Semantic data systems

Data storage and processing
(no reasoning)

RDF DATA STORAGE

RDF data store

RDF as an interface format

RDF Enabled Systems

RDF is not the logical model

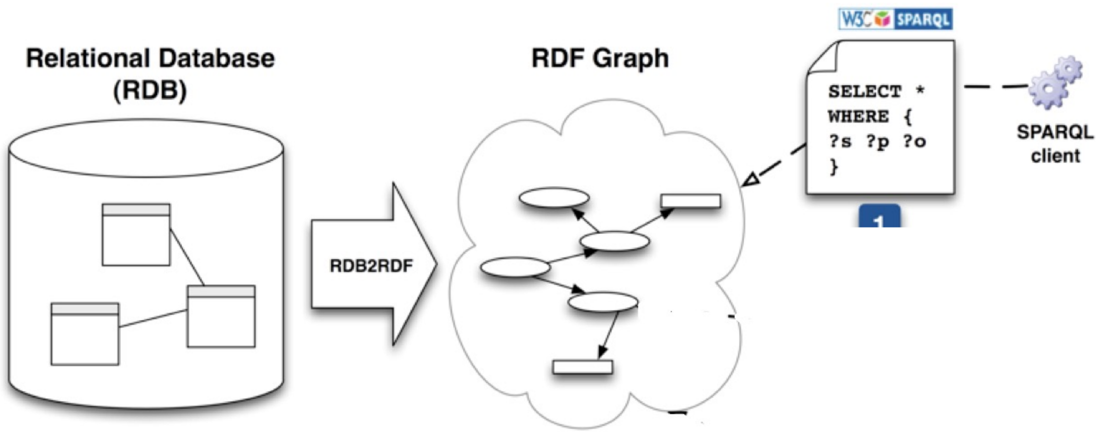
RDF as a storage format

RDF Native Systems

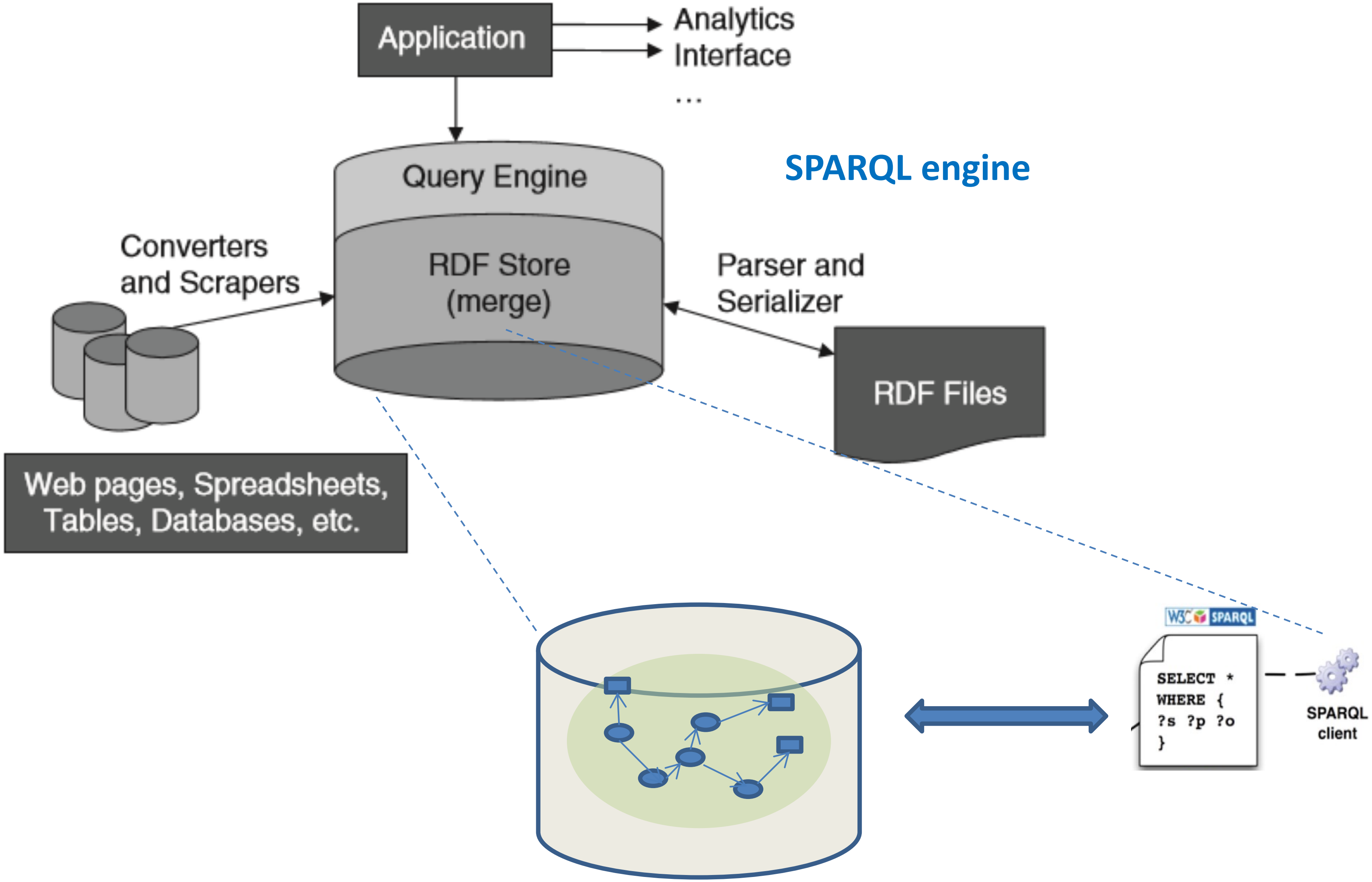
RDF (triples, graphs) as logical data model

RDF enabled systems

- Implemented as a middleware on top of RDBMS
- Relational data contained in traditional relational databases are translated into RDF data (and viceversa) and stored into an RDBMS
- SPARQL queries must be translated into SQL
- RDF is just a view for relational data, the system is not aware of RDF
- **Pros**: you can rely on DBMS functionalities
- **Cons**: external and internal model do not coincide, the DBMS is not optimized for RDF storage and SPARQL access, slower



RDF Native Systems



RDF Native Systems

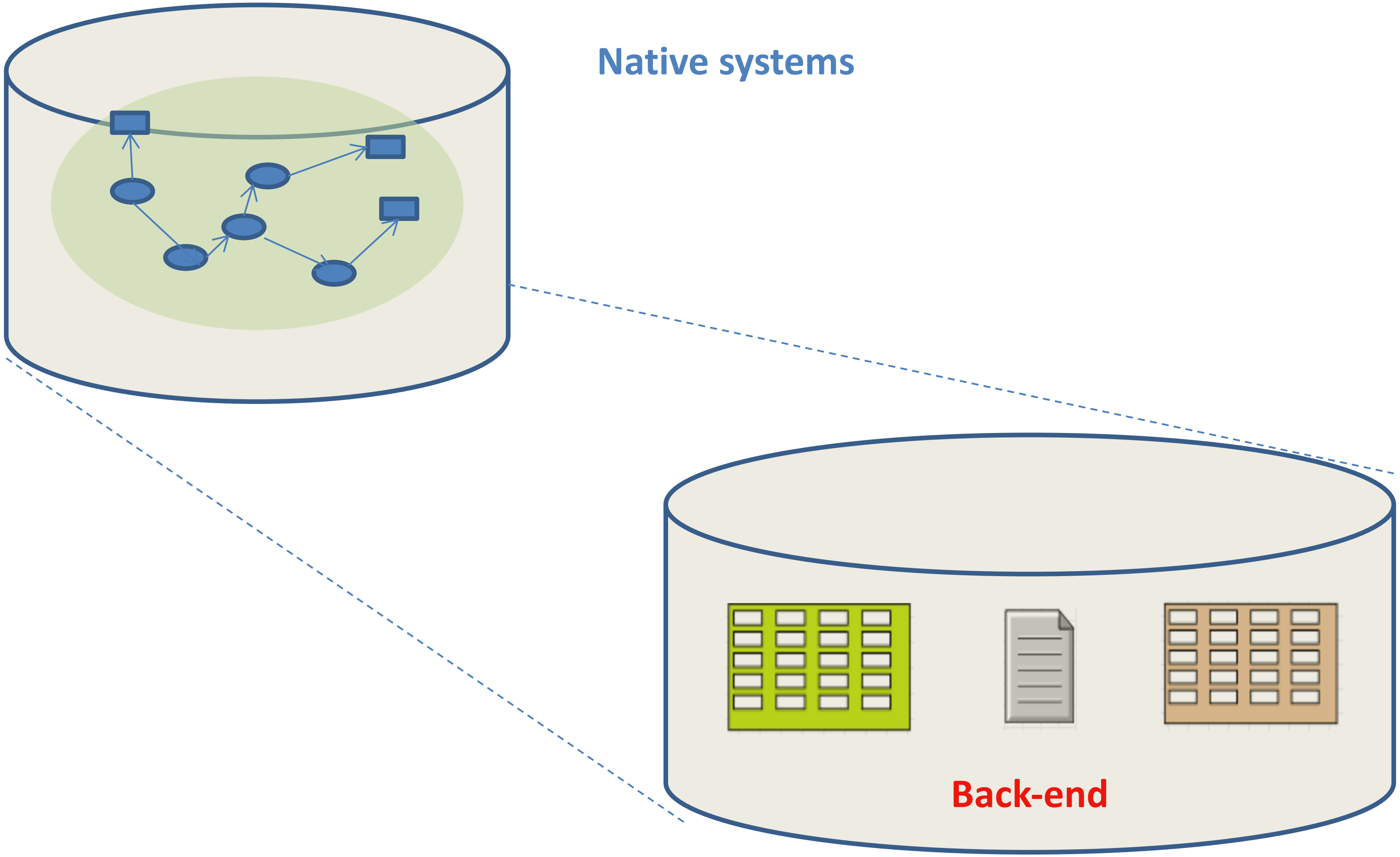
- New systems, tailored to RDF data management → TripleStore
- RDF (triples, graph) is the logical model
- SPARQL engine available
- Triples stored in memory or in a persistent back-end
- Persistence can be provided by
  - a relational DBMS or
  - a NoSQL graph-based system or
  - other systems (e.g., XML based)
- The backend system often supports large-scale functionalities
- **Pros**: faster, the system is optimized for RDF storage and SPARQL access
- **Cons**: less consolidated technologies with respect to RDBMS

- https://www.w3.org/2001/sw/wiki/Tools for a list of existing tools

RDF Native Systems with Relational backend

- Backend: RDBMS
- Basic principles:
  - store triples in table (but the system is aware that some tables represent triples), many alternatives available
  - convert SPARQL to equivalent
  - the database will do the rest
- Used by many TripleStores

RDF Native Systems with Relational backend



RDF Native Systems with Relational backend: Single Triple Table

- Store triples in one single giant three-attribute table (subject, predicate, object)

Example: Single Triple Table

```
ex:Katja    ex:teaches    ex:Databases;
            ex:works_for  ex:MPI_Informatics;
            ex:PhD_from   ex:TU_Ilmenau.
ex:Martin   ex:teaches    ex:Databases;
            ex:works_for  ex:MPI_Informatics;
            ex:PhD_from   ex:Saarland_University.
ex:Ralf     ex:teaches    ex:Information_Retrieval;
            ex:PhD_from   ex:Saarland_University;
            ex:works_for  ex:Saarland_University,
                          ex:MPI_Informatics.
```

Table **Triples**

subject	predicate	object
ex:Katja	ex:teaches	ex:Databases
ex:Katja	ex:works_for	ex:MPI_Informatics
ex:Katja	ex:PhD_from	ex:TU_Ilmenau
ex:Martin	ex:teaches	ex:Databases
ex:Martin	ex:works_for	ex:MPI_Informatics
ex:Martin	ex:PhD_from	ex:Saarland_University
ex:Ralf	ex:teaches	ex:Information_Retrieval
ex:Ralf	ex:PhD_from	ex:Saarland_University
ex:Ralf	ex:works_for	ex:Saarland_University
ex:Ralf	ex:works_for	ex:MPI_Informatics

Conversion of SPARQL to SQL

General approach to translate SPARQL into SQL:

(1) Each **triple pattern** is translated into a (self-) JOIN over the triple table

(2) **Shared variables** create **JOIN conditions**

(3) **Constants** create **WHERE conditions**

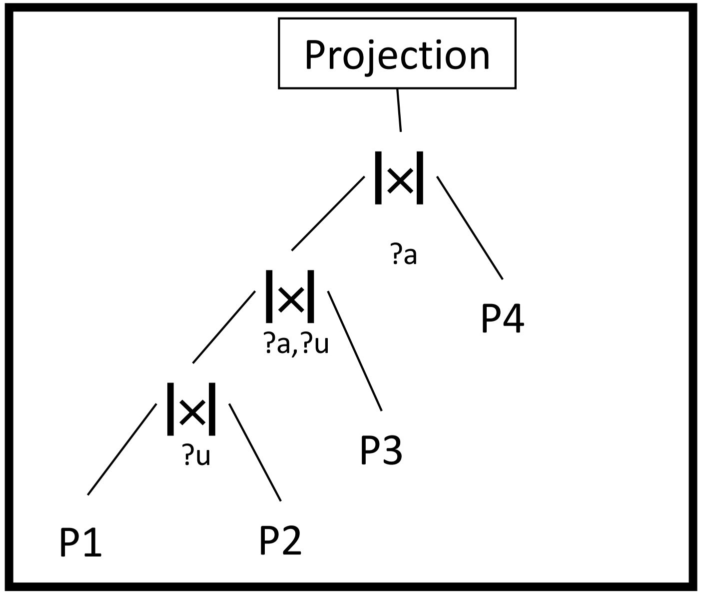
(4) **FILTER conditions** create **WHERE conditions**

(5) **OPTIONAL clauses** create **OUTER JOINS**

(6) **UNION clauses** create **UNION expressions**

Example: from SPARQL to SQL

SELECT ?a ? b WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }

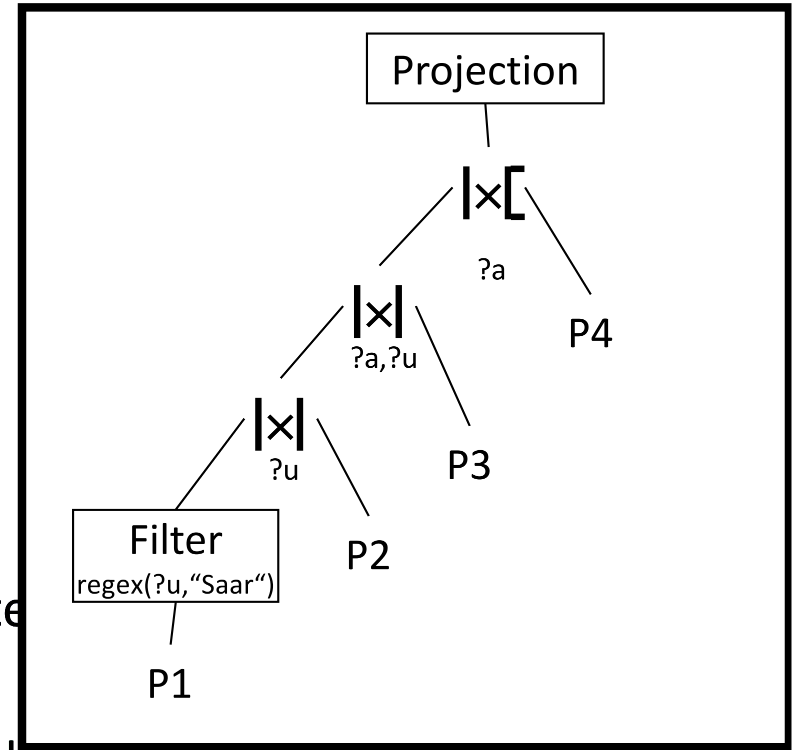


SELECT P1.subject as A, P2.subject as B
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate=“works_for” AND P2.predicate=“works_for”
AND P3.predicate=“phd_from”
AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object

Many self and outer joins

Example: from SPARQL to SQL

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))
```

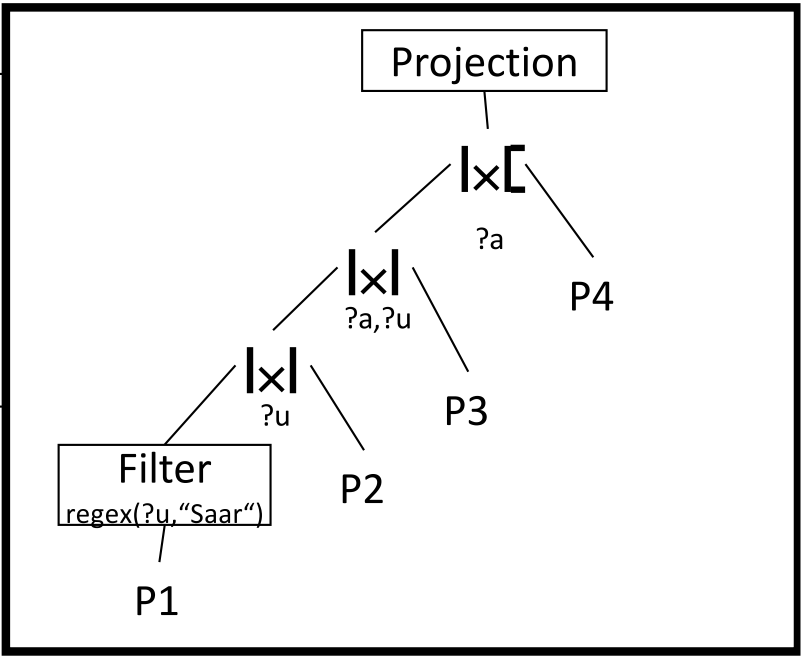



```
SELECT P1.subject as A, P2.subject as B
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate=“works_for” AND P2.predicate
    AND  P3.predicate=“phd_from”
    AND  P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
    AND REGEXP_LIKE(P1.object,  “Saar”)
```

Many self and outer joins

Example: from SPARQL to SQL

SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, “Saar”))




```
SELECT R1.A, R1.B, R2.T FROM
( SELECT P1.subject as A, P2.subject as B
  FROM Triples P1, Triples P2, Triples P3
  WHERE P1.predicate=“works_for” AND P2.predicate=“works_for”
     AND  P3.predicate=“phd_from”
     AND  P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
     AND REGEXP_LIKE(P1.object, “Saar”)
 ) R1 LEFT OUTER JOIN
 (   SELECT P4.subject as A, P4.object as T
     FROM Triples P4
     WHERE P4.predicate=“teaches”) AS R2
) ON (R1.A=R2.A)
```

Many self and outer joins

Single Triple Table: Pros and Cons

Advantages:

- No restructuring is required if the ontology changes (e.g., new classes, etc., realized by a simple INSERT command in the table)

Disadvantages:

- Performing a query means searching the whole database and queries involving joins become very expensive

Dictionary for Strings

Map all strings to unique integers (e.g., via hashing)

- Regular size (4-8 bytes), much easier to handle
- Dictionary usually small, can be kept in main memory

```
<http://example.de/Katja>    → 194760
<http://example.de/Martin>   → 679375
<http://example.de/Ralf>     →   4634
```

This may break original lexicographic sorting order

⇒ RANGE conditions are difficult!

⇒ FILTER conditions may be more expensive!

RDF Native Systems with Relational backend: Property Tables

Observations and assumptions

- Not too many different predicates
- Triple patterns usually have fixed predicate
- Need to access all triples with one predicate

Design consequence

- Use one two-attribute table for each predicate
- From one giant three-attribute table to many property tables

Example

```
ex:Katja     ex:teaches     ex:Databases;
             ex:works_for ex:MPI_Informatics;
             ex:PhD_from ex:TU_Ilmenau.
ex:Martin    ex:teaches     ex:Databases;
             ex:works_for ex:MPI_Informatics;
             ex:PhD_from ex:Saarland_University.
ex:Ralf      ex:teaches     ex:Information_Retrieval;
             ex:PhD_from ex:Saarland_University;
             ex:works_for ex:Saarland_University,
                            ex:MPI_Informatics.
```

works_for

subject	object
ex:Katja	ex:MPI_Informatics
ex:Martin	ex:MPI_Informtatics
ex:Ralf	ex:Saarland_University
ex:Ralf	ex:MPI_Informatics

teaches

subject	object
ex:Katja	ex:Databases
ex:Martin	ex:Databases
ex:Ralf	ex:Information_Retrieval

PhD_from

subject	object
ex:Katja	ex:TU_Ilmenau
ex:Martin	ex:Saarland_University
ex:Ralf	ex:Saarland_University

Example: from SPARQL to SQL

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
```

```
SELECT W1.subject as A, W2.subject as B
FROM works_for W1, works_for W2, phd_from P3
WHERE W1.object=W2.object
    AND W1.subject=P3.subject
    AND W1.object=P3.object
```

Fragmentation-based solutions and columnstores

Columnstores store *each* column (or group of columns) of a table separately

PhD_from	
subject	**object**
ex:Katja	ex:TU_Ilmenau
ex:Martin	ex:Saarland_University
ex:Ralf	ex:Saarland_University

PhD_from:subject
ex:Katja
ex:Martin
ex:Ralf

PhD_from:object
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University

Advantages:
- **Fast if only subject or object are accessed, not both**
- **Allows for a very compact representation**

Problems:
- **Need to recombine columns if subject and object are accessed**
- **Inefficient for triple patterns with predicate variable**
- **Space overhead in case the same subject is replicated among triples several times**

Compression in Columnstores

General ideas:

- Store subject the minimum number of times
- Use same order of subjects for all columns, including NULL values when necessary

subject
ex:Katja
ex:Martin
ex:Ralf
ex:Ralf

PhD_from
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University
NULL

teaches
ex:Databases
ex:Databases
ex:Information_Retrieval
NULL

works_for
ex:MPI_Informatics
ex:MPI_Informatics
ex:Saarland_University
ex:MPI_Informatics

- Additional compression to get rid of NULL values

PhD_from: bit[1110]
ex:TU_Ilmenau
ex:Saarland_University
ex:Saarland_University

Teaches: range[1-3]
ex:Databases
ex:Databases
ex:Information_Retrieval

Other Solutions: Property Tables

Group entities with similar predicates into a relational table (for example using RDF types or a clustering algorithm).

```
ex:Katja     ex:teaches     ex:Databases;
             ex:works_for ex:MPI_Informatics;
             ex:PhD_from ex:TU_Ilmenau.
ex:Martin    ex:teaches     ex:Databases;
             ex:works_for ex:MPI_Informatics;
             ex:PhD_from ex:Saarland_University.
ex:Ralf      ex:teaches     ex:Information_Retrieval;
             ex:PhD_from ex:Saarland_University;
             ex:works_for ex:Saarland_University,
                            ex:MPI_Informatics.
```

subject	teaches	PhD_from
ex:Katja	ex:Databases	ex:TU_Ilmenau
ex:Martin	ex:Databases	ex:Saarland_University
ex:Ralf	ex:IR	ex:Saarland_University
[illegible]	NULL	ex:TU_Vienna

subject	predicate	object
ex:Katja	ex:works_for	ex:MPI_Informatics
ex:Martin	ex:works_for	ex:MPI_Informatics
ex:Ralf	ex:works_for	ex:Saarland_University
ex:Ralf	ex:works_for	ex:MPI_Informatics

“Leftover triples”

Property Tables: Pros and Cons

Advantages:

- More in the spirit of existing relational systems
- Saves many self-joins over triple tables

Disadvantages:

- Query mapping depends on schema
- Schema changes very expensive

RDF Native Systems with Relational backend: is that all?

Well, no.

- Which indexes should be built?
  (to support efficient evaluation of triple patterns)
- How can we reduce storage space?
- How can we find the best execution plan?

Traditional RDBMS has to take into account RDF features:

- flexible, extensible, generic storage not needed here
- cannot deal with multiple self-joins of a single table
- often generate bad execution plans

RDF Native Systems with graph-based backend

- Backend: NoSQL graph-based system
- Implementing triple stores with a relational backend makes the system associative
  - Triples stored in different tables can be combined together only through joins
- On the other hand, NoSQL graph-based systems are navigational and provide native graph storage
  - Connections between nodes are stores and can be directly navigated through pointers
- NoSQL graph-based systems more suitable for deep or variable-length traversals and path queries
  - Lead to a very large number of joins in TripleStore with relational backend
- NoSQL graph databases should support specialized graph index structures , tailored to RDF

RDF USE CASES

RDF use cases

- Two possible scenarios
  - Local access
    - Store your RDF dataset in a given Triplestore
    - use the interaction protocol, based on SPARQL, available in the TripleStore
  - Remote access (more interesting)
    - Single RDF dataset available on the Web
    - Processing a single RDF dataset or a federation of many RDF datasets

Processing a single RDF dataset

- SPROT = SPARQL Protocol for RDF
- SPARQL endpoint
  - A service, conformant to SPROT, that accepts SPARQL queries and returns results via HTTP, in one or more machine-processable formats
  - Either generic (fetching data on the Web as needed) or specific (querying an associated TripleStore)
  - Issuing a SPARQL query is an HTTP GET request with parameter query
- A SPARQL endpoint is mostly conceived as a machine-friendly interface towards a knowledge base
- https://www.w3.org/wiki/SparqlEndpoints for a list of available SPARQL endpoints
- https://dbpedia.org/sparql: SPARQL endpoint for DBpedia (the semantic version of Wikipedia, https://www.dbpedia.org/)

Processing a single RDF dataset: SPARQL Client Libraries

- **More convenient than on the protocol level:**
  - SPARQL JavaScript Library
    http://www.thefigtrees.net/lee/blog/2006/04/sparql_calendar_demo_a_sparql.html
  - ARC for PHP http://arc.semsol.org/
  - RAP – RDF API for PHP
    http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/index.html
  - Jena / ARQ (Java) http://jena.sourceforge.net/
  - Sesame (Java) http://www.openrdf.org/
  - SPARQL Wrapper (Python)
    http://sparql-wrapper.sourceforge.net/
  - PySPARQL (Python)
    http://code.google.com/p/pysparql/

Processing a federation of many RDF datasets

- Many RDF datasets, stored in many independent repositories
- For some applications, you might need to use all the datasets in the context of the same query
- Two approaches
  - Controlled integration approach
  - Link traversal-based query execution

Controlled integration approach

- Assumptions
  - you know in advance the RDF data sources to be queried
  - each data source is exposed via a SPARQL endpoint
- Specify the service (i.e., the SPARQL endpoint) you want to use in the context of your query

Controlled integration approach: SPARQL 1.1 Federation Extension

- **SERVICE pattern in SPARQL 1.1**
  - Explicitly specify query patterns whose execution must be distributed to a remote SPARQL endpoint

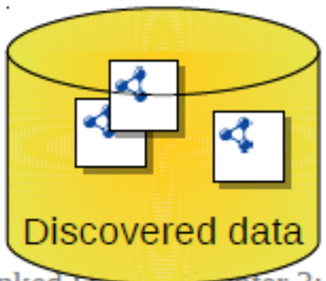
```
SELECT ?v ?ve WHERE
{
  ?v rdf:type umbel-sc:Volcano ;
     p:location dbpedia:Italy .
  SERVICE <http://volcanos.example.org/query> {
                                ?v p:lastEruption ?ve }
}
```

Controlled Integration Solutions

- Pros:
  - Queried data is up to date: you do not care about their storage and update
- Cons:
  - All relevant datasets must be exposed via a SPARQL endpoint
  - You have to know the relevant data sources beforehand
  - You restrict yourselves to the selected sources
  - You do not tap the full potential of the Web

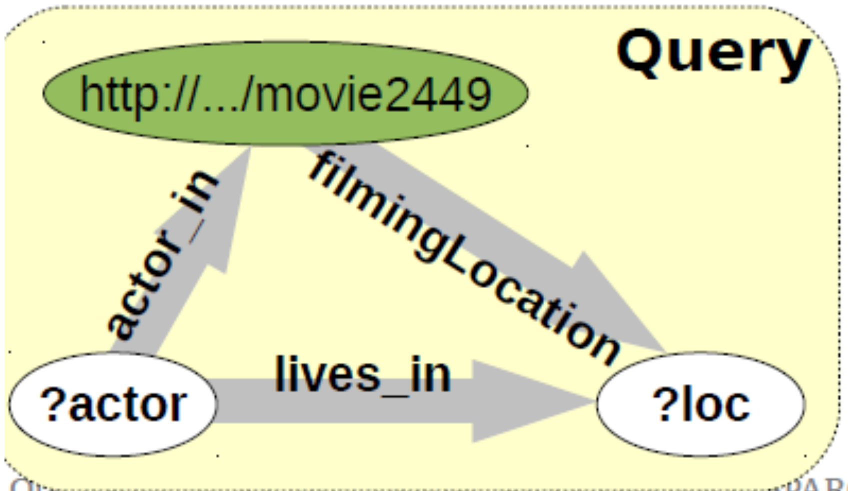
Link traversal-based query execution

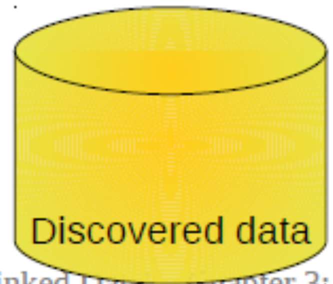
- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



Link traversal-based query execution

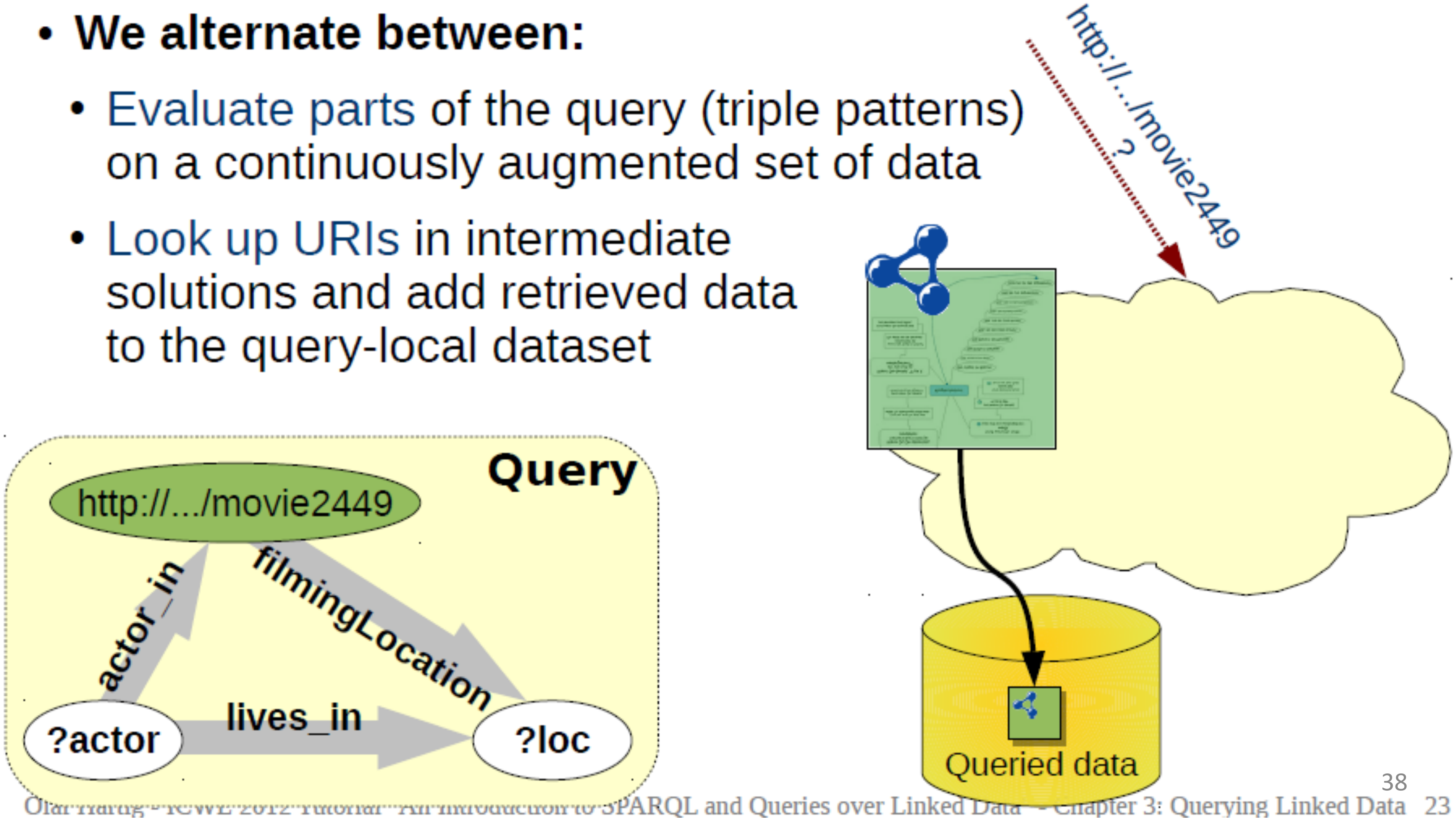
- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset





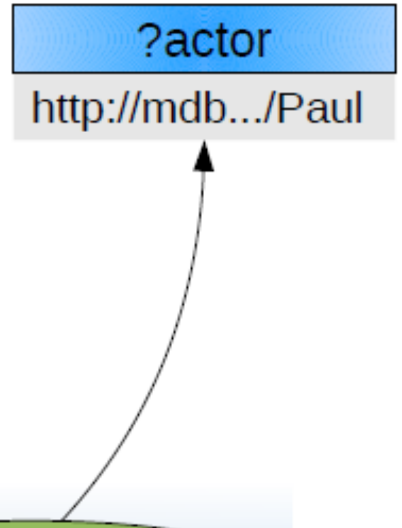
Link traversal-based query execution

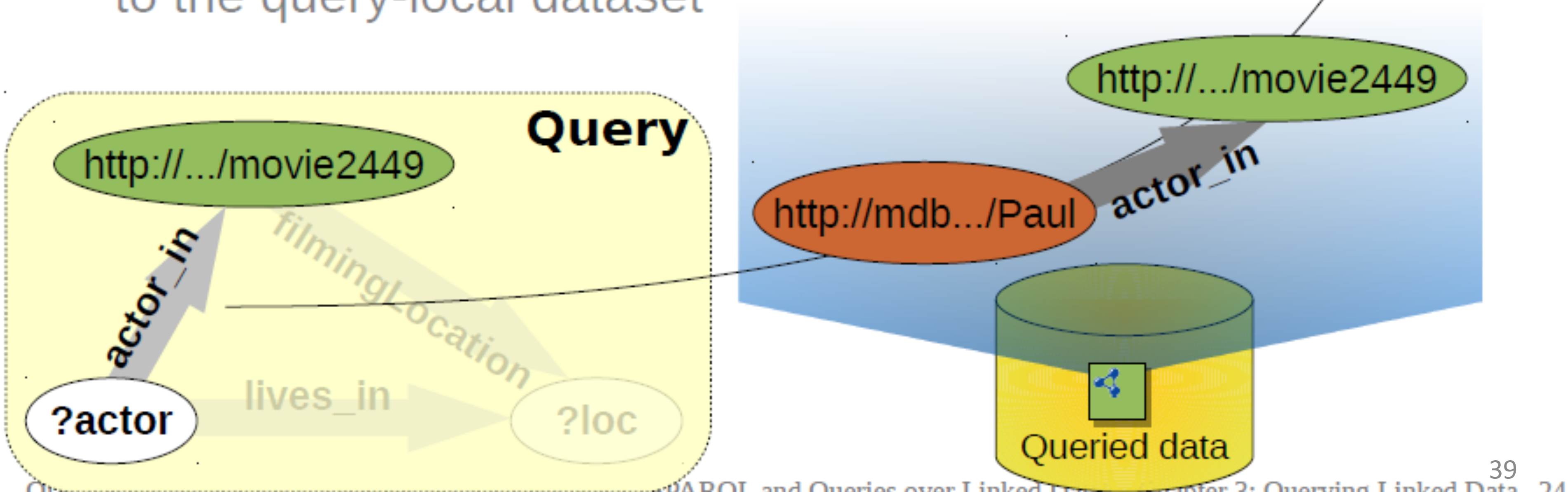
- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



Link traversal-based query execution

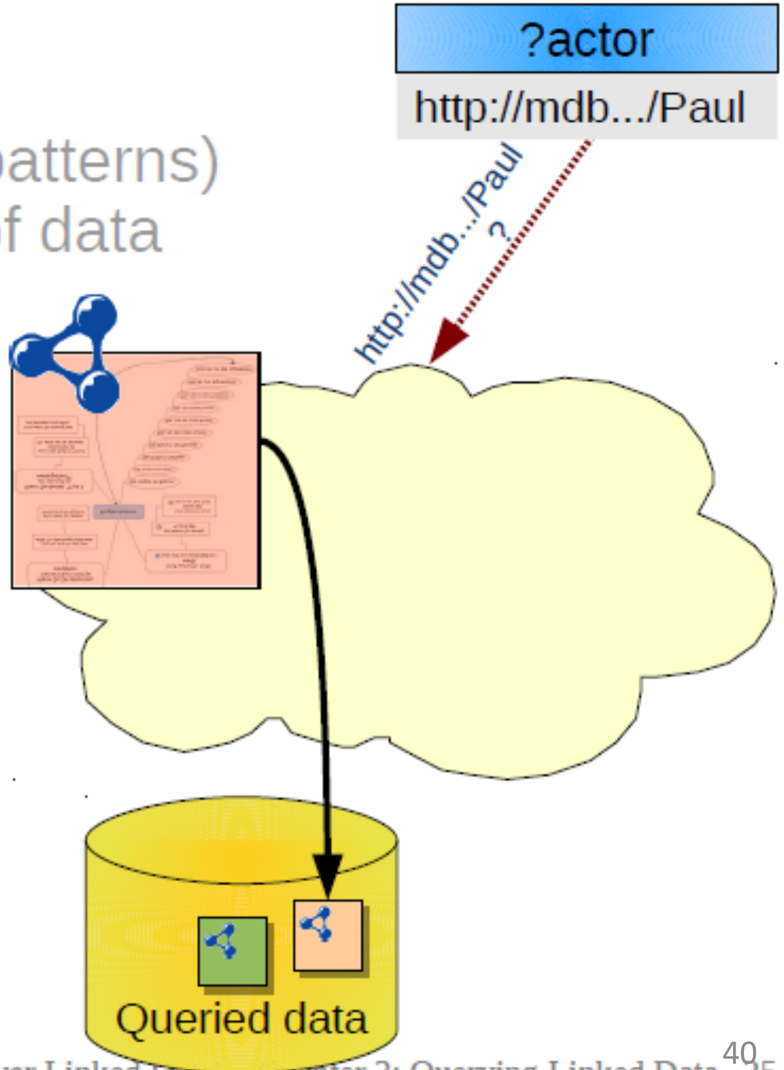
- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset

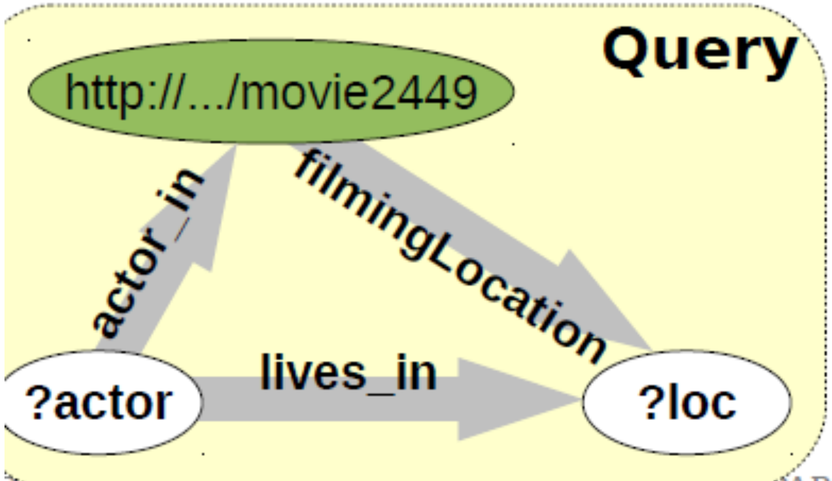




Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



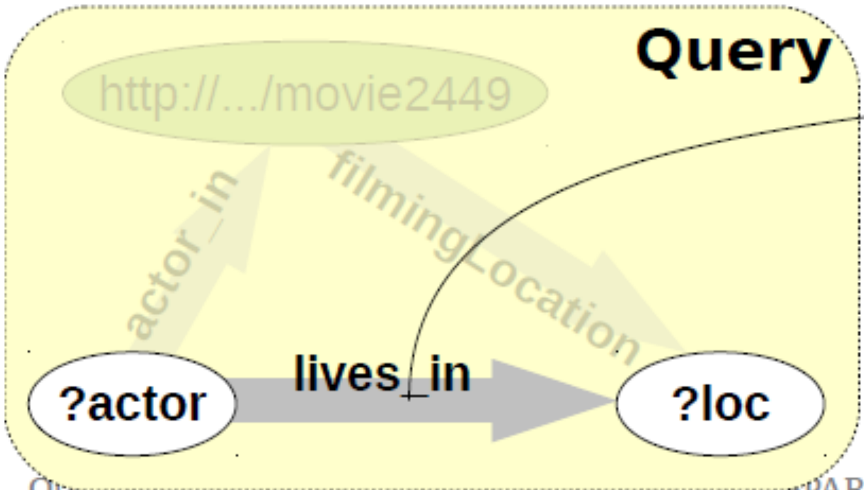


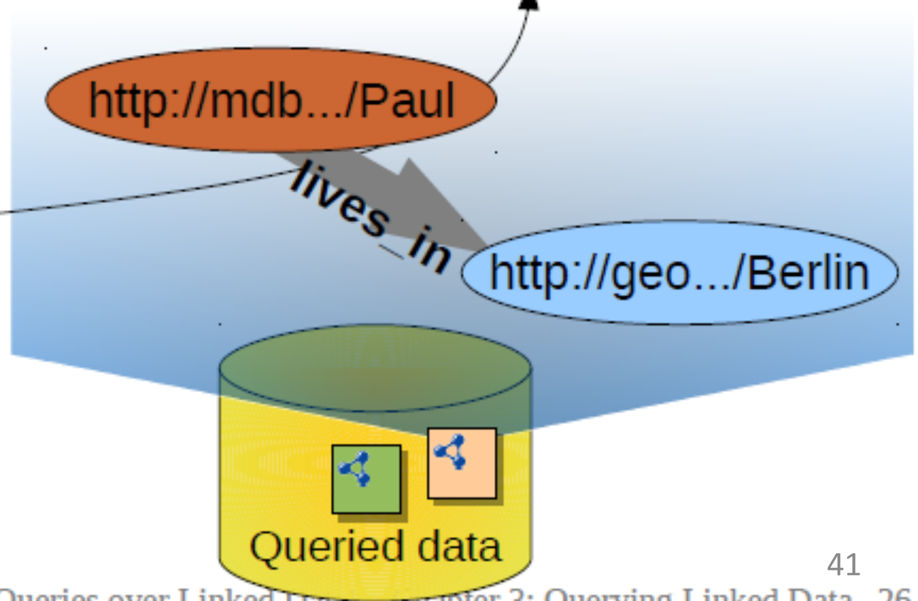
Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset

?actor
http://mdb.../Paul

?actor	?loc
http://mdb.../Paul	http://geo.../Berlin



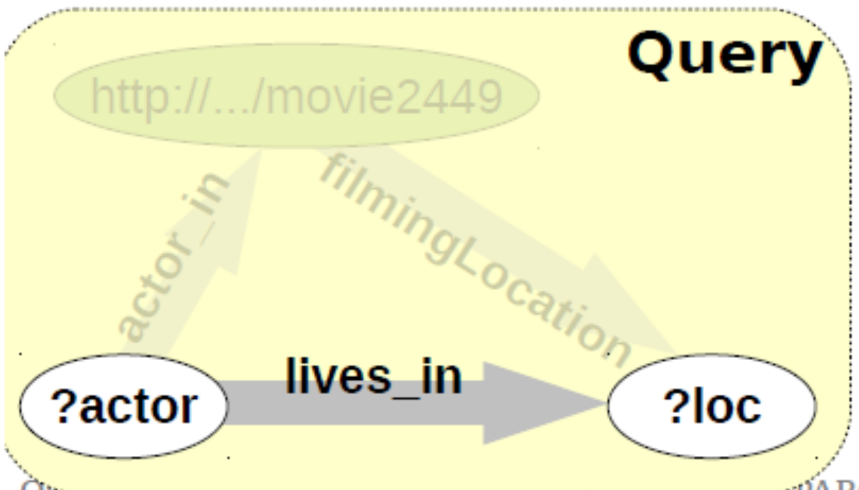


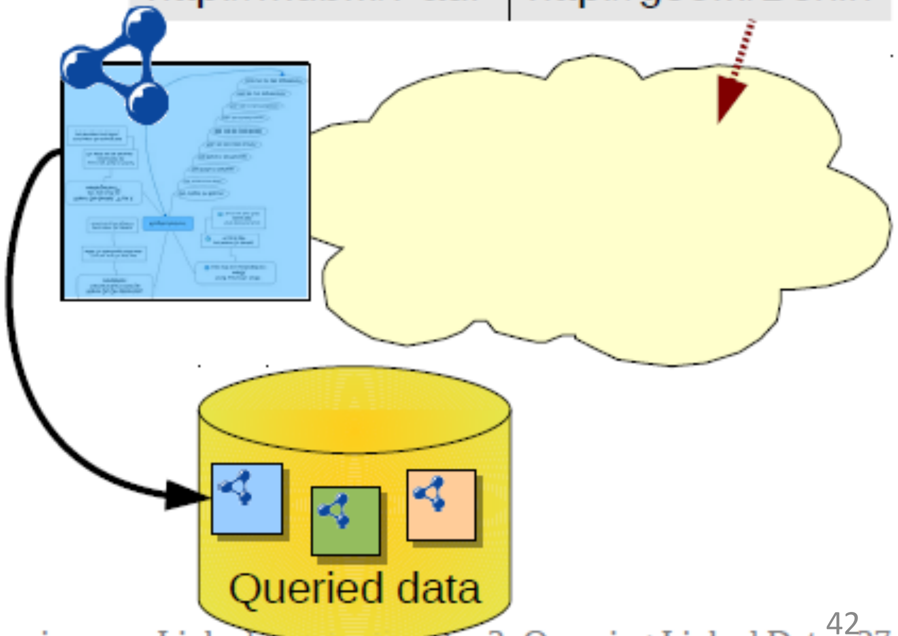
Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
  - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
  - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset

?actor
http://mdb.../Paul

?actor	?loc
http://mdb.../Paul	http://geo.../Berlin





Link traversal-based query execution

- Pros:
  - You might not know in advance the datasets to be queried
  - You tap the full potential of the Web
- Cons:
  - Slow: many navigations to get the RDF data sources related to a given resource (.e., describing a given resource)
  - Local RDF dataset incrementally generated
  - Very interesting from a research point of view, not very used in practice