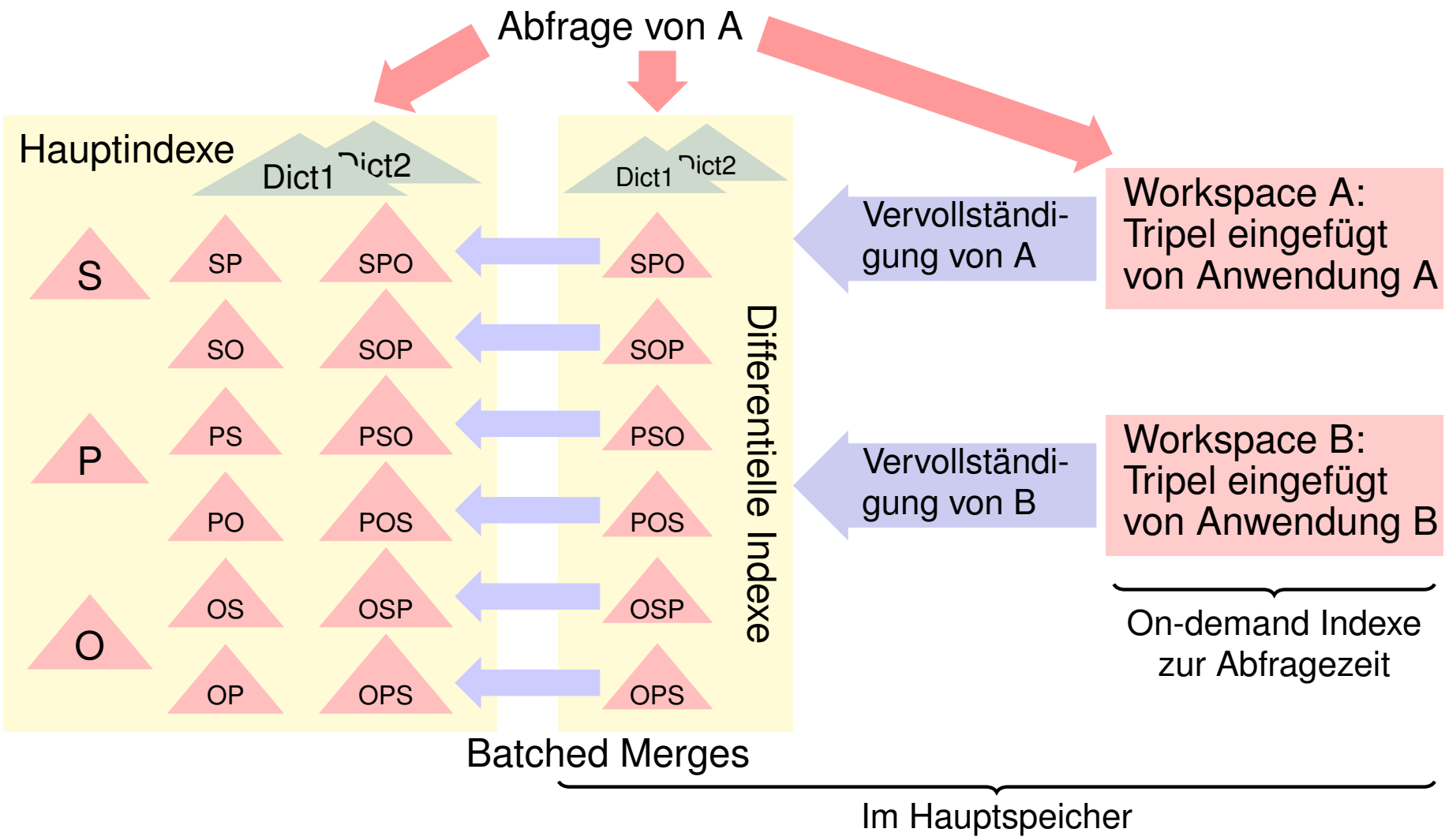
Annahmen:

- Lesende Abfragen wesentlich häufiger als Updates
- Updates meist Einfügen, weniger Löschen von Daten
- Verschiedene Anwendungen können nebenläufig updaten

Lösung:
Differenzielle Indexe

Differentielle Updates

Staging Architektur für Updates in RDF-3X



Löschen:

- Einfügen des gleichen Triples mit Löschflag
- Verändere scan/join Operatoren

Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

Prinzipien

Beobachtungen und Annahmen:

- Wenig verschiedene Prädikate
- Abfragemuster haben meist keine Variablen als Prädikate
- Bedarf zur Abfrage von Tripeln mit gleichem Prädikat

Design Konsequenzen:

- Nutzung einer Tabelle mit 2 Attributen pro Prädikat

Beispiel: Column Store

```
ex:Sebastian  ex:lehrt          ex:EinführungKI ;
              ex:arbeitetFür    ex:KIT ;
              ex:drVon          ex:TUDresden .
ex:Birte      ex:lehrt          ex:SemWebGrundlagen ;
              ex:arbeitetFür    ex:UUlm ;
              ex:drVon          ex:UoManchester .
ex:Markus     ex:lehrt          ex:KR&R ;
              ex:drVon          ex:KIT ;
              ex:arbeitetFür    ex:UoOxford ,
                                ex:SemanticWiki .
```

lehrt	
Subjekt	**Objekt**
ex:Sebastian	ex:EinführungKI
ex:Birte	ex:SemWebGrundlagen
ex:Markus	ex:KR&R

drVon	
Subjekt	**Objekt**
ex:Sebastian	ex:TUDresden
ex:Birte	ex:UoManchester
ex:Markus	ex:KIT

arbeitetFür	
Subjekt	**Objekt**
ex:Sebastian	ex:KIT
ex:Birte	ex:UUlm
ex:Markus	ex:UoOxford
ex:Markus	ex:SemanticWiki

Vereinfachtes Beispiel: Abfrage Konvertierung

```
SELECT ?a ?b ?t WHERE {
  ?a arbeitetFür ?u .
  ?b arbeitetFür ?u .
  ?a drVon ?u .
}
```

```
SELECT W1.Subjekt as a, W2.Subjekt as b
FROM arbeitetFür W1, arbeitetFür W2, drVon P3
WHERE W1.Objekt=W2.Objekt
  AND W1.Subjekt=P3.Subjekt
  AND W1.Objekt=P3.Objekt
```

Vereinfachtes Beispiel: Abfrage Konvertierung

```
SELECT ?a ?b ?t WHERE {
  ?a arbeitetFür ?u .
  ?b arbeitetFür ?u .
  ?a drVon ?u .
}
```

```
SELECT W1.Subjekt as a, W2.Subjekt as b
FROM arbeitetFür W1, arbeitetFür W2, drVon P3
WHERE W1.Objekt=W2.Objekt
  AND W1.Subjekt=P3.Subjekt
  AND W1.Objekt=P3.Objekt
```

Bisher ist das nur eine andere relationale RDF Repräsentation.
Aber was sind Column Stores?

Column Stores und RDF

Column Stores speichern Spalten einer Tabelle separat

drVon	
Subjekt	**Objekt**
ex:Sebastian	ex:TUDresden
ex:Birte	ex:UoManchester
ex:Markus	ex:KIT

drVon:Subjekt
ex:Sebastian
ex:Birte
ex:Markus

drVon:Objekt
ex:TUDresden
ex:UoManchester
ex:KIT

- Vorteile:
  - Schnell wenn nur auf das Subjekt oder nur das Objekt zugegriffen wird, nicht auf beides
  - Erlaubt sehr kompakte Repräsentation
- Probleme:
  - Subjekt und Objekt Spalten müssen wieder kombiniert werden bei Zugriff auf beides
  - Ineffizient für Muster mit Prädikatsvariablen

Kompression in Column Stores

Allgemeine Idee:

- Subjekt wird nur einmal gespeichert
- Nutzung der gleichen Ordnung nach Subjekt für alle Spalten, inkl. `NULL` Werte wenn nötig

Subjekt	**drVon**	**lehrt**	**arbeitetFür**
ex:Sebastian	ex:TUDresden	ex:GrundlagenKI	ex:KIT
ex:Birte	ex:UoManchester	ex:SemWebGrundlagen	ex:UUlm
ex:Markus	ex:KIT	ex:KR&R	ex:UoOxford
ex:Markus	NULL	NULL	ex:SemanticWiki

- Zusätzliche Kompression, um `NULL` Werte nicht zu speichern

drVon:bit[1110]
ex:TUDresden
ex:UoManchester
ex:KIT

lehrt:range[1-3]
ex:GrundlagenKI
ex:SemWebGrundlagen
ex:KR&R

Agenda

- Einleitung und Motivation
- Row Stores
  - Optimierungen
- Column Stores
- Andere Lösungen

Property Tabellen

Entitäten mit ähnlichen Prädikaten werden in einer relationalen Tabelle gespeichert (z.B. durch Nutzung der Typen oder eines Clustering Algorithmus)

```
ex:Sebastian  ex:lehrt         ex:GrundlagenKI ;
              ex:arbeitetFür   ex:KIT ;
              ex:drVon         ex:TUDresden .
ex:Birte      ex:lehrt         ex:SemWebGrundlagen ;
              ex:arbeitetFür   ex:UUlm ;
              ex:drVon         ex:UoManchester .
ex:Markus     ex:lehrt         ex:KR&R ;
              ex:drVon         ex:KIT ;
              ex:arbeitetFür   ex:UoOxford , ex:SemanticWiki .
```

Subjekt	lehrt	drVon
ex:Sebastian	ex:GrundlagenKI	ex:TUDresden
ex:Birte	ex:SemWebGrundlagen	ex:UoManchester
ex:Markus	ex:KR&R	ex:KIT

übrige Tripel

Subjekt	Prädikat	Objekt
ex:Sebastian	ex:arbeitetFür	ex:KIT
ex:Birte	ex:arbeitetFür	ex:UUlm
ex:Markus	ex:arbeitetFür	ex:UoOxford
ex:Markus	ex:arbeitetFür	ex:SemanticWiki

Property Tabellen: Pro und Kontra

Vorteile:

- Ähnlicher zu bestehenden relationalen DBMS
- Weniger self-joins als mit Tripel Tabellen

Nachteile:

- Potentiell viele `NULL` Werte
- Mehrwertige Attribute problematisch
- Abfrage Transformation hängt vom Schema ab
- Änderungen am Schema teuer

Noch viele weitere Techniken

- Speicherung von RDF Daten als bit-Vektor Matritzen
- Konvertierung von RDF in XML und Nutzung von XML Methoden (XPath, XQuery)
- Speicherung von RDF Daten in Graph-Datenbanken
- ...

Welche Technik ist die Beste?

- Performanz hängt stark von der Vorberechung, Optimierung und Implementierung ab

Vergleichende Studie

	Q1	Q2	Q3	Q4
cold caches				
RDF-3X	0.27+0.01	0.26+0.01	0.94+0.01	5.32+0.06
RDF-3X (2008)	4.02+0.03	0.36+0.01	4.5+0.04	112.0+3.32
COLSTORE	>30min	>30min	946.4+8.01	>30min
ROWSTORE	17.44 +0.29	>30min	641.2+0.88	373.0+2.15
warm caches				
RDF-3X	0.01+0.01	0.01+0.01	0.01+0.01	0.23+0.01
RDF-X (2008)	0.54+0.01	0.01+0.01	0.28+0.01	105.7+1.71
COLSTORE	>30min	>30min	547.9+83.0	>30min
ROWSTORE	0.69+0.03	>30min	637.1+4.97	212.0+1.43

	Q5	Q6	Q7	Q8	geom. Mittel
cold caches					
RDF-3X	2.37+0.04	0.46+0.01	3.99+0.05	0.35+0.01	0.94+0.01
RDF-3X (2008)	6.82+0.63	1.28+0.06	>30min	44.23+0.67	>7.43+0.07
COLSTORE	>30min	665.9+38.4	18.71+0.32	>30min	>208.0+1.49
ROWSTORE	>30min	6.89+0.07	>30min	142.3+13.4	>87.22+0.68
warm caches					
RDF-3X	0.08+0.01	0.03+0.01	0.03+0.01	0.01+0.01	0.04+0.01
RDF-X (2008)	3.72+0.02	0.73+0.01	>30min	15.12+0.20	>2.59+0.06
COLSTORE	>30min	650.0+31.7	0.86+0.02	>30min	>150.0+1.48
ROWSTORE	>30min	0.81+0.04	>30min	60.29+0.87	>43.98+0.22

Laufzeiten in Sekunden auf dem BTC 2008 Datensatz ([Neumann & Weikum, 2009])

Zusammenfassung

- Diverse Implementierungstechniken für Tripel Stores
- Eng verwandt mit Datenbanken
- Transaktionsmanagement oft (noch) vernachlässigt
- Oft auch Erweiterung um Regelsprachen
- Optimierungen essentiell für vernüftige Performanz